# Contributions of hybrid architectures to depth imaging : a CPU, APU and GPU comparative study

Issam Said


**THÈSE DE DOCTORAT DE**
**l'UNIVERSITÉ PIERRE ET MARIE CURIE**

spécialité
**Informatique**

École doctorale Informatique, Télécommunications et Électronique (Paris)

présentée et soutenue publiquement par

**Issam SAID**

pour obtenir le grade de

**DOCTEUR** en **SCIENCES**
de l'**UNIVERSITÉ PIERRE ET MARIE CURIE**

# Apports des architectures hybrides à l'imagerie profondeur : étude comparative entre CPU, APU et GPU

Thèse dirigée par Jean-Luc Lamotte et
Pierre Fortin

soutenue le **Lundi 21 Décembre 2015**

après avis des **rapporteurs**

| | |
|---|---|
| M. François Bodin | Professeur, Université de Rennes 1 |
| M. Christophe Calvin | Chef de projet, CEA |

devant le **jury** composé de

| | |
|---|---|
| M. François Bodin | Professeur, Université de Rennes 1 |
| M. Henri Calandra | Expert en imagerie profondeur et calcul haute performance, Total |
| M. Christophe Calvin | Chef de projet, CEA |
| M. Pierre Fortin | Maître de conférences, Université Pierre et Marie Curie |
| M. Lionel Lacassagne | Professeur, Université Pierre et Marie Curie |
| M. Jean-Luc Lamotte | Professeur, Université Pierre et Marie Curie |
| M. Mike Mantor | Senior Fellow Architect, AMD |
| M. Stéphane Vialle | Professeur, SUPELEC campus de Metz |

*"If I have seen further it is by standing on the shoulders of Giants."*

— **Isaac Newton**

# *Acknowledgements*

Foremost, I would like to express my deepest thanks to my two directors, *Pr. Jean-Luc Lamotte* and *Dr. Pierre Fortin*. Their patience, encouragement, and immense knowledge were key motivations throughout my Ph.D. I thank them for having instructed me, for having guided me to compose this thesis, and for having steered me to the end. I thank them for having cheerfully assisted me in moments of frustration and of doubt. I would like to gratefully thank *Henri Calandra* for his trust, valuable recommendations, insights and useful discussions. He accompanied me throughout this adventure, I look forward to the next one.

I have been privileged and honored to have had my work been reviewed by *Pr. François Bodin* and *Dr. Christophe Calvin*. I am indebted to them for having taken the time out of their busy schedules, and for having spent efforts to go through, to give insightful comments, and to correct the ideas shared in this dissertation. I also thank the rest of the committee members: *Pr. Lionel Lacassagne*, *M. Mike Mantor* and *Pr. Stéphane Vialle* for having accepted to examine this dissertation and to give valuable feedbacks.

This work was a close partnership between Total, AMD, CAPS and Lip6. I would like to extend my gratitude to Total for funding this project. AMD is acknowledged for providing the hardware which was used at the heart of this work. CAPS is kindly acknowledged for the technical support and the sophisticated skills.

Immeasurable appreciation and deepest gratitude for the help and support are extended to the following persons, who in one way or another, have contributed in making this study possible. *Bruno Conche* for his endless support and logistic help, and more importantly for his energy that he did not spare in order to put this project together, without him it would not have been a success. *Bruno Stefanizzi* for having pointed me to the right way whenever I had questions or needed support from AMD. *Laurent Morichetti*, *Joshua Mora* and *Robert Engel*, each of whom had made each journey of mine to the AMD Sunnyvale campus, a pleasant experience. I thank them for sharing their experiences with me, and for finding answers to all of my questions. *Greg Stoner* and *Gregory Rodgers*, for the helpful discussions and the valuable information about the AMD hardware and software stack roadmaps. *Terrence Liao* and *Rached Abdelkhalek*, for their precious advices and for the brain storming sessions during my trips to Pau, France. *Harry Zong*, *Jing Wen*, *Donny Cooper*, *Matthew Bettinger* and *Russell Jones* for their precious help in setting up a remote work environment. *Romain Dolbeau*, for having put his rich technical expertise at my disposal. My fellow colleagues in the PEQUAN team with whom I have shared memorable moments throughout this experience. The administrative staff of the PEQUAN team is also kindly acknowledged for having taken care of my professional trips.

On a more personal note, words cannot express my gratitude for my parents, *Khalifa* and *Moufida*, the reason of what I become today thank you for your great support and continuous care. I profusely thank my brothers *Bilel* and *Zied* for being there for me no matter what. *Asma*, thank you for being supportive and for the great moments we spent together discussing this project. *Rached*, thank you for your hospitality, for the time, for the trips and for the laughs and fears we lived together. *Saber*, thank you immensely for your wise thoughtful advices, and for the numerous funny moments we spent together. I thank my cousins and friends (*Khaled*, *Yasser*, *Rafik*, *Haithem*, *Fethi*,

*Mohammed*, *Elkhal*, *John*, *Erick*, *Nefili*, *Layla*, *Justyna*, *Chris*, *Daniel*, *Jeaven*, *Binomi*, *Sahma*, *Saif*, *Sana*, *Sari*, *Hanen*, and the list goes on) for the precious moments of joy that were much needed during this journey. *Moktar* and *Souad*, I owe you a deep sense of gratitude for you unconditional love and attention. *Ludmila*, *Igor*, *Anna*, *Beji*, and *Lorita* thank you for your never ending support, and more importantly for the initiation to the Russian "banya". To my wonderful wife *Tatjana*, whose sacrificial care for me, her quiet patience, her tolerance to my occasional vulgar moods, and her unwavering love made it possible for me to finish this work, I express my genuine appreciation. Finally, I thank my son *Arsen*, a treasure from the Lord that was offered to me and to my wife in the middle of this adventure, and who was the source of my inspiration and of my greatest happiness.

# Contents

*This thesis is dedicated to all of my family*
*for their love, endless support*
*and encouragement.*
*To my cousin Marwen (1993-2014).*
*To my grandma Khadija (1930-2013).*

# Chapter 1

# Introduction

For more than 150 years, after the world's first commercial extraction of rock-oil by *James Miller Williams* in *Oil Springs (Ontario, Canada)* in 1858, exploration and mining companies worldwide face the challenge of finding new oil reserves in order to satisfy an ever-growing energy demand. The oil discovery rate had continued to grow, and had peaked in the 1960*s*. Since then, it has declined with each passing decade. The giant oilfields — a giant oilfield is defined as containing more than 500 million barrels of ultimately recoverable oil — are less and less discovered and their production is dropping throughout the years, as skewed in table 1.1. In sharp contrast, the average age of the world's 19 largest giant fields is almost 70 years, and 70% of the daily oil supply comes from oilfields that were discovered prior to 1970 [199].

Today, most of the shallow and easily accessible basins have already been found. The alternative way to find new deposits is to explore much deeper below the surface in hostile locations and extreme or challenging environments (such as deepwaters, frigid zones and hot dusty deserts), where accessing the explored area is often difficult, and where the complex geologic structures make it harsh to prospect the subsurface and to extract the hydrocarbon energy stored in the rocks. Therefore, the cost of drilling in these complex topographies is rising as the number of major new discoveries is decreasing. As a matter of fact, the cost of drilling an onshore well is about $3.5 to $4.5 million, that of an offshore well ranges between $23 and $68 million, while the cost of deepwater drilling, in a complex geology for example, can grow up to $115 million [36].

To face this challenge, Oil and Gas (O&G) firms are turning to modern exploration technologies that includes sophisticated survey techniques and cutting-edge science in

| *date of discovery* | *number of discoveries* | *average production per field (MMbbls)* |
|---|---|---|
| pre-1950s | 19 | 557 |
| 1950s | 17 | 339 |
| 1960s | 29 | 242 |
| 1970s | 24 | 236 |
| 1980s | 15 | 176 |
| 1990s | 11 | 126 |

TABLE 1.1: Statistics about the discoveries of giant oilfields until the 1990*s*, in terms of number and current production in Million Barrels (MMbbls). From [199].

order to glean the location and character of crude oil deposits while reducing the uncertainty of exploration, and thus improving drilling success rates. Indeed, the industry is developing new seismic acquisition techniques and new imaging technologies that provide vital information needed before drilling. More importantly, seismic imaging technologies help to remotely identify oil accumulations trapped tens of kilometers underground and undersea. The seismic acquisition is the process of sending acoustic waves through the subsurface and collecting the echoes reflected by the rock layers, and *seismic imaging* (or *depth imaging*) delineates the subsurface geologic structures from the collected data. Amongst the seismic imaging techniques, *Reverse Time Migration* (RTM) is by far the most famous computer based technique used in the industry because of the quality and integrity of the images it provides. O&G companies trust RTM with crucial decisions on drilling investments. However, RTM consumes prodigious amounts of computing power across extremely large datasets (tens of terabytes of data), which requires large memory capacities and efficient storage solutions. Throughout the last decades, theses heavy requirements have somewhat hindered its practical success.

Given the enormous amounts of data that must be processed, analyzed, and visualized in the least amount of time, O&G organizations are today leveraging High Performance Computing (HPC) technologies for seismic imaging, to stay ahead. In particular, organizations are deploying ever more powerful and highly honed computational workflows on a variety of HPC facilities. With the advances in processor technology over the past few years, today's HPC clusters are capable of providing petaflops[1] of compute capabilities and are slowly heading to the exascale era[2], making them an appropriate match for the challenges in the O&G industry. Today, it is not unusual for O&G companies to rely on clusters built around the latest multicore CPUs, with petabytes of storage and with the fastest-available network infrastructures, in order to spread workloads across an array of compute nodes [57, 145, 172]. Additionally, O&G exploration firms are trying to accelerate seismic processing workflows, such as RTM, by optimizing their increasingly sophisticated algorithms to take advantage of hardware accelerators, such as graphic processing units (GPUs) [23, 63, 147], field-programmable gate arrays (FPGAs) [63] and the Intel Xeon Phi processors [84]. GPUs are the most widely deployed, given the massively parallel nature of their architecture and hardware design which makes them a good fit for RTM (and similar algorithms such as Kirchhoff migration) workloads.

However, the deployment of seismic workloads on high-end CPU clusters and GPU based solutions have shown several restrictions in terms of performance, memory capacities and power consumption. The use of GPU technologies also introduces additional technical challenges to the table. Adapting seismic imaging applications to GPUs requires mastering novel programming models such as CUDA and OpenCL which may be considered as a difficult task for scientists (especially geophysicists) whose primary concern is introducing more physics and accuracy to the algorithms. Besides, unlike mainstream processors, a GPU acts like a co-processor in a system, and is interconnected to the main CPU via a PCI Express bus (gen 2 or gen 3). This implies particular manipulations, in terms of computations and memory management.

Recently, AMD has proposed the APU technology: a combination of a CPU and an integrated GPU on the same silicon die, and in a small power envelope. With the APUs,

[1] A petaflops is $10^{15}$ Flop/s, a Flop/s is a measure of computer performance that corresponds to the number of floating point operations a processor is capable to carry out per second.

[2] When HPC facilities will be able to achieve a performance at the order of exaflops ($10^{18}$ Flop/s).

AMD has introduced a new on-chip interconnect that puts together the CPU and GPU featuring a unified memory between the CPU and the GPU. Throughout this work, we therefore assess the relevance of APUs in the seismic workloads arena. By means of memory, applicative and power efficiency benchmarks as well as a CPU/APU/GPU comparative study on both the node level and the large scale level, we try to find out whether the APU technology can deliver a compromise solution, in terms of application performance, power efficiency and programming complexity, that is profitable and valuable in an O&G exploration context.

The first part of the dissertation is dedicated to a state of the art review. We start in chapter 2, by introducing the different stages of the modern seismic exploration chain. We emphasize the workflow of the seismic migrations, that of the RTM in particular. Then we present the mathematical tools and numerical methods that are used in a seismic exploration context. In chapter 3, we summarize the current advances in HPC in terms of hardware architectures, programming models and power consumption. We follow up in chapter 4, by giving an overview of state-of-the-art accelerated implementations of the stencil computations (an important building block of the seismic applications), as well as state-of-the-art accelerated RTM implementations and similar workflows.

The chapter 5 is a detailed description of the position of this thesis as well as a presentation of our contributions.

We follow up with the second part of this thesis, which is dedicated to our contributions. In chapter 6, we start with a thorough evaluation of the APU technology. The evaluation includes the assessment of the new memory model, a performance study and comparison between CPU, GPU and APU by means of applicative benchmarks, a power efficiency evaluation (where we describe our power measurement methodology), the feasibility of the hybrid utilization (CPU+GPU) of APUs, and a performance study of directive based implementations of the stencil computations. Then in chapter 7, we study the performance, power efficiency and the programmability of two seismic applications at the node level: the seismic modeling, which is considered as the first step of the seismic migration workflow, and the RTM. We conduct a performance and power efficiency comparisons between the CPU, GPU and APU to assess the relevance of APUs in this context. In the chapter 8, we extend our study to the large scale implementations of the seismic modeling and of the seismic migration (RTM) on a CPU cluster, on a GPU cluster and on an APU cluster. We try throughout this chapter to find out whether the RTM implementation on the APU cluster might be a valuable solution that addresses the limiting factors on the CPU and GPU based solutions. Finally, we conclude in chapter 9 with a mention to possible perspectives.

# Part I

# State of the art

# Chapter 2

# Geophysics and seismic applications

## Contents



Hydrocarbon exploration remains very challenging for the mainstream O&G industry. Substantial efforts are put to maximize the production of discovered reservoirs and explore new ones, albeit very rare (see figure 2.1). The industry relies on geophysics and more specifically on *seismic exploration* to transform vibrations, induced in the earth from various sources, into interpretable subsurface pictures and models. The pressing need for sharper and more informative structural images pushes the industry to permanently innovate and tackle a host of grand challenges in terms of seismic technology: to identify lithology (rock types), to infer petrophysical properties, to estimate the fluid content etc.

In this chapter, we present a brief overview of the basic goals and procedures for seismic exploration. We define the seismic migration operation since it is considered as the main imaging tool for petroleum deposits mapping, and we emphasize on the *Reverse Time Migration* (RTM) which is being increasingly used by the industry at the heart of a wide range of seismic imaging applications. We also review some fundamental mathematics associated with the wave propagation phenomenon, being the essential and mandatory tool to help understand the physics behind the seismic exploration.

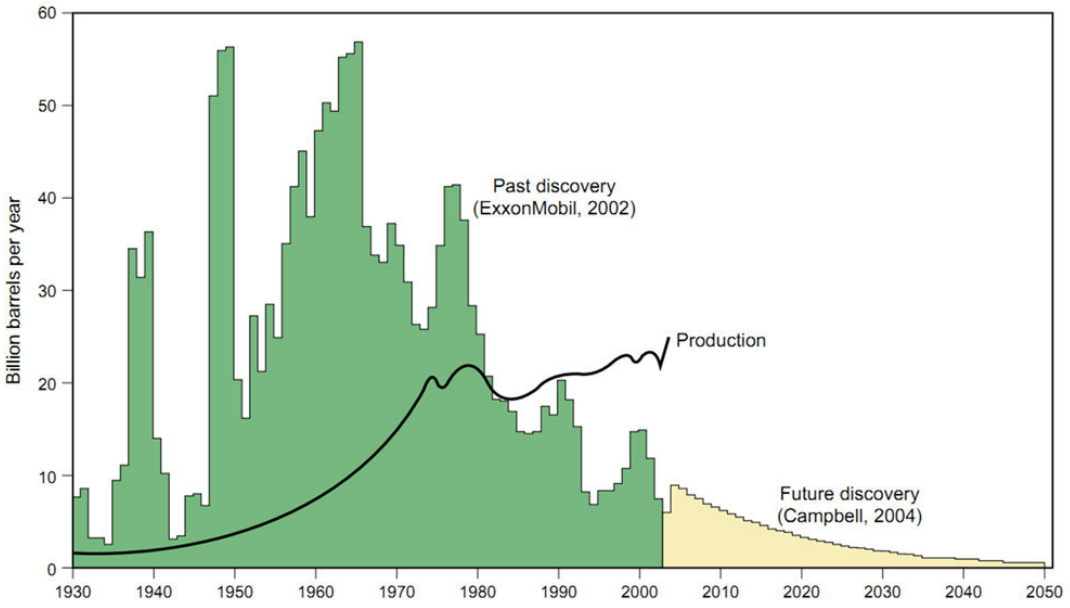



FIGURE 2.1: Oil discoveries and oil production, 1930 to 2050. Extracted from [25].

## 2.1 Introduction to seismic exploration

The principal goal of *seismic exploration*, more commonly referred to as *exploration geophysics* or also as *reflection seismology* and often abbreviated as *seismic*, is to obtain structural subsurface information from *seismic data* i.e. data collected by recording elastic or acoustic wave motion on Earth. Seismic is one of the geophysical methods, summarized in table 2.1, used in hydrocarbon exploration. The primary environments for seismic exploration are land (*onshore*), and marine (*offshore*). The land environment covers almost every type of terrain that exists on Earth, each bringing its own logistical problems. That includes jungle, desert, forest etc. The marine environment consists essentially of seas and oceans. However, there are also *transition zones* (TZ), i.e. the areas where the land meets the sea such as rivers, presenting unique challenges because the water is too shallow for large seismic vessels but too deep for the use of traditional methods of exploration on land.

| *Method* | *Measured parameter* | *Physical property* |
|---|---|---|
| Gravity | Spatial variations in the strength of the gravitational field of the Earth | Density |
| Magnetic | Spatial variations in the strength of the geomagnetic field | Magnetic susceptibility and remanence |
| Electromagnetic | Response to electromagnetic radiation | Electric conductivity/resistivity and inductance |
| Seismic | Travel times of reflected/refracted seismic waves | Seismic velocity (and density) |

TABLE 2.1: A summary of the geophysical methods used in hydrocarbon exploration. From the *University of Oslo, Department of Geosciences*.


FIGURE 2.2: The seismic exploration workflow.

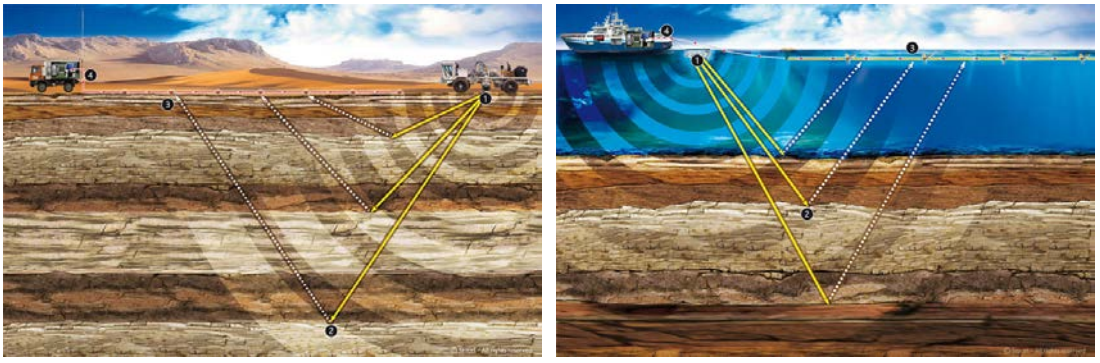
Seismic exploration allows the O&G industry to map out subsurface deposits of crude oil, natural gas, and minerals by seismically imaging the earth's reflectivity distribution. It is also used by petroleum geologists and geophysicists to interpret potential petroleum reservoirs, by extracting the seismic attributes out of the obtained images. The seismic exploration workflow, as described in figure 2.2, consists of three main stages: seismic acquisition, seismic processing and seismic interpretation[1]. For general informations about seismic exploration, the reader is kindly referred to Biondi [41], Coffeen [65], Sengbush [192] and Robein [181, 182].

### 2.1.1 Seismic acquisition

Seismic acquisition is the act of gathering data in the field, and making sure that it is of sufficient quality (this requires pre-processing such as noise attenuation and filtering). In seismic acquisition, an elastic or acoustic wavefield is emitted by a *seismic source* at a certain location at the surface. The reflected wavefield is measured by *receivers* located along lines (*2D seismics*) or on a grid (*3D seismics*). We refer to this process as a *shot experiment*. After each shot the source is moved to another location and the measurement is repeated. Figure 2.3 distinguishes between the land seismic acquisition (onshore) and the marine seismic acquisition (offshore). In land surveys, the seismic source can be a *vibroseis* or dynamite, the receivers are called *geophones* and are towed by trucks. In marine surveys, the source is often an air gun and the receivers are designated as *hydrophones* and are towed by vessels.

In order to collect data, many strategic choices have to be made. They are related to the physics and the location of the survey area, to the geometry of the acquisition

[1]This categorization is becoming more and more obsolete as technologies, that repeatedly iterate through those three stages, are emerging [179].



(a) Onshore seismic acquisition. (b) Offshore seismic acquisition.

FIGURE 2.3: Seismic acquisition steps at land (a) and at sea (b): 1) the seismic source emits controlled energy; 2) the seismic energy is transmitted and reflected from the subsurface layers; 3) the reflected energy is captured by receivers placed on the surface; 4) the acquisition systems record the data and pre-process it. From Sercel [193].

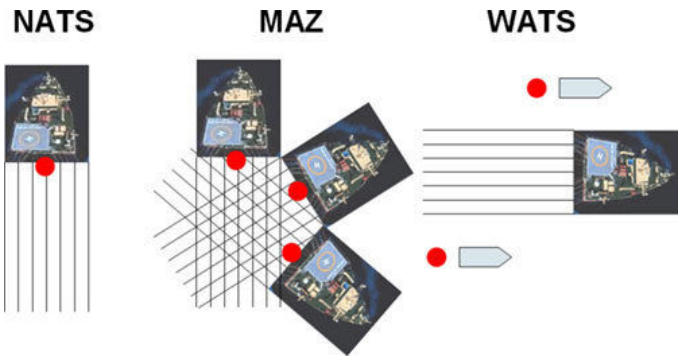


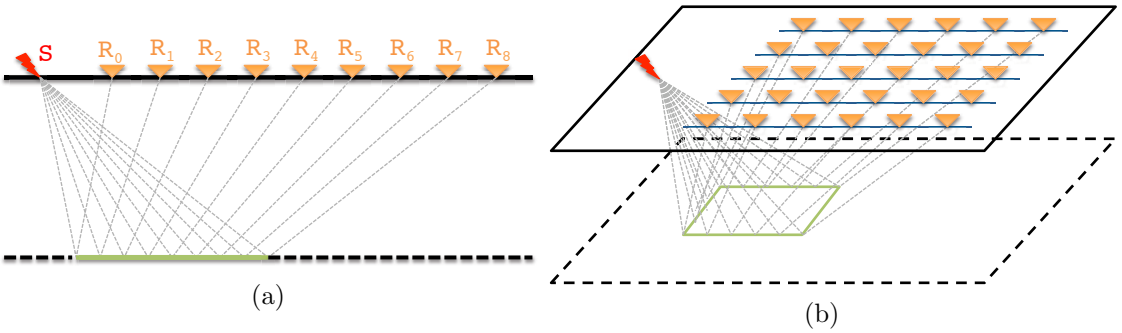

FIGURE 2.4: Seismic acquisition geometries: from left to right, Narrow Azimuth Towed Streamers, Multi-Azimuth, Wide Azimuth Towed Streamers. From PGS [174].

and to the accuracy of the targeted geophysical properties. These choices are often driven by economic considerations, since the cost of a survey may vary from $18.000 to $45.000 per square mile [21]. For example, specific acquisition parameters such as energy source effort and receiver station intervals, together with the data recording or *listening time*, have to be carefully defined. In addition, in the old days 2-D seismic reflection (see figure 2.5a) was the only tool for exploration because cost effective. Today, conventional 2-D seismic is only able to identify large structural traps while 3-D seismic (see figure 2.5b)[2] is able to pinpoint complex formations. Therefore, 3-D reflection has entirely replaced 2-D seismology in the O&G industry, albeit expensive. Furthermore, the acquisition geometry determines the coverage azimuth range and the consistency level of the illumination of reservoirs. Figure 2.4 represents schematic diagrams of the common acquisition geometries used in the O&G industry. The reader can find more detailed descriptions about the most common seismic acquisition geometries in [115]. Further, one can learn about cutting edge technologies in terms of seismic surveys, such as coil shooting in [101].

The basic principle of the seismic reflection is explained in figure 2.5. We differentiate between the 2-D seismic acquisition and the 3-D seismic acquisition, but the principle remains the same in the two cases. We activate a source ($S$) to send artificially-generated seismic waves into the subsurface. The waves get reflected off layer boundaries (called *reflectors* in the seismic literature). We record the arrival times and amplitudes of the reflected waves on the surface and detected by the receivers ($R_{0..n}$).

The size and scale of seismic surveys has increased alongside the significant concurrent increase in compute power during the last years. The collected data, i.e. *seismic traces* (see figure 2.6), is often humongous and was stored, in the past, in *tapes* and was very hard to process by computers. Each seismic trace corresponds to a seismic signal detected by one receiver throughout time. A wide variety of seismic data formats were proposed to digitize the seismic data and standardize its manipulation; the most famous ones in the industry are SEGY [191], SEP [200], SU [66] and RSF [148], to name a few. So far, the choices of seismic survey parameters such as the shot position (the position of the seismic source), the shot interval (the distance between two successive seismic perturbations), the receiver interval (the distance that separates two successive receivers situated in the same streamer), the shooting frequency (the frequency of activating the seismic source), etc. are of prime importance as they make immediate impact on the

[2]This is only a simplified illustration of the 3-D seismic reflection. In the industry, more than one seismic source is required to conduct a 3-D survey.



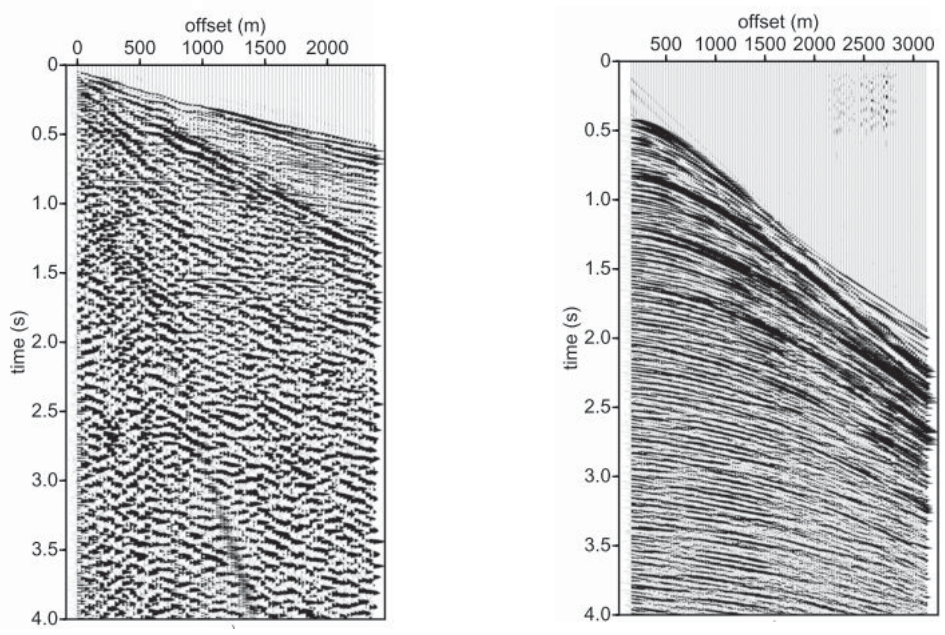

(a) (b)

FIGURE 2.5: Seismic surveys in 2-D (a) and in 3-D (b). The seismic source is the red sign. Receivers are the orange triangles. Dotted black lines are basic representations of the subsurface reflectors. Green lines represent the covered area. Dashed gray lines illustrate the wave energy paths. The blue lines (in b) are called *streamers*.

generated seismic traces which are used in the following stages of the seismic exploration cycle.

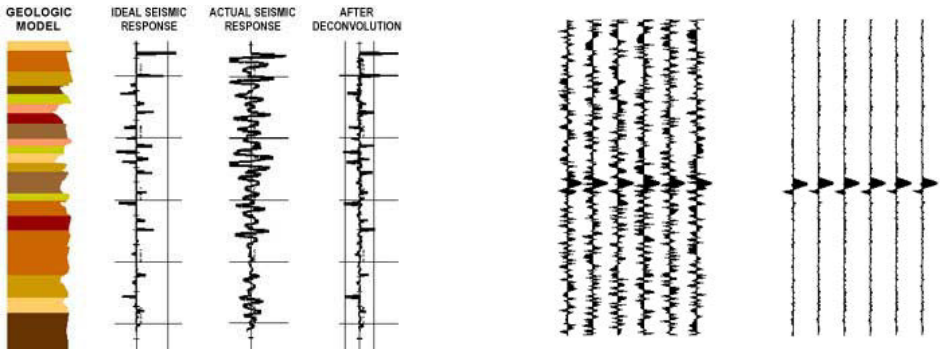
### 2.1.2 Seismic processing

In the seismic processing stage, we want to manipulate the gathered data, after acquisition, such that we generate an accurate image of the subsurface. A long run separates the raw data from being transformed into structural pictures. Processing consists of the application of a chain of computer treatments to the acquired data, guided by the hand




(a) From a land survey. (b) From a marine survey.

FIGURE 2.6: Example of seismic traces. Each wiggle is an illustration of the evolution of the wave amplitude, as well as the wave travel time, as a function of the "offset" (in meters) throughout time (in seconds) as measured by a given receiver. The offset is the distance between each receiver and the seismic source. Source *Drijkoningen, TU Delft* [78].




(a) Signal deconvolution. (b) Seismic traces stacking.

FIGURE 2.7: Signature deconvolution and stacking. Source *CGG*.

of processing geophysicists. There is neither a standard classification nor an order to define theses operations because they depend on the nature of the collected data, in the one hand, and because processing is a subjective manipulation, in the other hand. We try, throughout this section, to describe the most relevant processing routines and leave the opportunity to the reader to dive into the geophysics literature [59, 127, 143, 181], in order to learn more about seismic processing.

To begin with, the reflected seismic response can be a mixture of the seismic source pulse, the effect of the Earth upon that pulse, and background noise, all convolved together. The data is usually cleaned up from those spurious signals that might have been accumulated during seismic surveys. For instance, the seismic source may introduce signals, into the Earth, to which the underlying structures remain irresponsive because they do not depend on the signal put in. Those signals have to be removed. This is considered as *pre-processing* or *data conditioning*, and usually includes signal processing techniques, such as *signal deconvolution* and *anti-aliasing filtering*. Figure 2.7a shows an example of a seismic trace after applying a signal deconvolution.

Besides, seismic traces are usually sorted and those that share the same geometry properties are *stacked*, i.e. the signals are summed, to attenuate the background noise and thus increase the signal-to-noise ratio. The more seismic traces we can stack together into one seismic trace, the clearer is the seismic image. Stacking can be done by putting together traces from the same reflecting point (*Common Reflection Point (CRP) stacking* or *CRP gather*), from the same shot position *(Common Shot Gather (CSG))*, from the same midpoint (*Common Midpoint (CMP) stacking*) or from the same depth point (*Common Depthpoint (CDP) stacking*)[3] etc. [59]. Figure 2.7b emphasizes the noise attenuation after the CRP stacking of six seismic traces.

Furthermore, before arriving at the receivers the seismic energy may be reflected a number of times: this is known as the *multiple reflections* phenomenon (see figure 2.8) as opposed to *primary reflections*. For example, during offshore surveys, the energy bouncing back-and-forth within the water produces false reflections and obscures the real data. Multiple attenuation is needed to remove multiples embedded in the data without interfering with primary events. This is referred to as *Demultiple* in the literature, and many advanced numerical algorithms are proposed to do so, such as *Surface-Related Multiple Elimination* (SRME) [144]. Note that some research, such as Guitton [99], focus on imaging the multiples and integrating them to the primary reflections rather

[3] In the case where the reflectors are horizontal, CDP is equivalent to CMP stacking.

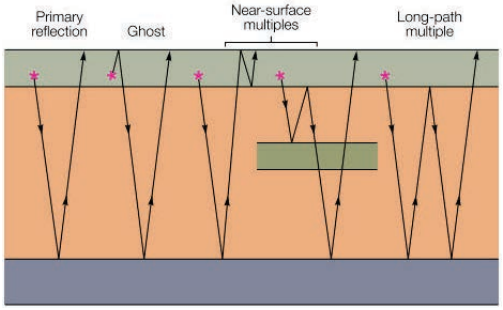



FIGURE 2.8: Illustration of the multiple reflections. Source Ashton C. et al. [35].

than removing them. Another seismic processing is seismic traces interpolation. This manipulation is used to enhance the energy and highlight the areas close to the subsurface reflectors. Any missing seismic trace is filled in by signal interpolation.

At this point, the data is ready to more advanced processing operations such as *seismic imaging* or *inversions* [44]. The main goal of seismic imaging is to transform the pre-processed seismic traces to the most accurate possible graphical representation of the Earth's subsurface geologic structure. A key point in imaging is that the reflected wave is proportional to the amplitude of the incidence wave. The proportionality coefficient is called the *reflection coefficient*. Imaging has the objective of computing this reflection coefficient. Hence, the final image is a representation of the reflection coefficient at each point of the subsurface. This can be performed by means of *seismic migration*.

Migration is using the two-way travel time, amongst other attributes provided by seismic traces, to place (or migrate) the dipping temporal events in their true subsurface spatial locations. Processing these reflections produces a synthetic image of the subsurface geologic structure. We show in figure 2.9 an example of a seismic processing chain. The traces in 2.9a are subject to a water bottom multiple reflection (arrowed). In 2.9b, it is removed by demultiple and the image shows the result of suppressing the water bottom multiple. The seismic traces are, then, enhanced by interpolation in 2.9c. Finally, the image, in 2.9d most closely resembles the true sub-surface geology. It is obtained after seismic migration. More advanced processing techniques, such as *Prestack*

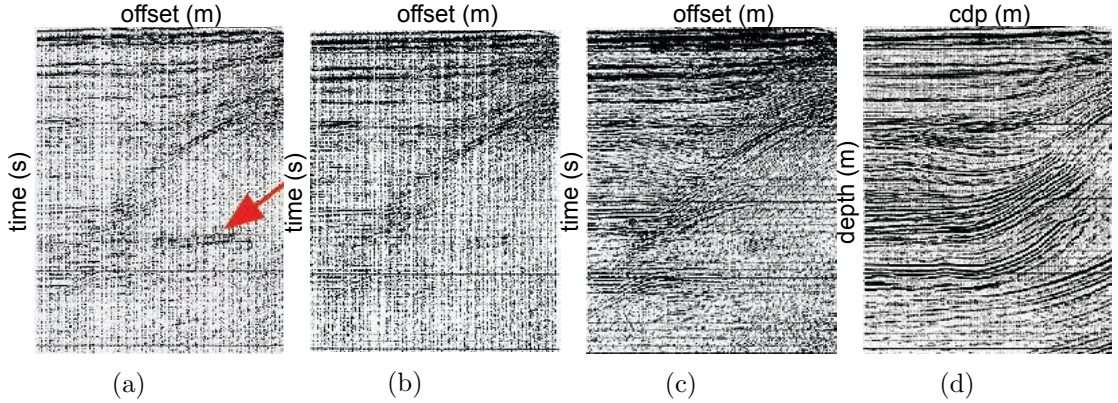



FIGURE 2.9: The result of a sequence of seismic processing algorithms. (a) represents the raw traces. From (a) to (b) demultiple is applied. From (b) to (c) interpolation is performed. From (c) to (d) seismic migration is used to produce the final subsurface image. Source *CGG*.

*Depth Migration* (PSDM), can significantly improve seismic imaging, especially in areas of complex geology. Finally, we recall the seismic processings are numerous and require advanced mathematical algorithms. Those are often applied to 3D seismic data which require enormous computing resources. Not to mention the massive volumes of data involved.

### 2.1.3 Seismic interpretation

The final stage of the seismic exploration cycle is seismic interpretation. The purpose of interpretation is to interpret the processed seismic images and integrate other geoscientific information in order to make assessments of where the O&G reservoirs may be accumulated and to learn about their characterization. Interpreters or *interpretation geophysicists*, are involved at this stage to analyse the seismic data. Relevant information consist of structures and features which can be related to geological phenomena such as faults, fractures, anticlines etc. This can deliver valuable insights about the nature of rocks, about which time they were formed and about their environment.

Computer algorithms are used to help interpret seismic data. For instance, numerical algorithms are used for the calculation of seismic attributes such as amplitude, phase and frequency based on the migrated seismic image. In practice, the seismic attributes (especially the amplitude) are related to the subsurface reflectivity which in turn provides information about the rock and the pressure-formation. Other seismic attributes are used in interpretation, namely *coherence*, *dip* and *azimuth*, and *gradient correlation cube*. For instance, the coherence is an attribute that measures the continuity between seismic traces in a specified window, applied on a seismic section. Figure 2.10 shows a composite of a section of a 3D seismic cube and a section of the corresponding coherence cube. For other examples of attributes calculation used in the interpretation stage we refer the reader to Abdelkhalek [22].

## 2.2 Seismic migrations and Reverse Time Migration (RTM)

In section 2.1 we have mentioned that seismic migration is classified as a final processing step in order to generate structural pictures of the subsurfaces. It is in fact the most important routine of the whole processing flow. In this section, we give a short overview of seismic migrations in general. We particularly insist on the Reverse Time Migration (RTM), where we describe the components of its workflow along with its advantages compared with the conventional migration techniques.

### 2.2.1 Description and overview of migration methods

The purpose of migration is to reconstruct the reflectivity distribution of the buried structures on Earth, from the seismic data recorded at the surface. For that to do, reflections events (especially non *zero-offset* reflections) are collapsed and moved, i.e. migrated, to their proper spatial location. Schustler [188] explains how seismic traces are migrated, and enumerates the challenges that might be related to migration such as diffraction, out-of-plane reflections and conflicting dips.

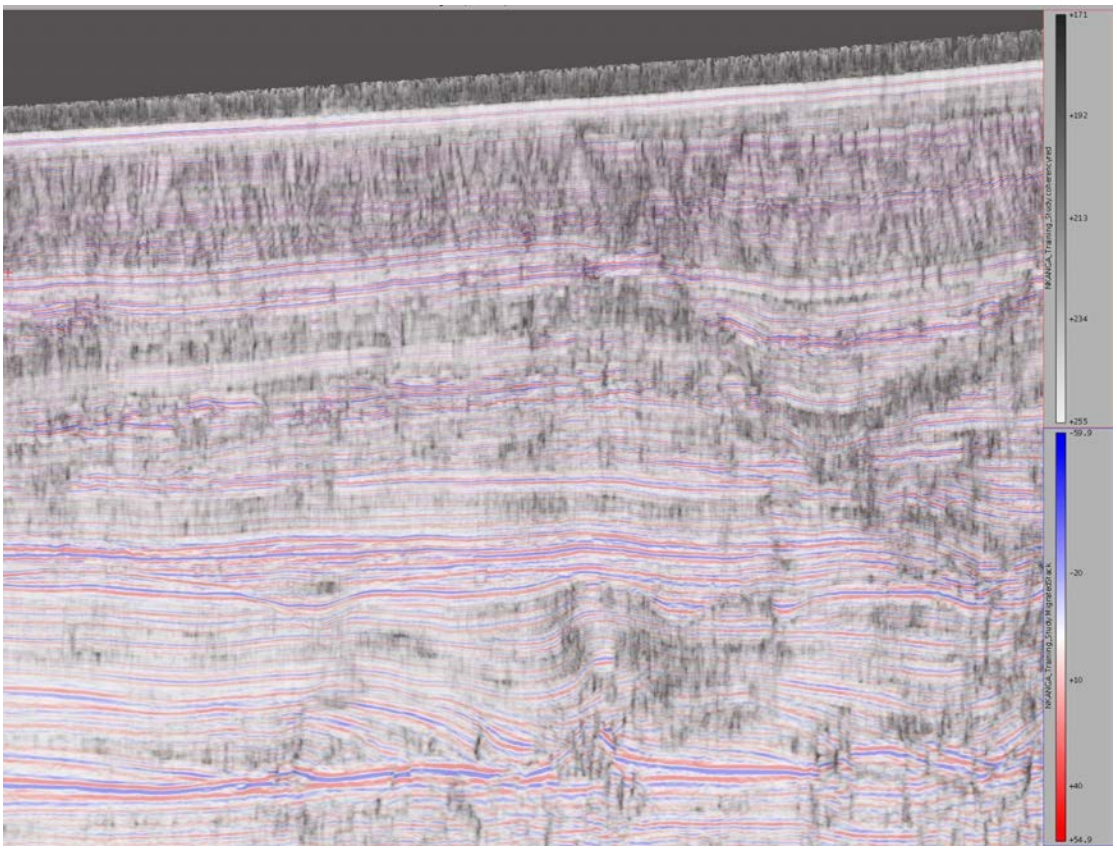
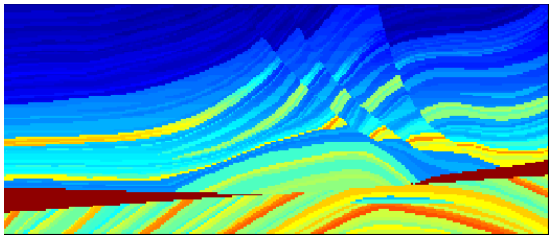
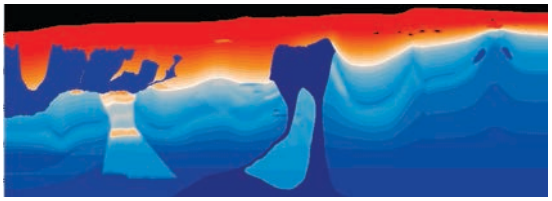


FIGURE 2.10: A seismic section (colored) superposed by its corresponding coherence attribute section (grayed). The color bar is the amplitude and the gray scale is to evalute the coherence. Courtesy of Abdelkhalek, Total S.A [22].

Migration relies upon pre-processed input data (seismic traces) and an accurate *velocity model*. Synthetic velocity models were proposed (see figure 2.11) by the O&G community in order to validate migration algorithms and display their potential power for imaging complex structures. However, in the case of real data, the velocity model of the subsurface is unknown. As a matter of fact, migration relies on various velocity estimation procedures, e.g. iterative prestack migration [42], to aid in imaging. In other words, migration is also a velocity analysis tool. Conceptually, migrations can be categorized with respect to different parameters. From a dimensionality perspective, migration is either *2D* or *3D*. 3D migration requires data to be acquired in 3D and presents rich azimuth content.
From data stacking standpoint, migration can be *prestack* or *poststack*. In poststack



(a) The Marmousi model [116].



(b) The BP 2004 model [190].

FIGURE 2.11: Examples of synthetic velocity models provided by the O&G community.

migration, the seismic traces are stacked in bins, each of which is reduced to only one seismic trace. This is much less expensive to process but is also less accurate. In prestack migration, traces are not stacked and every single trace is processed which require huge computational effort.
Furthermore, we can categorize migrations upon wether they support or not lateral velocity variations. *Time* migration is insensitive to lateral variation of the velocities and is more appropriate to constant and depth dependent velocities. In the contrary, *depth* migration can handle strong variations of the velocities and is thus more appropriate for complex geological structures.
Mathematically, we can split migrations into two categories. The first one is *Ray-based* migrations, such as *Kirchhoff* migration and *Beam* migration [105]. The second is *Wave-field extrapolation* migrations, such as *One-way* migration and *Two-way* migration [159].

Historically, migration was achieved by graphical methods in the 1960's [194]. This was followed by diffraction summations. In the 1970's, several important developments took place. Based on the pioneering work of Jon Claerbout [59, 60], migration methods based on wave theory were developed. Claerbout derived migration as a finite-difference solution of an approximate wave equation. Kirchhoff wave-equation migration (Schneider [186], Gardner [89]), and frequency-wavenumber migrations (Gazdag [90] and Stolt [202]) appeared shortly thereafter. Those were initially time migration methods, but due to the pressing need for more accuracy they were changed into depth migrations. In the early 1980's, Baysal et al. [37] along with Whitmore [221] and McMechan [151], proposed the Reverse Time Migration, based on the exact wave equation. The last twenty years have seen extensions of these methods to three dimensions and to prestack migration, and enhancements of their efficiency and accuracy. For further reading about migrations, we refer to [42, 94].

### 2.2.2 Reverse Time Migration

RTM is a two-way wave equation based pre-stack or post-stack depth migration. RTM is becoming more and more important as a tool of seismic imaging in the O&G industry. If the velocity model is complex or is subject to strong velocity gradients, such complexities will produce turning (or diving) rays and multiples when using conventional migration techniques (detailed in [188]). The RTM addresses these issues by directly using the two-way wave equation without any approximations or assumptions. The workflow of the RTM technique is depicted in the flowchart 2.12. Note that we do not mention in the figure that RTM also needs a velocity model as an input and that this workflow is repeated for each shot experiment. First, the source wavefield, i.e the wavefield whose origin is the seismic source, is propagated forward in time (we refer to this stage as *forward modeling* or also *seismic modeling*). Then, the receiver wavefield, i.e. a wavefield that is incident from the receivers, is then propagated back in time (this phase is called *backward modeling* or *retro-propagation*). Finally, the imaging condition is applied with respect to Claerbout's [58] imaging principle: "a reflector exists where the source and the receiver wavefields coincide in time and space".
As a matter of fact, the source wavefield and the receiver wavefield are cross-correlated throughout time. We denote $I(x, y, z)$ the reflectivity coefficient of the subsurface, i.e. the resulting seismic image, at the coordinate $(x, y, z)$. The source wavefield is presented by a $(\mathbb{R}^3, \mathbb{N}) \to \mathbb{R}$ function $S(x, y, z, t)$ and the receiver wavefield by a similar function $R(x, y, z, t)$, each at the coordinate $(x, y, z)$ and at time $t$. We can identify the RTM as

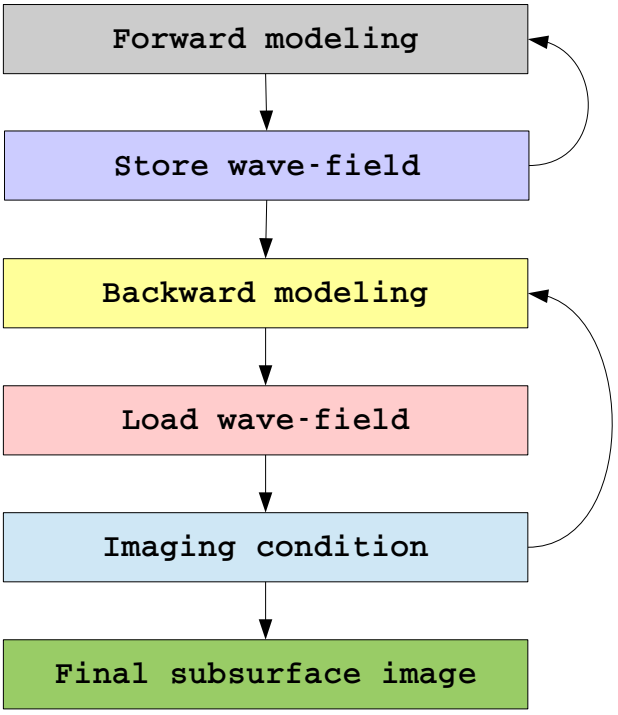



FIGURE 2.12: The Reverse Time Migration flowchart.

the linear operator described in the equation (2.1).

$$I(x,y,z) = \sum_{shot}\sum_{t} S_{shot}(x,y,z,t) * R_{shot}(x,y,z,t) \tag{2.1}$$

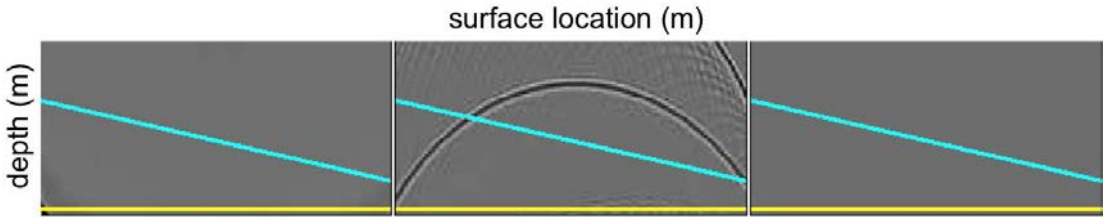
However, in some cases especially for large impedance contrasts and complex geological structures, the source and receiver wavefields can not be serrated efficiently. In these cases, the cross-correlation described in equation (2.1) leads to low frequency artefacts and illumination effects [123].
In order to eliminate the illumination effects, the image is often divided, after cross-correlation, by the source illumination (see equation (2.2)), or by the receiver illumination (see equation (2.3)), or even better by a combination of both source illumination and receiver illumination (see equation (2.4)). This calculation corresponds to the imaging condition of the RTM algorithm.

$$I(x,y,z) = \frac{\sum_{shot}\sum_{t} S_{shot}(x,y,z,t) * R_{shot}(x,y,z,t)}{\sum_{shot}\sum_{t} S^2_{shot}(x,y,z,t)} \tag{2.2}$$

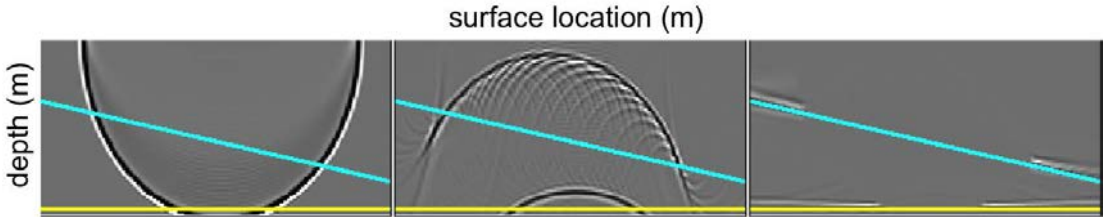
$$I(x,y,z) = \frac{\sum_{shot}\sum_{t} S_{shot}(x,y,z,t) * R_{shot}(x,y,z,t)}{\sum_{shot}\sum_{t} R^2_{shot}(x,y,z,t)} \tag{2.3}$$

$$I(x,y,z) = \frac{\sum_{shot}\sum_{t} S_{shot}(x,y,z,t) * R_{shot}(x,y,z,t)}{\sum_{shot}\sum_{t} S^2_{shot}(x,y,z,t)} + \frac{\sum_{shot}\sum_{t} S_{shot}(x,y,z,t) * R_{shot}(x,y,z,t)}{\sum_{shot}\sum_{t} R^2_{shot}(x,y,z,t)} \tag{2.4}$$

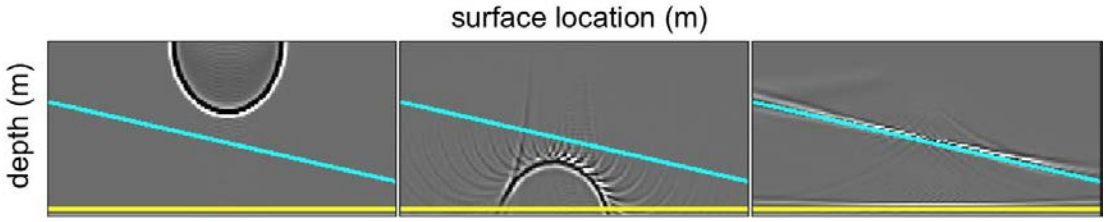
In the scope of this work we make use of the imaging condition defined in (2.2). We show an example of the RTM technique in figure 2.13, where we present the three different steps along with the resulting seismic image (see the dipping reflector).




(a) Snapshot at $t = 1.20$ $s$ of the source wavefield (left), the receiver wavefield (middle) and the image progression (right). No reflector is imaged yet.




(b) Snapshot at $t = 0.75$ $s$ of the source wavefield (left), the receiver wavefield (middle) and the image progression (right). The bottom reflector is almost fully imaged.




(c) Snapshot at $t = 0.30$ $s$ of the source wavefield (left), the receiver wavefield (middle) and the image progression (right). All reflectors (the bottom and the shallow) are fully imaged.

FIGURE 2.13: A Reverse Time Migration example: the source and receiver wavefields are correlated, at three subsequent time-steps, in oder to image two reflectors. Source: Biondi [41].

## 2.3 Numerical methods for the wave propagation phenomena

Most differential equations are much too complicated to be solved analytically, thus the development of accurate numerical approximation schemes is essential to understand the behavior of their solutions. The wave equation, being a Partial Differential Equation (PDE), is no exception. This section presents an overview of the state-of-the-art numerical methods used for seismic modeling and seismic imaging. Given that RTM is based on the wave equation, we present the general equations that govern the propagation of waves in elastic and acoustic media. These methods were widely studied for seismic imaging and one can find more details in Virieux et al. [215] and in Carcione et al. [50].

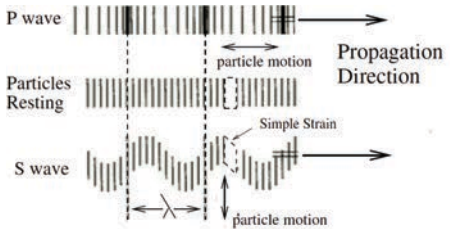



FIGURE 2.14: Particle motions for P (top) and S (bottom) waves. $\lambda$ is the wavelength and the simple strain illustrates a unit deformation. From [189].

### 2.3.1 The wave equation

#### 2.3.1.1 Seismic waves and propagation media

Before introducing the theory that governs the wave propagation phenomenon, we briefly recall the type of seismic waves and the nature of propagation media. A wave propagation is called *elastic* when the traversed medium can change in shape as a result of a deforming force otherwise the propagtion is *acoustic*. If the medium has constant density, we call it *homogeneous*, *heterogeneous* if it has not. Besides, we call a medium *isotropic* if it has the same physical characteristics independently of directions. In the contrary, the medium is called *anisotropic*.
The seismic waves are either *body* waves, that is they travel through the interior of the Earth, or *surface* waves if they travel along the Earth's surface. We distinguish two types of body waves: *Compressional* waves, also referred to as *Primary (P)* waves[4], and *Shear* waves, also called *Secondary (S)* waves. Figure 2.14 illustrates the propagation directions of P and S waves for small elemental volumes (*particles*). P waves propagate in parallel with the particle motion whereas S waves propagate perpendicularly to the particle motion. In homogeneous and isotropic media the velocities of P and S waves are, respectively, $V_p = \sqrt{\frac{\lambda+2\mu}{\rho}}$ and $V_s = \sqrt{\frac{\mu}{\rho}}$, where $\lambda$ and $\mu$ are the *Lamé* parameters, and $\rho$ is the density. Note that shear waves do not propagate in acoustic media (water for example) because the shear modulus $\mu$ is null in fluids.

#### 2.3.1.2 The elastic wave equation

The general wave equation is established using the Newton's second law of motion and Hook's law, with some constraints considered: the media is elastic, isotropic and subject to infinitesimal displacements in order to satisfy the elasticity condition. For the sake of simplicity the motion of the wave is initially presumed to be one dimensional, the wave equation will be later derived to the three dimensional case. We denote the particle displacement $\eta$, the dimension of the wave motion $Z$, and the particle position is given by the $z$ coordinate. Newton's law (2.5), for small elemental volumes, states that the acceleration ($\gamma$) of a particle when multiplied by its mass ($m$) is equal to the sum of forces applied on it ($f$).

$$f = m\gamma \tag{2.5}$$

Considered that the pressure ($p$) is the force on an object that is spread over a surface area and given that the particles are infinitesimal (we consider the unit surface), the

[4]Note that there are other types of wave, i.e. Love waves and Rayleigh waves, which are surface waves that we deliberately ignore here.

force is equivalent to the pressure differential. Similarly, here we consider studying the wave kinematics in a unit volume thus the mass can be replaced by the density ($\rho$). Note that the variables in the equation are scalar fields since we consider 1D case and that in the 3D case vector fields should be considered instead. The equation (2.5) is then equivalent to the formula (2.6).

$$\frac{\partial \Delta p}{\partial z} = \rho \frac{\partial^2 \eta}{\partial t^2} \tag{2.6}$$

Hook's law (2.7) states that the *strain* (deformation) of an elastic object or material is proportional to the *stress* applied to it.

$$\Delta p = K \frac{\partial \eta}{\partial z}, \tag{2.7}$$

where $K$ is the *Bulk* modulus. The wave equation is thus derived as follows:

$$K \frac{\partial^2 \eta}{\partial^2 z} = \rho \frac{\partial^2 \eta}{\partial t^2} \tag{2.8}$$

Extending the equation to the 3D cartesian coordinate system $(X, Y, Z)$, and using the indicial notation implies the system of equations (2.9) and (2.10) [24, 38]:

$$\begin{aligned}
\frac{\partial \sigma_{xx}(x,y,z,t)}{\partial t} &= (\lambda(x,y,z) + 2\mu(x,y,z)) \frac{\partial v_x(x,y,z,t)}{\partial x} + \\
&\lambda(x,y,z) \left( \frac{\partial v_y(x,y,z,t)}{\partial y} + \frac{\partial v_z(x,y,z,t)}{\partial z} \right) \\
\frac{\partial \sigma_{yy}(x,y,z,t)}{\partial t} &= (\lambda(x,y,z) + 2\mu(x,y,z)) \frac{\partial v_y(x,y,z,t)}{\partial y} + \\
&\lambda(x,y,z) \left( \frac{\partial v_x(x,y,z,t)}{\partial x} + \frac{\partial v_z(x,y,z,t)}{\partial z} \right) \\
\frac{\partial \sigma_{zz}(x,y,z,t)}{\partial t} &= (\lambda(x,y,z) + 2\mu(x,y,z)) \frac{\partial v_z(x,y,z,t)}{\partial z} + \\
&\lambda(x,y,z) \left( \frac{\partial v_x(x,y,z,t)}{\partial x} + \frac{\partial v_y(x,y,z,t)}{\partial y} \right) \\
\frac{\partial \sigma_{xy}(x,y,z,t)}{\partial t} &= \mu(x,y,z) \left( \frac{\partial v_x(x,y,z,t)}{\partial y} + \frac{\partial v_y(x,y,z,t)}{\partial x} \right) \\
\frac{\partial \sigma_{xz}(x,y,z,t)}{\partial t} &= \mu(x,y,z) \left( \frac{\partial v_x(x,y,z,t)}{\partial z} + \frac{\partial v_z(x,y,z,t)}{\partial x} \right) \\
\frac{\partial \sigma_{yz}(x,y,z,t)}{\partial t} &= \mu(x,y,z) \left( \frac{\partial v_y(x,y,z,t)}{\partial z} + \frac{\partial v_z(x,y,z,t)}{\partial y} \right)
\end{aligned} \tag{2.9}$$

$$\begin{aligned}
\frac{\partial v_x(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)} \left( \frac{\partial \sigma_{xx}(x,y,z,t)}{\partial x} + \frac{\partial \sigma_{xy}(x,y,z,t)}{\partial y} + \frac{\partial \sigma_{xz}(x,y,z,t)}{\partial z} \right) \\
\frac{\partial v_y(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)} \left( \frac{\partial \sigma_{xy}(x,y,z,t)}{\partial x} + \frac{\partial \sigma_{yy}(x,y,z,t)}{\partial y} + \frac{\partial \sigma_{yz}(x,y,z,t)}{\partial z} \right) \\
\frac{\partial v_z(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)} \left( \frac{\partial \sigma_{xz}(x,y,z,t)}{\partial x} + \frac{\partial \sigma_{zy}(x,y,z,t)}{\partial y} + \frac{\partial \sigma_{zz}(x,y,z,t)}{\partial z} \right)
\end{aligned} \tag{2.10}$$

where $v_x(x,y,z,t)$, $v_y(x,y,z,t)$ and $v_z(x,y,z,t)$, are the components of the particles velocity vector at time $t$; $\sigma_{ij}(x,y,z,t)$ with $i,j \in (x,y,z)^2$ are the stress tensor components

at time $t$ (note that the tensor is symmetric, i.e. $\sigma_{xz} = \sigma_{zx}$); $\rho(x, y, z)$ is the density of the medium; $\lambda(x, y, z)$ and the *shear modulus* $\mu(x, y, z)$ are the Lamé parameters that describe the linear-stress relation [196].

#### 2.3.1.3 The acoustic wave equation

The acoustic approximation states that shear effects in the data are negligible and that the dominant wave type is a compressional wave. Thus the shear modulus $\mu(x, y, z)$ is null. The equations (2.9) and (2.10) are simplified as follows:

$$\begin{aligned}
\frac{\partial \sigma_{xx}(x,y,z,t)}{\partial t} &= \lambda(x,y,z)\left(\frac{\partial v_x(x,y,z,t)}{\partial x} + \frac{\partial v_y(x,y,z,t)}{\partial y} + \frac{\partial v_z(x,y,z,t)}{\partial z}\right) \\
\frac{\partial \sigma_{yy}(x,y,z,t)}{\partial t} &= \lambda(x,y,z)\left(\frac{\partial v_y(x,y,z,t)}{\partial y} + \frac{\partial v_x(x,y,z,t)}{\partial x} + \frac{\partial v_z(x,y,z,t)}{\partial z}\right) \\
\frac{\partial \sigma_{zz}(x,y,z,t)}{\partial t} &= \lambda(x,y,z)\left(\frac{\partial v_z(x,y,z,t)}{\partial z} + \frac{\partial v_x(x,y,z,t)}{\partial x} + \frac{\partial v_y(x,y,z,t)}{\partial y}\right)
\end{aligned} \tag{2.11}$$

$$\begin{aligned}
\frac{\partial v_x(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)}\frac{\partial \sigma_{xx}(x,y,z,t)}{\partial x} \\
\frac{\partial v_y(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)}\frac{\partial \sigma_{yy}(x,y,z,t)}{\partial y} \\
\frac{\partial v_z(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)}\frac{\partial \sigma_{zz}(x,y,z,t)}{\partial z}
\end{aligned} \tag{2.12}$$

The equation (2.11) implies that $\frac{\partial \sigma_{xx}(x,y,z,t)}{\partial t} = \frac{\partial \sigma_{yy}(x,y,z,t)}{\partial t} = \frac{\partial \sigma_{zz}(x,y,z,t)}{\partial t}$, which can lead to the hyperbolic $1^{st}$ order system:

$$\begin{aligned}
\frac{\partial p(x,y,z,t)}{\partial t} &= K(x,y,z)\left(\frac{\partial v_x(x,y,z,t)}{\partial x} + \frac{\partial v_y(x,y,z,t)}{\partial y} + \frac{\partial v_z(x,y,z,t)}{\partial z}\right) \\
\frac{\partial v_x(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)}\frac{\partial p(x,y,z,t)}{\partial x} \\
\frac{\partial v_y(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)}\frac{\partial p(x,y,z,t)}{\partial y} \\
\frac{\partial v_z(x,y,z,t)}{\partial t} &= \frac{1}{\rho(x,y,z)}\frac{\partial p(x,y,z,t)}{\partial z}
\end{aligned} \tag{2.13}$$

where $p(x, y, z, t) = \frac{\sigma_{xx}(x,y,z,t)+\sigma_{yy}(x,y,z,t)+\sigma_{zz}(x,y,z,t)}{3}$ is the pressure field, and $K(x, y, z)$ is the Bulk modulus. To complete the equation we have to add the seismic source term $s(t)$, positioned at the coordinate $(x_s, y_s, z_s)$. The system of equations (2.13) becomes the following $2^{nd}$ order equation:

$$\frac{1}{K(x,y,z)}\frac{\partial^2 p(x,y,z,t)}{\partial t^2} - \nabla.\left(\frac{1}{\rho(x,y,z)}\nabla p(x,y,z,t)\right) = s(t)\delta(x-x_s)\delta(y-y_s)\delta(z-z_s) \tag{2.14}$$

where $\nabla.$ is the *divergence* operator, $\nabla$ the *gradient* operator and $\delta$ the Dirac delta function. We define $c = \sqrt{\frac{K}{\rho}}$ as the compressional particle velocity. The divergence and the gradient operators are correlated and are replaced by the *Laplace* operator $\Delta$. In

the case where the density $\rho$ is constant, the equation (2.14) becomes:

$$\begin{aligned} &\frac{1}{c^2(x,y,z)}\frac{\partial^2 p(x,y,z,t)}{\partial t^2} - \Delta p(x,y,z,t) = s(t)\delta(x-x_s)\delta(y-y_s)\delta(z-z_s) \\ &with : \Delta p(x,y,z,t) = \frac{\partial^2 p(x,y,z,t)}{\partial x^2} + \frac{\partial^2 p(x,y,z,t)}{\partial y^2} + \frac{\partial^2 p(x,y,z,t)}{\partial z^2}. \end{aligned} \tag{2.15}$$

Note that the displacement field $u(x,y,z,t)$, which determines the displacement of the particles during the propagation, is governed by a similar equation as the equation (2.13). Solving the pressure field $p(x,y,z,t)$ is thus equivalent to solving the displacement field $u(x,y,z,t)$:

$$\frac{1}{c^2(x,y,z)}\frac{\partial^2 u(x,y,z,t)}{\partial t^2} - \Delta u(x,y,z,t) = s(t)\delta(x-x_s)\delta(y-y_s)\delta(z-z_s). \tag{2.16}$$

This is the acoustic wave equation that we tend to solve numerically in the rest of this section. It is also the equation used to simulate the wave propagation in the seismic modeling and in the seismic imaging, i.e. in the Reverse Time Migration.

### 2.3.2 Numerical methods for wave propagation

Numerically, the solutions to the wave equation can be approximated using a wide variety of numerical methods. Depending on the targeted accuracy and on the available computational resources, one can consider a spectral formulation, a strong formulation or a weak formulation. One can also adopt a time-domain approach or a frequency-domain approach. The spectral formulation produces efficient results for simple geological structures whereas the strong formulation via finite-difference methods can give a good compromise between the quality of images and the computational costs. On the other hand, weak formulation via finite-elements, e.g. continuous or discontinuous Galerkin methods, are more suitable for areas with complex subsurfaces. For a thorough overview of the most common numerical methods used in resolving the wave equation we recommend the following two readings [50] and [215]. In this section, we briefly introduce the methods that we find most relevant to the acoustic wave equation solver.

#### 2.3.2.1 Integral methods

These methods are based on the *Huygen's principle* that states that every point in the wavefield can be considered as a secondary source. For the integral form of the scalar wave equation in homogeneous media we use the Green's function $G$

$$G(\mathbf{x}, \mathbf{x_s}, t) = \frac{\delta(t - |\mathbf{x} - \mathbf{x_s}|/c_0)}{4\pi|\mathbf{x} - \mathbf{x_s}|} \tag{2.17}$$

$$p(\mathbf{x}, t) = \int\int G(\mathbf{x}, \mathbf{x_s}, t - t')\ q(\mathbf{x_s}, t')\ d\mathbf{x_s}\ dt' \tag{2.18}$$

Green's function are used as a response to a source in the studied media. The source location is $\mathbf{x_s}$. $p$ is the pressure generated by the particles displacement in media, $c_0$ is the wave velocity and $q$ is a mass flow rate per unit volume. These approaches are more efficient in homogeneous medium.

#### 2.3.2.2 Asymptotic methods

They are also called ray-tracing methods and are used when the medium is heterogeneous. In such media, the Green's functions cannot be computed simply. An example of the asymptotic approach is the Kirchhoff approximation widely used in migration as described in [41]. Kirchhoff approximations are based on the assumption of high frequencies.

#### 2.3.2.3 Direct methods

Direct methods are based on a discretization of the computational domain. The approximation of solutions to the PDE that defines the wave equation can be done using strong formulations such as finite-difference and pseudo-spectral approaches. We can also rely on weak formulations like finite-element and finite-volume methods. We also need a time integration in order to approximate the wave equation. Depending on the formulation chosen for the equation, the space and time derivatives can be either second or first order. The source term is added to the right hand side of the PDE in order to consider the inhomogeneous solutions.

**2.3.2.3.1 Pseudo-Spectral Methods** Pseudo-spectral (PS) methods also known as the Fourier methods are strong formulations of partial differential equations. Using these approaches, pressure values $p(\mathbf{x})$ are approximated using basis functions $\psi_j$ like in equation (2.19)

$$p(\mathbf{x}) = \sum_{j=1}^{N} p(\mathbf{x_j})\psi_j(\mathbf{x}) \tag{2.19}$$

In the case of regular grids, one can use Fourier polynomials as basis functions. On the other hand Chebychev polynomials are used for irregular grids. In [135], we have a description of the Fourier methods applied to forward modeling with comparison with finite-difference and finite-element methods. Contrary to finite-difference, pseudo-spectral methods are global. Modifications when they occur affect the whole computing grid. When we opt for the pseudo-spectral methods, we reduce the number of unknowns. We also reduce the number of grid points compared to finite-difference while achieving the same accuracy.

Pseudo-spectral methods can show some limitations when the topography is complex. Finer grid discretization in order to adapt to the complexity of the surface results in higher computational cost. This impacts the efficiency of these numerical methods, restraining them to relatively simple topographies.

**2.3.2.3.2 Finite Difference Methods** Finite difference methods (FDM) are also strong formulations of partial differential equations. They are based on the discretization of the computational grid where we compute values of the wavefield. Usually, *Taylor series expansions* are used to compute the derivatives as well as to estimate the errors due to numerical dispersions. For finite-difference, we have approximations of spatial

derivatives using for example the equation (2.20) where $\Delta x$ is the spacing between two values of the field $u$.

$$\frac{du}{dx} = \lim_{\Delta x \to 0} \frac{u(x + \Delta x) - u(x)}{\Delta x} \tag{2.20}$$

The derivative can be approximated as given in equation (2.21). This is a forward difference approximation.

$$\frac{du}{dx} \approx \frac{u(x + \Delta x) - u(x)}{\Delta x} \tag{2.21}$$

One can have a backward difference approximation of the derivative like described in equation (2.22)

$$\frac{du}{dx} \approx \frac{u(x) - u(x - \Delta x)}{\Delta x} \tag{2.22}$$

Combining the forward (2.21) and backward (2.22) approximations one can have a central approximation given by the equation (2.23).

$$\frac{\partial u}{\partial x} = \frac{u(x + \Delta x) - u(x - \Delta x)}{2\ \Delta x} \tag{2.23}$$

Thorough reviews on finite difference methods applied to the seismic wave equation in general can be found in [173] and in [27]. In [158] and [128], studies of the FDM applied in isotropic and anisotropic media are presented, and aspects ranging from the grids used to the boundary conditions are discussed.

Major difficulties in FDM are due to the discretization grids. The space steps are constrained by the minimal value of the velocity in the media. For heterogeneous medium, the space discretization steps need to be very small which results in a huge computational demand. FDM can be applied in both frequency and time domains. Frequency domain may be more efficient than time domain in inversion problemswhen multiple source locations are used [167, 177, 214]. In the case of forward modeling, time domain is widely used since it is more adapted to the computation requirements of such applications [158, 215]. FDM need to satisfy important conditions in order to guarantee their effectiveness. It consists of stability, convergence and consistency.

- *Stability* means that the solution is bounded when the analytical solution of the PDE is bounded.
- *Consistency* means that the truncation tends to zero when the spatial grid spacing and the time spacing tend to zero.
- *Convergence* is satisfied when the approximated solutions, using finite-difference, approach the analytical solutions of the partial differential equation.

For finite-difference methods, we can have either an explicit or an implicit scheme. Explicit schemes use values of the wavefield at the previous time-step in order to update a given grid point for the current time step. On the other hand, implicit schemes update the whole grid at the current time-step by means of a linear system solving an inversion of a matrix [55].

**2.3.2.3.3 Finite Element Methods** Finite-element methods (FEM) are a weak formulation of partial differential equations. These methods use predefined functions as

a starting point to approximate the wavefield. Finite-element approaches are more efficient in complex topography than finite-difference and pseudo-spectral methods. They allow a neater approximation of the surface. According to the conditions ensured in approximating the wave motion, the finite-element method can be adopted in the following different ways.

**2.3.2.3.3.1 Continuous Finite Element** Continuous finite-element methods such as spectral-element (SE) suppose that the wavefield is continuous on the boundaries separating the elements where the values are computed locally from the elements where the values are explicitly given.

**2.3.2.3.3.2 Discontinuous Finite Element** Discontinuous Galerkin methods remedy to the strong affirmation we have in the continuous formulation of the standard finite-element methods. These new methods introduce the notion of flux exchange instead. Baldassari [36] presents more details about the Galerkin methods and uses them in a Reverse Time Migration workflow.

### 2.3.3 Application to the acoustic wave equation

In this section we approximate the solutions of the acoustic wave equation (2.16) by means of the Finite Difference Method. This choice is motivated by our desire to explicitly approximate the solutions of the wave equation and by the good tradeoff between numerical accuracy and computational demand that FDM offer. We consider that the medium is isotropic and that the density is constant. We consider that the propagation space is a *regular* grid governed by a cartesian coordinate system. As a matter of fact, this is the wave equation solver that will be used to evaluate the heterogeneous architectures considered in this work. However, we point out that the techniques detailed in this section apply for more complex formulations of the wave equation, i.e. for variable density media and elastic propagation.

#### 2.3.3.1 Numerical approximation

For time discretization, a $2^{nd}$ order explicit centered scheme is used, where we rely on the Taylor series expansion to approximate the $2^{nd}$ order derivatives of $u(x, y, z, t)$, with respect to time, between time $t$ and time $t + \Delta t$ as follows:

$$\frac{\partial^2 u(x,y,z,t)}{\partial t^2} \approx \frac{u(x,y,z,t+\Delta t) - 2u(x,y,z,t) + u(x,y,z,t-\Delta t)}{\Delta t^2}. \tag{2.24}$$

This scheme is also known as the *leap frog* scheme. For the sake of simplicity, we denote $U^n(x, y, z)$ the approximation of the displacement field at time $t = n\Delta t$.
For space discretization, we consider the grid spacings $\Delta x, \Delta y$, and $\Delta z$ along the $X$, $Y$ and $Z$ axis respectively. Let $U^n_{i,j,k}$ the value of the wavefield $u$ at time $t = n\Delta t$ at the grid point ($x = i\Delta x$, $y = j\Delta y$, $z = k\Delta z$) (see figure 2.15). A centered $p^{th}$ order finite difference scheme is used to approximate the $2^{nd}$ order derivatives with respect to $x$, $y$

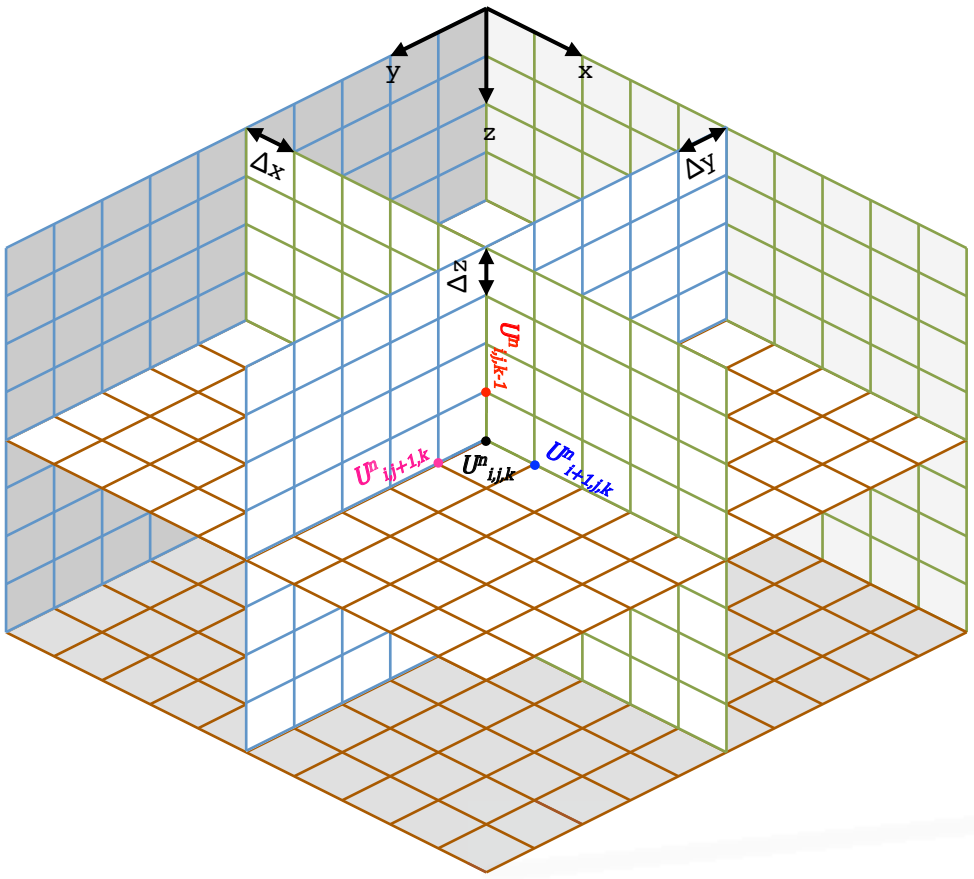



FIGURE 2.15: Space discretization of a 3D volume.

and $z$ as follows:

$$\begin{aligned}
\frac{\partial^2 u(x,y,z,t)}{\partial x^2} &\approx \frac{1}{\Delta x^2} \sum_{l=-p/2}^{p/2} a_l U^n_{i+l,j,k} \\
\frac{\partial^2 u(x,y,z,t)}{\partial y^2} &\approx \frac{1}{\Delta y^2} \sum_{l=-p/2}^{p/2} a_l U^n_{i,j+l,k} \\
\frac{\partial^2 u(x,y,z,t)}{\partial z^2} &\approx \frac{1}{\Delta z^2} \sum_{l=-p/2}^{p/2} a_l U^n_{i,j,k+l}
\end{aligned} \tag{2.25}$$

The finite difference coefficients, $(a_l)_{\frac{-p}{2} \leq l \leq \frac{p}{2}}$, with $a_{-l} = a_l \; \forall l \in [0, \frac{p}{2}]$, are obtained by Taylor series expansion. The table 2.2 summarizes the finite difference coefficients for schemes with an order ranging from 2 to 8. In our case we chose $p = 8$. The choice of the coefficients plays a primary role in terms of numerical dispersion of the finite

| *Scheme order* | $a_0$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |
|---|---|---|---|---|---|
| 2 | $-2$ | 1 | | | |
| 4 | $\frac{-5}{2}$ | $\frac{4}{3}$ | $\frac{-1}{12}$ | | |
| 6 | $\frac{-49}{18}$ | $\frac{3}{2}$ | $\frac{-3}{20}$ | $\frac{1}{90}$ | |
| 8 | $\frac{-205}{72}$ | $\frac{8}{5}$ | $\frac{-1}{5}$ | $\frac{8}{315}$ | $\frac{-1}{560}$ |

TABLE 2.2: Taylor coefficients for centered finite difference numerical schemes.

difference scheme. Other coefficients and dispersion relations are also used to optimize the numerical scheme [140].

Based on the discretization schemes (2.24) and (2.25), the acoustic wave equation (2.16) is transformed into the following difference equation:

$$\frac{1}{c^2_{i,j,k}} \frac{U^{n+1}_{i,j,k} - 2U^n_{i,j,k} + U^{n-1}_{i,j,k}}{\Delta t^2} - \Delta U^n_{i,j,k} = s^n \delta(x - x_s)\delta(y - y_s)\delta(z - z_s), \tag{2.26}$$

where $c^2_{i,j,k}$ is the indicial notation of the velocity squared $c^2(x, y, z)$ at each grid point, $\Delta U^n_{i,j,k} = \frac{1}{\Delta x^2}\sum_{l=-p/2}^{p/2} a_l U^n_{i+l,j,k} + \frac{1}{\Delta y^2}\sum_{l=-p/2}^{p/2} a_l U^n_{i,j+l,k} + \frac{1}{\Delta z^2}\sum_{l=-p/2}^{p/2} a_l U^n_{i,j,k+l}$ is the indicial notation of the *Laplace* operator, and $s^n$ is the simplified notation of the source term $s(t)$ at time $t = n\Delta t$. We conclude that the numerical solution of the acoustic wave equation at time $(n+1)\Delta t$ is determined as a function of the solution at time $n\Delta t$ and $(n-1)\Delta t$ as follows:

$$U^{n+1}_{i,j,k} = 2U^n_{i,j,k} - U^{n-1}_{i,j,k} + c^2_{i,j,k}\Delta t^2 \Delta U^n_{i,j,k} + c^2_{i,j,k}\Delta t^2 (s^n \delta(x - x_s)\delta(y - y_s)\delta(z - z_s)). \tag{2.27}$$

We changed the hyperbolic partial differential equation (PDE) (2.16) into a discrete problem (2.27), which can be solved in a finite sequence of arithmetic operations, implementable on a computer. In the rest of the document we make use of the equation (2.27) with $p = 8$, i.e. the $8^{th}$ order centered finite difference scheme, as the numerical solver of the acoustic wave equation.

#### 2.3.3.2 Stability analysis and CFL

We presented a numerical method, i.e. a centered finite difference scheme, to discretize the acoustic wave equation. Finite difference discretization, as well as all the numerical methods used to solve PDEs, are subject to numerical *errors*. An error is a difference between the analytical solution of the PDE and the solution of the discrete problem. In numerical analysis, a numerical scheme is proved convergent if and only if it is proved *consistent* and *stable* (the Lax Equivalence Theorem [122]). To learn more about the consistency, convergence and stability of the numerical methods we refer the reader to [77]. In practice, consistency essentially requires that the discrete equations defining the approximate solution are at least approximately satisfied by the true solution. For the finite difference method, this is an evident requirement (Taylor's theorem). Thus in order to prove the convergence of our finite difference scheme we should define a quantitative measure of its stability. We consider the CFL condition named after Richard Courant, Kurt Friedrichs and Hans Lewy who described it in their paper [70]. It consists of the following inequality:

$$\left(c\frac{\Delta t}{h}\right) \leq \alpha, \tag{2.28}$$

where $\Delta t$ and $h$ are the discretization steps in time and space respectively, $c$ is the maximum velocity and $\alpha$ is a constant that depends on the numerical scheme considered. By von Neumann stability analysis, we extend the CFL condition to the 3D finite difference wave equation solver as follows:

$$\left(c\frac{\Delta t}{\Delta x}\right)^2 + \left(c\frac{\Delta t}{\Delta y}\right)^2 + \left(c\frac{\Delta t}{\Delta z}\right)^2 \leq \alpha, \tag{2.29}$$

where $\alpha = 4\left(\sum_{l=-p/2}^{p/2} |a_l|\right)$. Or in a more generalized formulation, presented in [141], where the discretization step is assumed to be the same in all the directions, i.e. $\Delta x = \Delta y = \Delta z = h$, and the number of dimension is $N_{dim}$:

$$\left(c\frac{\Delta t}{h}\right) \leq 2\left(\frac{\sum_{l=-p/2}^{p/2} |a_l|}{N_{dim}}\right)^{1/2}. \tag{2.30}$$

#### 2.3.3.3 Boundary conditions

By solving the acoustic wave equation (2.16) we would like to simulate the wave propagation in a half space, i.e the earth (land or sea) surface plus the underlying subsurface structures. However, the numerical equation (2.27) is defined in a finite discrete domain. In order to simulate the half space constraint in the numerical equation, we need to capture the reflected waves on the artificial domain sides and bottom of the numerical grid. This can be done by adding *absorbing layers* to the computational domain faces to progressively damp the energy coming from the incident waves. Berenger [39] introduced in 1994 the *Perfectly Matched Layers (PML)* as an absorbing boundary condition to simulate the propagation of the 2D electromagnetic waves. PML were then extended to 3D wave propagations in general. Finite difference method is often used in conjunction with PML in different variants. Convolution PML (CPML) can be used to improve the behavior of the discrete PML which is completely independent of the host medium. Thus, no modifications are necessary when applying it to inhomogeneous, lossy, anisotropic, dispersive or non-linear media [183]. More recently a formulation of the unspotted CPML that can easily be extended to higher-order time schemes, called the auxiliary differential equation PML (ADE-PML), has been introduced in [134] for the seismic wave equation. An improved sponge layer, called the split Multi-axial PML (M-PML), has been suggested in [154].

Mathematically, the acoustic wave equation (2.16) is altered in order to add an absorbing term (regular PML) [136]:

$$\begin{aligned}\frac{1}{c^2(x,y,z)}\frac{\partial^2 u(x,y,z,t)}{\partial t^2} - \Delta u(x,y,z,t) + 2\gamma(x,y,z)\frac{\partial u(x,y,z,t)}{\partial t} + \\ \gamma(x,y,z)^2 u(x,y,z,t) = s(t)\delta(x-x_s)\delta(y-y_s)\delta(z-z_s),\end{aligned} \tag{2.31}$$

where $\gamma(x,y,z)$ represents the damping coefficients. Numerically, when neglecting the $\gamma(x,y,z)^2$ term, the propagation in the absorbing layers is thus governed by the following equation:

$$\begin{aligned}U_{i,j,k}^{n+1} = \frac{2}{\Gamma_{i,j,k}+1}U_{i,j,k}^{n} - \frac{\Gamma_{i,j,k}-1}{\Gamma_{i,j,k}+1}U_{i,j,k}^{n-1} + \frac{c_{i,j,k}^2\Delta t^2}{\Gamma_{i,j,k}+1}\Delta U_{i,j,k}^{n} \\ + c_{i,j,k}^2\Delta t^2(s^n\delta(x-x_s)\delta(y-y_s)\delta(z-z_s)),\end{aligned} \tag{2.32}$$

where $\Gamma_{i,j,k}$ is the indicial notation of $\gamma(x,y,z)$. Finally, we recall that the wave propagation in the rest of the computational domain is governed by the equation (2.27).

# Chapter 3

# High performance computing

## Contents



The *Top500* list [9], a LINPACK-based supercomputer ranking list that is updated every six months, of November 2014 indicates that the computing landscape looks more and more diversified. With the advent of powerful hardware accelerators, the HPC facilities are turning heterogeneous, where commodity processors are usually used in conjunction with auxiliary chips such as the Graphics Processing Unit (GPU) or the

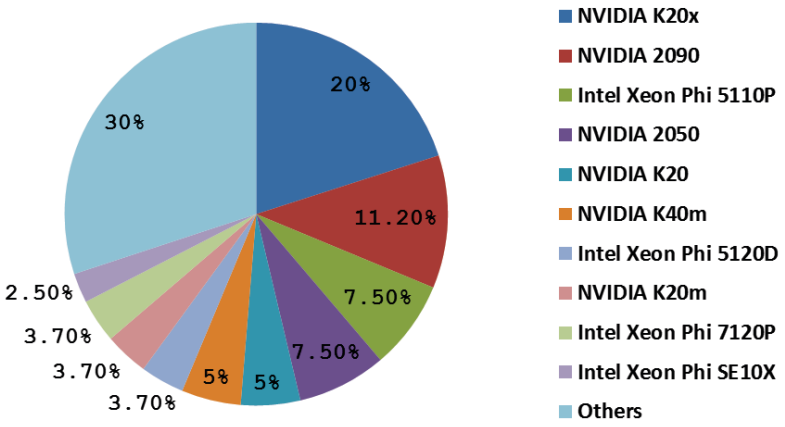



FIGURE 3.1: A classification of the hardware accelerators that feature the Top500 supercomputers, based on the ranking list of November 2014.

Field-Programmable Gate Array (FPGA) to enhance the performance of scientific applications. A quick glance at the chart presented in figure 3.1, which is a summary of the most used accelerators in the latest Top500 supercomputers list, gives a better idea about the recent trends in HPC facilities. Besides, leveraging the compute power of such hybrid architectures can be challenging since various programming models are to be considered. For example, the emergence of GPUs led scientific programmers to introduce new programming models such as CUDA [4] and OpenCL [19]. Furthermore, today we are about to reach a new milestone in the history of computing as the power consumption seems to be a serious concern. *Green500* [6], another list that provides since 2007 a ranking of the most energy-efficient supercomputers in the world, is gaining more and more interest inside the HPC community.

In this chapter, we give in section 3.1 a succinct overview of the most used processors and hybrid architectures in the scientific community. We follow up with a summary of the programming models used to leverage these architectures, in section 3.2. Then, we finish the chapter in section 3.3 where we explain how does the power consumption interfere with HPC.

## 3.1 Overview of HPC hardware architectures

In this section, we describe the recent advances of processors architecture with a special emphasis on the GPU in a general purpose computation context (GPGPU), and on more recent technologies such as the *Accelerated Processing Unit* (APU) proposed by AMD. First, we quickly survey the latest developments in CPU architectures. Then, we give an overview of the GPGPU ecosystem mainly dominated by the two vendors AMD and NVIDIA. Finally, we introduce the APU technology in a nutshell and give details about its underlying architecture.

### 3.1.1 Central Processing Unit: more and more cores

For decades, CPU manufacturers tended in their constant search for performance and compute power to increase the CPU clock rates and we have seen through the history of CPUs frequencies growing up to reach 4.4 $GHz$ (with the turbo mode) at most. Moore's law, named after Gordon Moore who predicted in 1975 that the number of transistors will be doubled every two years, has driven the CPU evolution for 40 years, where the CPU sockets were getting more and more dense, in terms of transistors, thanks to the transistor shrinkage.
However, during the last decade CPU sockets capacitance has started to stall due to a physical constraint which is the power consumption. Indeed the power budget is becoming a leading design constraint when populating a piece of silicon with functional circuits. Vendors started to realize that they could not keep raising the frequency anymore without risking that circuits would be subject of overheating. Alternatively, they dawned the *multi-core* age by reducing the frequency and by duplicating CPU *cores*, i.e. independent processing units each having their own ALUs, FPUs and caches, on chip. In figure 3.2 one can see that the direct result of power constraint is a stall in the CPU frequency. The figure also shows the transition from mono-core CPUs to multi-core CPUs that started during the last decade.
Furthermore, another factor that has also strongly influenced the design of modern CPUs

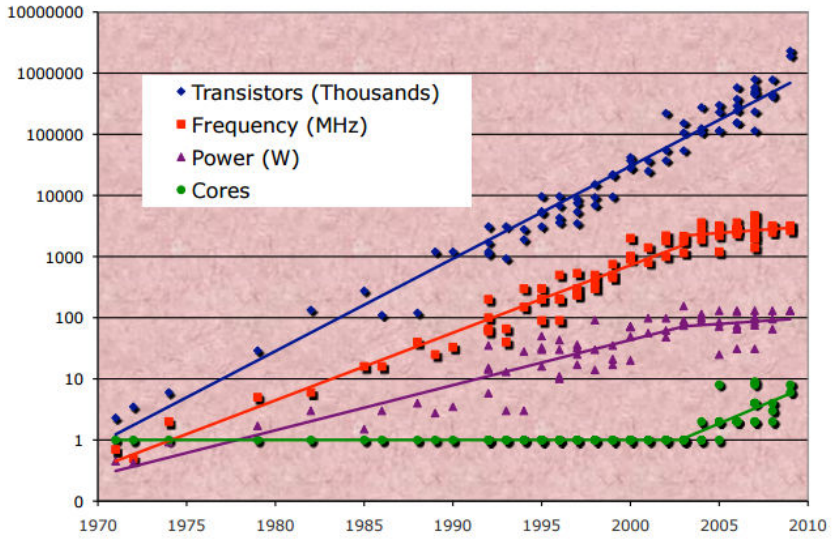



FIGURE 3.2: The evolution of CPUs since 1970 in terms of power consumption, transistor density, frequency and number of cores. Source [106].

architecture is the mismatch between the CPU frequency and memory speeds (see figure 3.3). This clock rate gap forced the vendors to dedicate the major part of the silicon of a CPU to hierarchical levels of caches. For instance, the size of the second-level cache grew rapidly, reaching 2 MB in some instances [26]. In addition, multiplying the cores within the socket is translated to more demand for data, and some vendors (such as Intel) thus introduced a third-level cache. The figure 3.4 depicts the latency of the different levels of the state-of-the-art CPU memory hierarchy from registers to hard disks. It also gives an idea about the average capacity of each memory level. The memory latency is still much greater than the processor clock step (around 300 times greater or more). The memory throughput is improving at a better rate than its latency, but it's still lagging behind the processor speed. Today, high performance CPUs (CPUs that are used in HPC facilities) are often composed of one or multiple sockets, each of which has multiple cores that range from two to twenty. They are featured by two or three levels of caches. We sketch an abstract view of the architecture of modern CPUs in figure 3.5. Most of the state-of-the-art CPU architectures fall in this example. Note that the L3 cache is optional since not all the CPUs are equipped with a third level cache.

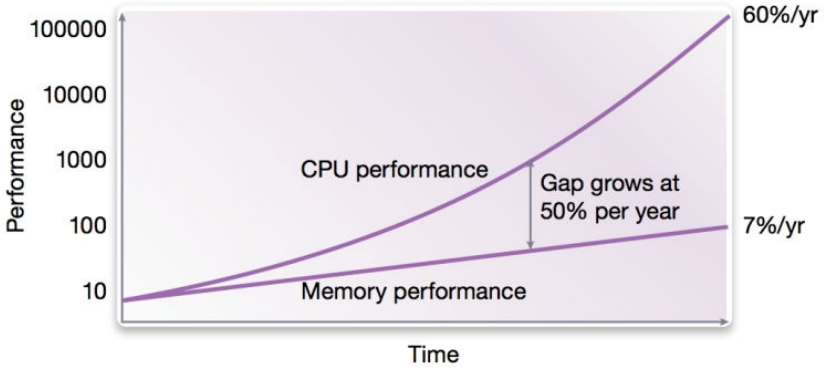



FIGURE 3.3: The performance gap between the CPU clock rate and the DRAM clock rate throughout the years, from [5].

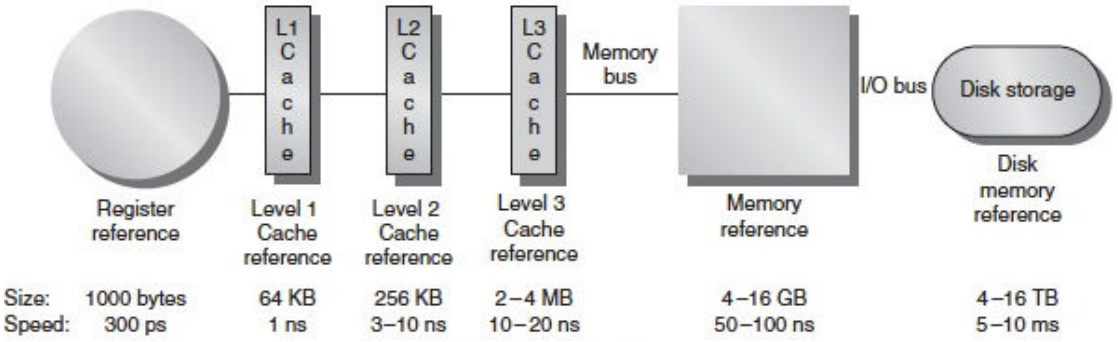



FIGURE 3.4: The average sizes and latencies of state-of-the-art memory hierarchy elements. The time units change by a factor of $10^8$ from the latency of disks to that of registers, and the size units change by a factor of $10^{10}$ from the capacity of registers to that of disks. From [110].

Intel launched successive families of high-end multi-core CPUs. Sandy Bridge was released in 2011, with up to 8 cores per CPU socket, 20MB of L3 cache and 150 Watts of maximum TDP (Thermal Design Power). The Ivy Bridge family was the successor in 2013 and demonstrated between 3% and 6% of performance enhancement compared to Sandy Bridge (clock to clock comparison). The number of cores per die was increased (up to 18 cores) and the L3 cache sizes were increased up to 38MB while the power envelop was almost kept the same as the previous generations. Most recently the cutting edge Intel CPU family, code named Haswell, was released. It features up to 18 cores per CPU socket to deliver 8% more performance than that of Ivy Bridge.
Albeit not dominant in the HPC CPU market, AMD also lined up high-end CPUs for HPC. The Bulldozer micro-architecture was released in 2011, it featured up to 4 cores with a maximum TDP of 140 Watts. Bulldozer CPUs had up to 8MB of L3 cache. The Piledriver family was released in 2012. Piledriver processors had up to 16 cores per die. The Streamroller micro-architecture was released in 2014 to build the Warsaw Server CPU product line. They were also featured by 12 to 16 CPU cores per socket, and

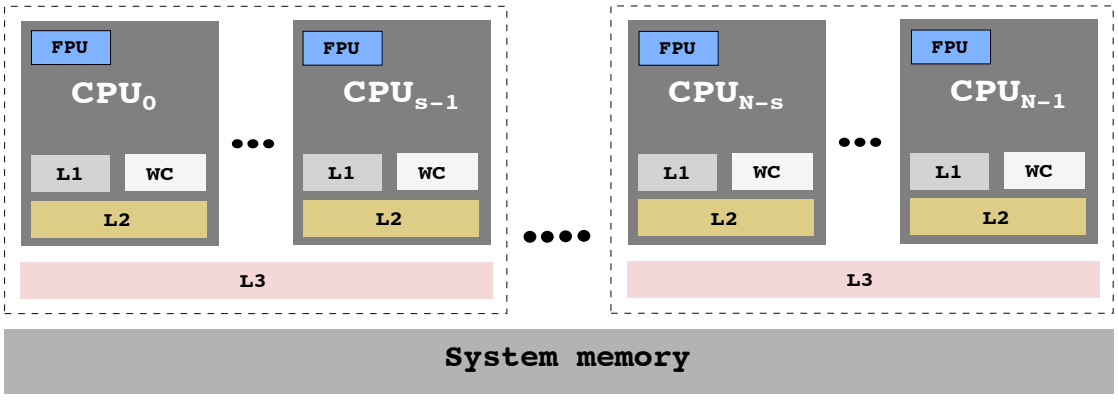



FIGURE 3.5: An abstract view of the architecture of a multi-core CPU. $N$ is the overall number of CPU cores. L1, L2 and L3 refer respectively to the first cache level, the second cache level and the third cache level. $s$ is the maximum number of cores that share one L3 cache (usually in one socket). $WC$ are Write Combining buffers and are often used for cache non-coherent memory accesses.

| | |
|---|---|
| *Architecture* | AMD 10h |
| *Model* | Phenom |
| *Clock rate (GHz)* | 2.8 |
| *CPU #cores* | 6 |
| *Main memory (MB)* | 8096 |
| *Peak bandwidth (GB/s)* | 50 |
| *Single precision peak flops (GFlop/s)* | 134 |

TABLE 3.1: The technical specifications of the AMD CPU that is surveyed in the scope of this work.

were reported to consume only 100 Watts of TDP. AMD plan to release the Zen micro-architecture in 2016 which promises 40% of performance improvement over the previous generations. Table 3.1 summarizes the technical specifications of the *AMD Phenom TM II x6 1055t Processor* that is used in this study.

These architectures are slightly different but they all are subject to parallel processing which can be exposed at multiple level. First, the Instruction Level Parallelism (ILP) enables the processor to execute more than one instruction from the same thread in parallel using one or multiple pipelines at the core level. Second, comes the Data Level Parallelism (DLP) where the same instruction is executed on different data entries (vector processing) using SIMD (Single Instruction Multiple Data) registers. The third level of parallelism is the Thread Level Parallelism (TLP) which corresponds to the capability of executing multiple threads on the same core, i.e. SMT (Simultaneous Multi-Threading), to the parallel execution of multiple threads on different CPU cores (in shared memory systems) or even to the parallel execution of multiple processes on different compute nodes of a distributed machine (CPU clusters).

### 3.1.2 Hardware accelerators: the other chips for computing

It's no secret that applicative workloads are becoming larger and more complex than ever. In some cases, the traditional CPUs hardware cannot meet high computational demands. As a matter of fact, high-performance computing applications are now demanding more than traditional CPUs can deliver, creating a technology gap between demand and performance. This limits users from extracting the performance out of this hardware. Application demands have outpaced the conventional processor's ability to deliver performance. An alternative solution is hardware acceleration that augments mainstream processors with specialized coprocessors.
During the last decade, many vendors have proposed hardware accelerators based solutions for general purpose computing. The Graphic Processing Units (GPUs) are the most famous ones. NVIDIA and AMD are largely dominating the market by introducing successive GPU architectures for more than two decades now. In addition to GPUs, high performance facilities are sometimes populated with other accelerators, such as FPGAs and Intel Xeon Phi. Intel Xeon Phi processors are based on the MIC (Many Integrated Core) architecture, which combines near 60 x86 cores on the same die. The MIC products are not stand-alone processors yet, and they are used as an external PCI Express card for computing. However, it is reported that the first stand-alone ones will be launched in 2016. Moreover, other technologies that rely on founding a high number of cores on the same chip start to emerge. The MPPA, a SOC (System On Chip) that

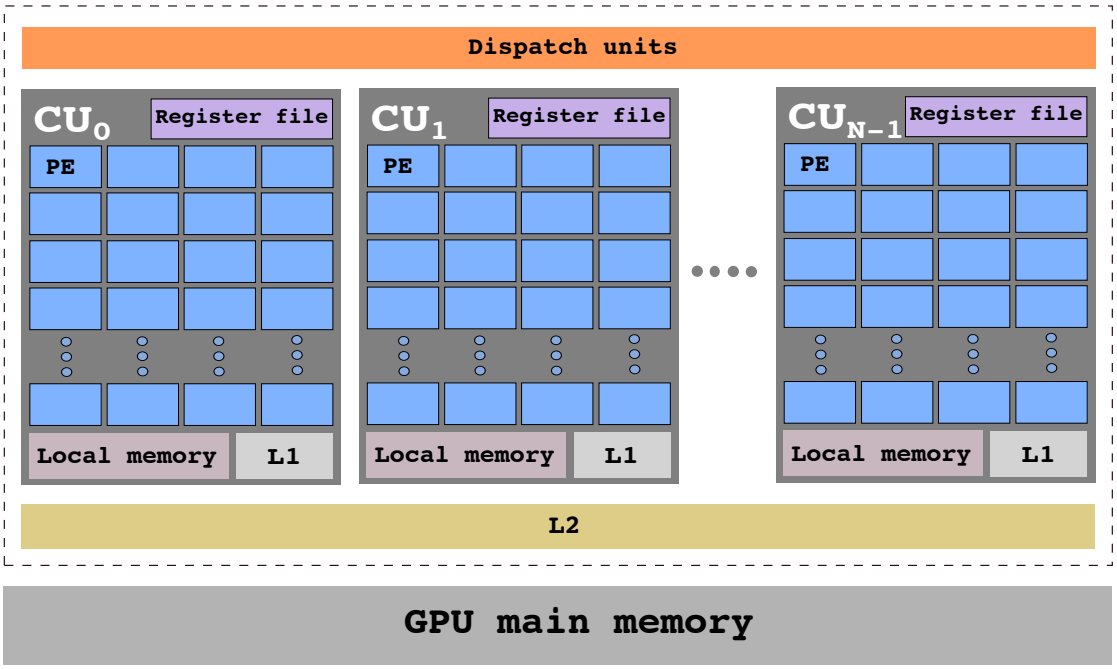



FIGURE 3.6: An abstract view of the architecture of a modern GPU.

contains 256 cores and proposed by Kalray [8], is one of them.
Five out of the fastest ten supercomputers (based on the Top500 list of November 2014) in the world are built upon hardware accelerators (mainly GPUs or Intel Xeon Phi).

In this section, we particularly focus on the architectures of GPUs. A GPU is a processor optimized for graphics workloads. The GPU has recently evolved towards a more simplifed architecture, typically it is composed of several massively parallel *multi-processors*, also referred to as streaming multi-processors or compute units in the literature (we denote them as CU), each of which has a large number of processing elements (we refer to them as PEs in this document) that all run the same instructions on different data (SIMD). These compute units contain, in addition to the arithmetic and logic units, branch units, memory fetch units and additional hardware that is not relevant to general purpose computations. The GPU needs thread schedulers and some control units to dispatch the work amongst the PEs. We give a simplified view of a common architecture of modern GPUs in figure 3.6. Note that we describe only the relevant hardware components and that the architecture may slightly differ from a vendor to another. The GPU main memory, referred to as GPU global memory, is accessible by all the threads on the GPU and has a high latency (at the order of 600 ns). Often the local memory, i.e. a scratchpad memory that is local to each compute unit and thus is accessible only by the threads that are allocated in the corresponding CU, is used as a manually managed cache by first moving data from the global memory to the local memory and using that memory in any further calculations. This mitigates the impact of the GPU main memory latency. For the same purpose, recently GPUs have been equipped with a hierarchy of caches (L1 and L2). The registers (Register file) are private to each thread and are used to store intermediate data and variables during computation.

The execution model of GPUs requires the deployment of a tremendous number of threads (tens of millions) and keeping them active simultaneously on the hardware without changing contexts. The schedulers are responsible for filling the stalls caused

by memory accesses for instance by deploying more threads that are ready to use the GPU cycles. This execution model gives rise to a natural parallelism, that can be categorized as TLP, as the *wraps* or *wavefronts* (a collection of threads that are executed simultaneously on a CU) execution are interleaved to hide latencies. On a lower level, ILP can be exploited which implies using more registers per thread. Unlike CPUs, where a large number of the transistors is dedicated to supporting non-computational tasks like branch prediction and caching, GPUs use additional transistors for computation achieving greater compute power with the same transistor count.
GPUs do not operate on the computer main memory. Often a GPU is connected to its own off-chip memory (the GPU global memory) which is used to store data. The size of this graphics memory varies but it is currently about 3 to 12 Gigabytes for high-end GPUs. Before the GPU can start to work on a given data, that data first needs to be moved to the GPU main memory. The speed of that operation depends on the connection between the main memory and the graphics board via the PCI Express bus. Therefore it varies heavily to a maximum of 6 $GB/s$ (PCI Express gen 2, 16x) or 12 $GB/s$ (PCI Express gen 3). When used in HPC, GPUs can have over an order of magnitude higher memory bandwidth and higher computation power (in terms of GFflop/s) than CPUs. For example a high-end Intel Ivy Bridge EX processor with 15 cores hits a theoretical single precision performance of 672 $GFlop/s$ and 25 $GB/s$ of memory bandwidth, while an NVIDIA K40m GPU offers 4029 $GFlop/s$ of theoretical single precision performance and near 320 $GB/s$ of theoretical bandwidth.

During the last five years, NVIDIA has launched successively three GPU architectures on top of which the company has released a brand (Tesla) that features double precision, ECC memory etc. and that targets general purpose computing. Those GPUs were extensively used by high performance facilities. The Fermi micro-architecture was introduced in 2010 where each streaming multi-processor (SM) was composed of 32 PEs, also introduced by the vendor as *CUDA cores*. Each SM has 64 $KB$ of scratch-pad memory that can be partitioned by the users into level 1 cache and local memory (depending on the GPU model). The single precision peak performance of Fermi GPUs is 1.5 $TFlop/s$. Depending on the model, the GPU memory capacity was about 6 $GB$ and could deliver up to 192 $GB/s$ of peak bandwidth. The following architecture was Kepler, where NVIDIA had introduced a new generation of streaming multi-processors, called SMX, each holding 192 CUDA cores. NVIDIA reported a 3× speedup in terms of performance per watt. With Kepler architecture, NVIDIA has also introduced the *dynamic parallelism*, which allows GPU threads to spawn new threads on their own and launch kernels without the help of the CPU. In addition to the *Hyper-Q* technology, which allows multiple CPU processes (can be MPI processes) to simultaneously utilize a single GPU. Kepler GPUs peak at 4.5 $Tflop/s$ in terms of single precision performance and 320 $GB/s$ in terms of bandwidth (depending on the model). More recently, the Maxwell GPU architecture has been introduced. This GPU generation focuses more on power efficiency. The second level cache size is increased up to 2 $MB$.

AMD GPUs are evolving along generations and the way they are structured varies with the device family. Each GPU family is based on a GPU micro-architecture. In the Evergreen family, processing elements are arranged as five-way very long instruction word (VLIW) processors. Consequently, five scalar operations can be co-issued in a VLIW instruction. Then, the Northern Islands GPU family has come up with a new design, in which the processing elements of one multi-processor are arranged as four-way VLIW processors. Northern Islands GPUs have two wavefronts schedulers. We generally refer to the hardware design of Evergreen and Northern Islands GPUs as *vector design*.

| *Architecture* | Cayman | Tahiti |
|---|---|---|
| *Model* | HD6970 | HD7970 |
| *GPU family name* | Northern Islands | Southern Islands |
| *Clock rate (GHz)* | 0.88 | 0.925 |
| *Compute units* | 24 | 32 |
| *Off-chip memory (MB)* | 2048 | 3072 |
| *Local memory (KB)* | 32 | 64 |
| *Peak bandwidth (GB/s)* | 176 | 256 |
| *Single precision peak flops (GFlop/s)* | 2700 | 3700 |

TABLE 3.2: The list of the AMD GPUs that are surveyed in the scope of this work.

As a matter of fact, Evergreen and Northern Islands are based on a vectorized micro-architecture called *TeraScale graphics* that defines a relaxed memory model without caches.
Recently, AMD has released a new micro-architecture name *Graphics Core Next (GCN)*. The first GCN based GPU family is introduced in 2012 and is called Southern Islands. The processing elements are not VLIW processors anymore, and are arranged as four separate lanes of scalar processing elements. The wavefronts schedulers are doubled (four schedulers). The Southern Islands architecture is sometimes referred to as the *scalar design* (in reality it is a dual scalar/vector hardware design) in the literature. For most AMD GPUs, the processing elements are arranged in four SIMD arrays consisting of 16 processing elements each. Each of the SIMD arrays executes a single instruction across a block of 16 work-items. That instruction is repeated over four cycles to process the 64-element wavefront. In the Southern Islands family, the four SIMD arrays can execute code from different wavefronts. AMD continued to release GCN based GPUs with the Sea Islands and Volcanic Islands families in 2013 and 2014. Table 3.2 summarizes the technical specifications of the AMD GPUs surveyed in the scope of this work. Note that the main difference between the AMD GPUs and NVIDIA GPUs consists in the size of the SIMD arrays and in the number of the thread schedulers.

### 3.1.3 Towards the fusion of CPUs and accelerators: the emergence of the Accelerated Processing Unit

Hardware accelerators are becoming a customary component on mainstream HPC facilities. Thanks to their huge compute power and to their high internal memory bandwidth, GPUs are considered as a compelling platform for computationally demanding tasks. However, a GPU is not a stand-alone processor and require a commodity CPU, to which it is connected via the PCI Express bus, to operate. In 2011 AMD has released a new technology that promises to improve the GPU architecture: the Accelerated Processing Unit (APU). An APU is a new kind of processor that combines the advantages of a CPU and a GPU. The AMD APU has a multi-core CPU and a GPU, all fused in the same silicon die, which eliminates the need to use the PCI Express bus as the CPU and the integrated GPU can both have access to the system main memory. In order to distinguish between the traditional GPUs, referred to as *discrete GPUs* in the rest of the document, from those merged within APUs, we call the latter *integrated GPUs*.
The figure 3.7 shows a minimal description of the architecture of the AMD APU. In this figure, we represent an APU with a quad-core CPU, but in general AMD APUs are released in many variants (two CPU cores or four CPU cores). The integrated GPU

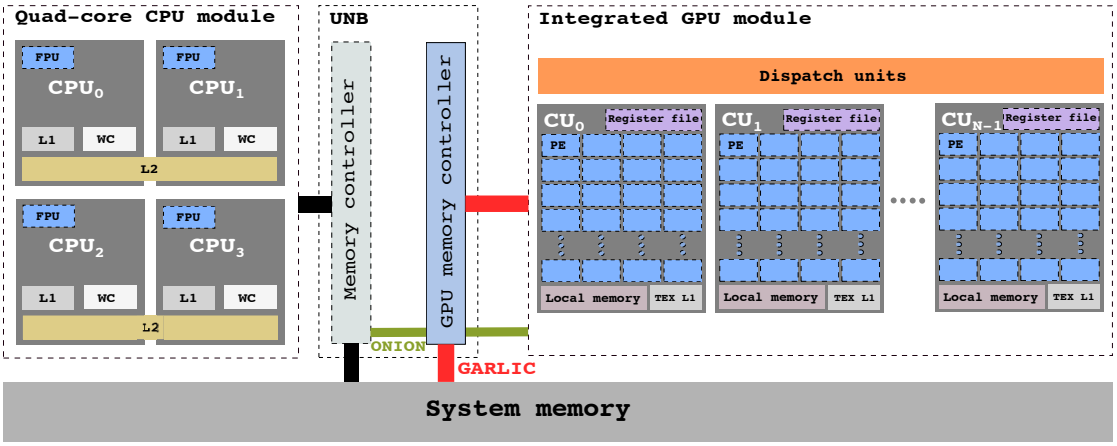



FIGURE 3.7: A high level illustration of the architecture of the early generations of AMD APUs. In this example the APU has four CPU cores. The integrated GPU has access to the main system memory, through the UNB (Unified North Bridge), using either the *Onion* memory bus or the *Garlic* bus. The integrated GPU does not have caches, the TEX L1 are the texture caches.

does not have an off-chip memory, instead it has access to the system memory by means of two new memory buses introduced by AMD: the *Onion* bus and the *Garlic* bus. Memory accesses through Onion are coherent with the CPU cores caches, whereas those performed through the Garlic bus are not. The theoretical peak bandwidth of the memory buses depends on the clock rate of the DRAM installed in the system. For example, with a system memory clocked at 1833 $MHz$, the theoretical memory bandwidth of Onion is 8 $GB/s$. Garlic, being a wider bus, peaks at $25,6$ $GB/s$. The main APU feature that AMD advanced, at the beginning of the project roadmap, is to allow CPU cores and GPU CUs to share a unified memory space. However, merging two different memory addressing systems is a challenging task at both hardware and software stacks. At an early stage of the project, the system main memory is partitioned. We represent, in the figure 3.8, the different memory partitions of an APU. The CPU cores, in a similar manner to regular multi-core CPUs, have access to the system memory, commonly named *host memory*. The integrated GPU has a dedicated subset of the main memory that is referred to as "GPU memory" and often called *device memory*. This memory is accessible by the integrated GPU multi-processors at full bandwidth of the Garlic bus. Furthermore, in order to allow the integrated GPU to access the host memory without explicitly copying data in the GPU memory, the device-visible host memory partition can be used. Similarly, the CPU can map the host-visible device memory (also known as GPU *persistent memory*) into its virtual memory space in order to share data with the integrated GPU without explicit copies. The memory buffers that are created in either the host-visible device memory or the device-visible host memory are called "zero-copy" buffers. Note that in the early generations of APUs, the host-visible memory and


FIGURE 3.8: An illustration of the different memory partitions of an APU. The system memory and the host-visible device memory are visible to the CPU cores. The GPU memory (being a sub-partition of the system memory) and the device-visible host memory are visible to the integrated GPU compute units.

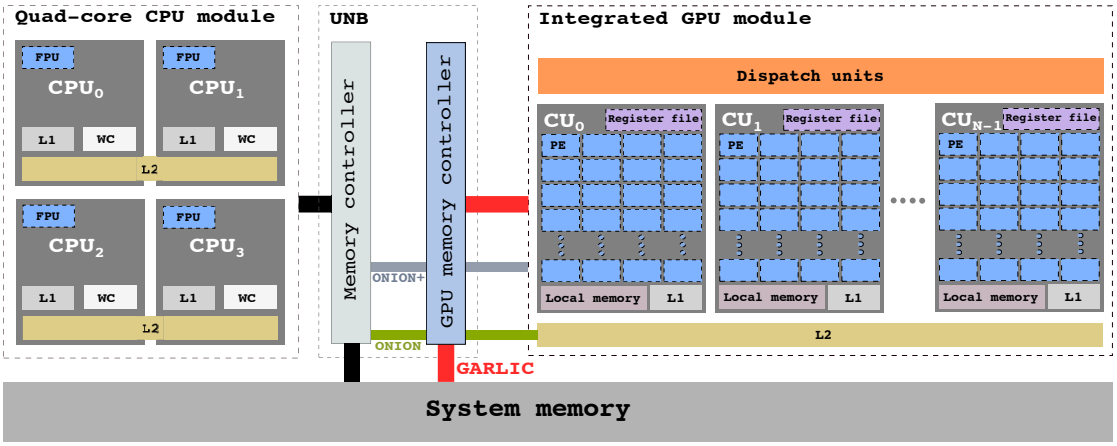



FIGURE 3.9: A high level illustration of the architecture of GCN based APUs. The GPU can access the "host coherent memory" using the *Onion+* memory bus.

device-visible memory have a limited size (between 512 $MB$ and 1 $GB$).
APUs are evolving with respect to the AMD CPU roadmap along with that of the discrete GPUs with one generation above. Put another way, the latest integrated GPU is usually one generation older than the latest discrete GPU. The first generation was code-named Llano and released in late 2011. Llano was a first shot with an integrated GPU based on the Evergreen architecture and had five GPU CUs. The second generation was Trinity (released in late 2012) with an integrated GPU that belongs to the Northern Islands family and had six GPU CUs. The Trinity and Llano APUs are both based on the vectorized micro-architecture family (TeraScale graphics) and both have the same memory subsystem (described in figure 3.7).
In 2014, AMD announced an important upgrade in the APU roadmap and launched an APU code-named Kaveri, a combination of four Streamroller CPU cores and eight Sea Islands GPU CUs. With the Kaveri APU, AMD made the move from TeraScale graphics to the newer GCN (specifically the GCN 1.1). Kaveri shares the same GPU micro-architecture as that of the latest discrete GPU lineup. After Kaveri, AMD is synchronizing the architecture of their APUs and discrete GPUs, thus optimizations made for their discrete GPUs will immediately feed back into their APUs. The difference between the Kaveri implementation of GCN and the discrete GPUs one, aside from the association with the CPU in silicon, is the addition of the coherent shared unified memory. As a matter of fact, thanks to GCN, caches are added to the integrated GPUs and the memory subsystem has changed. We illustrate in figure 3.9 the new APU architecture. Memory accesses through the Onion bus require sweeping over the GPU L2 caches as well as the CPU caches. Given that GPU L2 caches and CPU caches are not synchronized, the Onion memory bus does not ensure coherency anymore. A third memory path *Onion+* is added and has the same bandwidth as Onion. Onion+ memory accesses bypass the GPU L2 caches which allows coherency since only the CPU caches are checked before memory accesses. The memory accessed through Onion+ is referred to as "host coherent memory" in the literature [29]. Furthermore, the sustained bandwidth of the Onion bus is enhanced and peaks at 60% of the maximum sustained bandwidth of the *Garlic* bus. Besides, within the Kaveri APU the entire system memory is pageable and is addressable by both the CPU cores and the GPU compute units.
Today, the APU project still has an active roadmap. AMD unveiled in 2015 another generation of APUs code-named "Carrizo" [1]. The memory subsystem of the Carrizo APU is updated. The memory buses Onion, Onion+ and Garlic are replaced by a unique

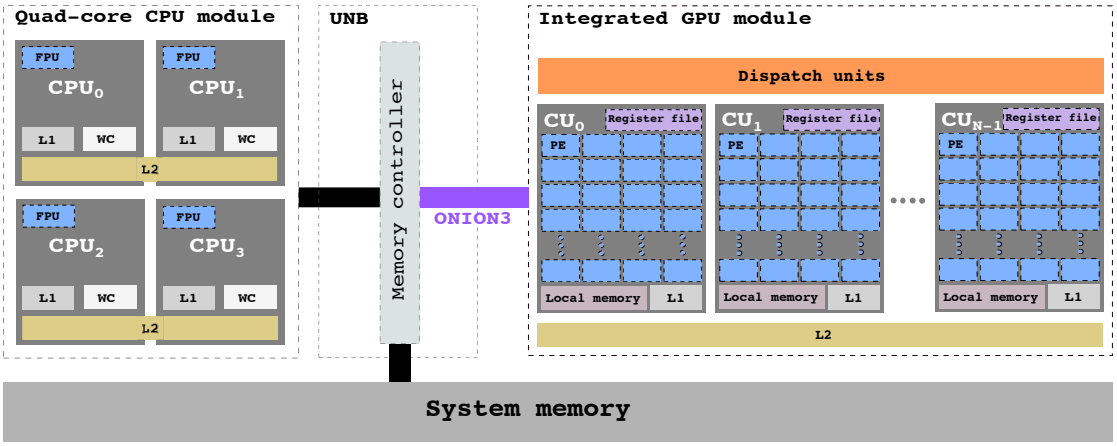



FIGURE 3.10: The architecture of the upcoming APUs. All memory accesses are performed through a unified memory bus *Onion 3*.

memory bus labeled *Onion 3*. All the memory accesses are managed by the same memory controller and are coherent with both CPU and GPU caches. The figure 3.10 is a representation of this the new APU design. Furthermore, AMD plans to launch a high performance APU with a similar memory subsystem as that of Carrizo but populated with a big number of GPU multi-processors and equiped with stacked memory (also known as High Bandwidth Memory (HBM)). The new chip is labeled "Zen APU". We summarize the different APU generations released from 2013 and the upcoming generations scheduled for 2016 in the figure 3.11. In the scope of this work, we only investigate the first three APU generations, i.e. Llano, Trinity and Kaveri. The table 3.3 summarizes the technical specifications of each one of them.

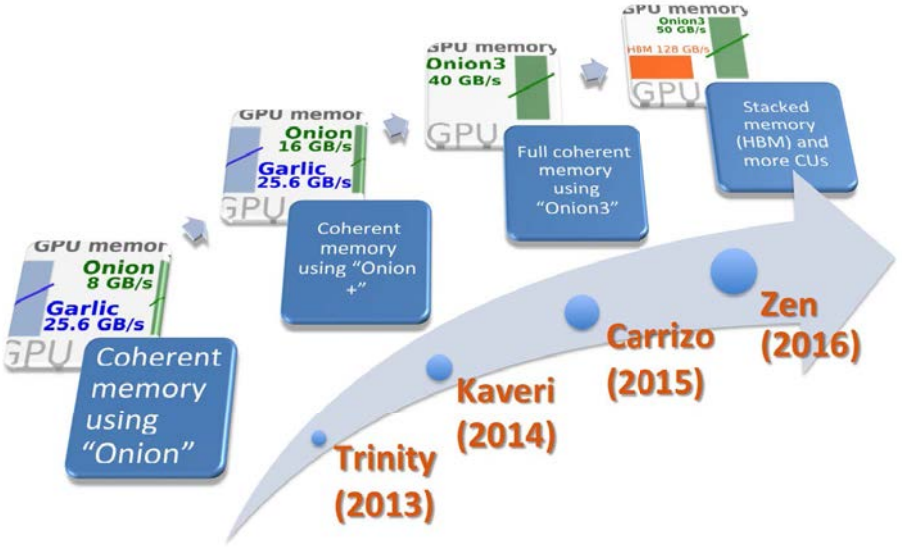



FIGURE 3.11: An illustration to summarize the APU roadmap from 2013 to 2016 and the different features.

| *Architecture* | Llano | Trinity | Kaveri |
|---|---|---|---|
| *Model* | A8-3850 | A10-5700 | A10-7850K |
| *CPU architecture* | AMD 10h | Piledriver | Streamroller |
| *CPU #cores* | 4 | 4 | 4 |
| *CPU clock rate (GHz)* | 3.0 | 3.8 | 3.8 |
| *Integrated GPU architecture* | Evergreen | Northern Islands | Sea Islands |
| *GPU clock rate (GHz)* | 0.600 | 0.711 | 0.720 |
| *Compute units* | 5 | 6 | 8 |
| *Integrated GPU memory (MB)* | 512 | 512 | 2048 |
| *Local memory per CU (KB)* | 32 | 32 | 64 |
| *Peak bandwidth (GB/s)* | 25.6 | 25.6 | 25.6 |
| *GPU single precision peak flops (GFlop/s)* | 480 | 546 | 734 |

TABLE 3.3: The list of the AMD APUs that are surveyed in the scope of this work.

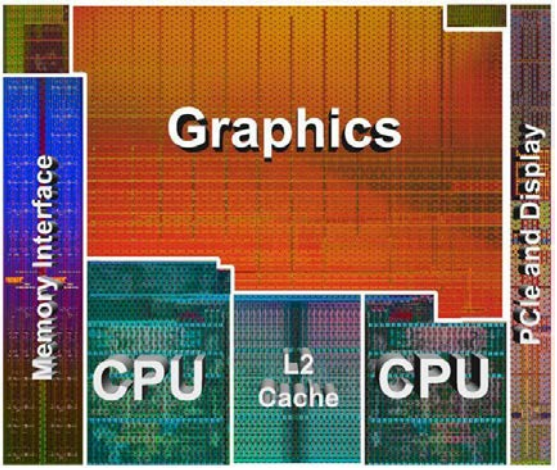



FIGURE 3.12: AMD's Kaveri die with two Streamroller CPU cores and a Sea Islands GPU, from [2].

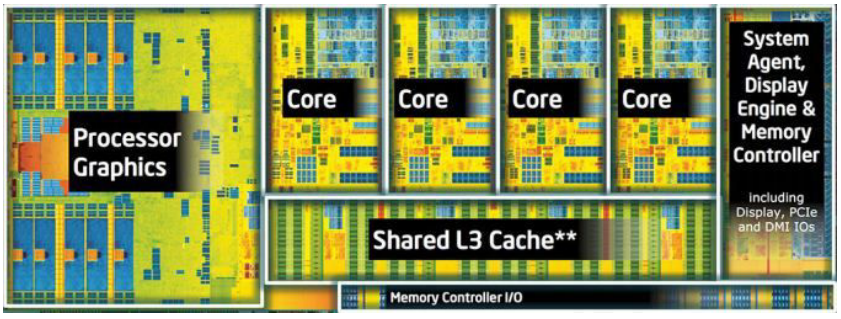



FIGURE 3.13: Intel's Haswell GT2 die with a HD Graphics 4200 GPU, from [2].

Finally, other vendors considered integrating CPUs and GPUs in the same silicon die in a design similar to that of the APU. The Intel Sandy Bridge, Ivy Bridge and Haswell processors are actullay a combination of CPU cores and integrated GPUs. Although the integrated GPUs of Sandy Bridge and Ivy Bridge were not often used for general purpose computations, the hardware design of the Haswell processor (see figure 3.13) may gain interest within the scientific community [7]. We can see from the die shot presented in the figure 3.12 that AMD has dedicated a proportionally larger amount of the chip to graphics than Intel has. Besides, AMD has put more GPU cores on the die than Intel has done. However, Intel has integrated more CPU cores, albeit generally slower in terms of frequency than the AMD CPU cores of the APU, on the silicon die. It is also to be noted that the Haswell memory system is different from that of the Kaveri APU as Intel has introduced a level 4 cache (for Iris Pro models only) that can be shared between the CPU and the GPU. Furthermore, NVIDIA has started to put serious efforts to rethink the interaction between GPUs and CPUs. On the one hand, NVIDIA has started pairing CPU cores and GPU cores on the same die. For instance the Tegra X1 is a combination of a Maxwell GPU and two to four ARM CPU cores designed by NVIDIA and code-named Denver-CPU. On the other hand, NVIDIA has released a high speed interconnect, labeled NVLink, that replaces the PCI Express bus and expect to ensure data sharing between a discrete GPU and a CPU at rates five to twelve times faster than the PCI Express Gen 3 interconnect.

To sum up, fusing CPUs and GPUs in the same die is a step in the right direction for efficient computations without data transfers overheads. However, the APU project is still a work in progress, the APUs are an order of magnitude less compute powerful than discrete GPUs and their memory bandwidth does not match that of the discrete GPUs.

## 3.2 Programming models in HPC

The trend towards heterogeneous computing has implied an increasing need for a specialized software infrastructure, such as parallel programming languages, compilers and APIs (Application Program Interfaces) to leverage massively parallel hardware. In this section we provide an overview of the common programming models that are used to target heterogeneous platforms in a scientific computation context. First we present the programming languages and APIs that are dedicated to develop applications on hardware accelerators. We particularly emphasize the abstractions provided by the Open Computing Language (OpenCL). Then we briefly present high level approaches such as directive based compilers.

### 3.2.1 Dedicated programming languages for HPC

#### 3.2.1.1 Overview

Many existing scientific applications have been adapted to make effective useof multi-core CPU platforms using a wide variety of programming models. OpenMP [11] is the de-facto standard for writing shared-memory thread-parallel programs. By means of a language extension based on high-level constructs and directives, OpenMP makes it

easier to parallelize loops and application hotspots amongst the CPU cores. OpenMP is usually used jointly with the Message Passing Interface (MPI) library in CPU clusters in order to exploit both the intra-node parallelism and the inter-node parallelism.
As an alternative to the OpenMP+MPI model, programming concepts that are based on the PGAS (Partionned Global Address Space) have arisen. PGAS is a programming model that assumes a global memory address space that is logically partitioned amongst a set of threads each of which has a local view of that space. For example, Coarray Fortran (CAF) [3], which is now integrated into the Fortran 2008 standard proposal, extends the Fortran language syntax with a construct called coarrays, which are essentially data structures that are shared between different images (processes) of a program. Accesses to these coarrays result in remote memory accesses. Similarly, UPC [10] is an extension to the C language offering benefits of the PGAS model to programs written primarily in C. In UPC, program instances are called threads and data is divided up between shared and private spaces. In addition, language qualifiers are provided which describe whether data is shared and how arrays should be distributed among the available threads. The number of threads can be specified at both compile and runtime. PGAS based approaches are attractive in terms of programmability but still suffer from portability issues.

When it comes to hardware accelerators, programming a parallel chip that is not a CPU can be challenging and requires unusual approaches. This can apply rather to any kind of accelerator, such as GPUs or FPGAs. While many of the technologies in general purpose GPU programming (GPGPU) are new, GPUs do have a relatively long history dating back to at least 1987 [47]. However, it was not until the beginning of the 2000's that the OpenGL [129] API and DX11 DirectCompute [157] added programmable shading to their capabilities, exposing GPU programming to the mass market. Until CUDA (Compute Unified Device Architecture) [4] and CTM (Close To Metal) [28][1] emerged in 2006, programmable shaders were practically the only way to program the graphics cards in mainstream computers. Shaders were not designed for general purpose computing and so put limitations on what could be done. While NVIDIA has improved the CUDA platform, AMD has evolved CTM to CAL (Compute Abstraction Layer) as part of the Stream SDK in December 2007 [28]. In June 2008, Apple and AMD along with various industry players in GPU as well as in other accelerator technologies formed the OpenCL (Open Computing Language) working group [19] under the Khronos Group. Khronos is responsible for other well known industry open specifications and made a choice to drive the OpenCL specification.
OpenCL and CUDA have since then become increasingly popular in the high performance computing community. They are extensions to high level languages (C, C++) which simplifies their learning curves. Theses technologies allow users to adapt complex scientific applications to GPU architectures by rewriting codes in an SPMD fashion. But, GPUs are architecturally different: GPUs from NVIDIA and AMD have different architectures and furthermore even GPUs from the same vendor are different. Hence, the user may need to optimize the same OpenCL or CUDA code for each individual architecture in order to get the best performance. CUDA and OpenCL have few differences in terminology, but otherwise CUDA and OpenCL written programs roughly operate similarly. The most important difference, and also the main reason to choose OpenCL, is that CUDA only targets NVIDIA GPUs while OpenCL is heterogeneous and can be ran on various GPUs, CPUs and other processors.

[1] CTM was introduced by ATI. AMD announced the acquisition of ATI Technologies on July 2007.

In the following we briefly describe the CUDA programming model and then give more details, in a separate section, about the execution and memory models of OpenCL. According to the NVIDIA literature [18], each multi-processor within a GPU contains multiple streaming processors or CUDA cores, that share the instruction stream, each of which has a pipelined arithmetic logic unit (ALU) and a floating point unit (FPU). The code that gets executed on the GPU is also called a *kernel*. This kernel is launched on a grid of thread blocks. The threads operate in warps, i.e. a collection of 32 threads that execute simultaneously the same instructions. The warps inside the same block can be synchronized, but no synchronization is possible between blocks. The different blocks are placed on the different multi-processors by a scheduler that deactivates stalled warps waiting for input to or output from memory and launches thread warps that are ready for execution to hide latency. The same physical multi-processor can execute several blocks, and the order in which blocks are assigned to multi-processors is undefined. The memory available on the graphics card is shared between threads and thread blocks. Each thread has its own registers, whose access is instantaneous (0 cycle) but whose total number is limited. Each thread block can use a small amount of low-latency on-chip shared memory, which can be read from and written to by all the threads of the same block. This is an efficient way for threads within one block to exchange data. Read/write accesses to it are always very fast (4 cycles), however the cost of some access patterns is higher than others.

#### 3.2.1.2 The OpenCL programming model

OpenCL is an industry standard used for task-parallel and data-parallel heterogeneous computing on a variety of modern platforms such as multi-core CPUs, GPUs, FPGAs, the Cell processor and other processor designs. OpenCL provides a software abstraction in the form of a compute model and a memory model that aim at characterizing the most common architectures of mainstream HPC platforms. Besides, OpenCL defines a set of core functionalities that is supported by all devices, as well as optional functionalities that may be implemented using an extension mechanism that allows vendors to expose unique hardware features and experimental programming interfaces. In practice, OpenCL offers a broad set of programming APIs and compilers to leverage heterogeneous compute facilities by exposing their underlying architectures to the programmers. Although OpenCL cannot mask significant differences in hardware architectures, it does guarantee portability. This makes it much easier for developers to begin with a correctly functioning OpenCL program tuned for one architecture, and adapt it to other architectures. It is to be noted that although OpenCL ensures the code portability, it does not guarantee the portability of the applications performance due to the rapid evolution of HPC architectures.

The figure 3.14 illustrates the OpenCL compute and memory models in details. The OpenCL programming model abstracts CPUs, GPUs, and other accelerators as *device*. OpenCL devices are usually driven by a commodity CPU refered to as a *host*. Each device comprises general compute units (CUs), each of which has multiple processing elements (PEs). The processing elements of the same CU use SIMD execution of scalar or vector instructions. Every instance of an OpenCL kernel (a function that executes on OpenCL devices), called work-item, executes on a PE, simultaneously with other work-items on the other PEs of the same device, and operates on an independent dataset. *Work-items* are further grouped into workgroups. Each workgroup executes on the same compute unit by groups of work-items. On AMD hardware, those groups are

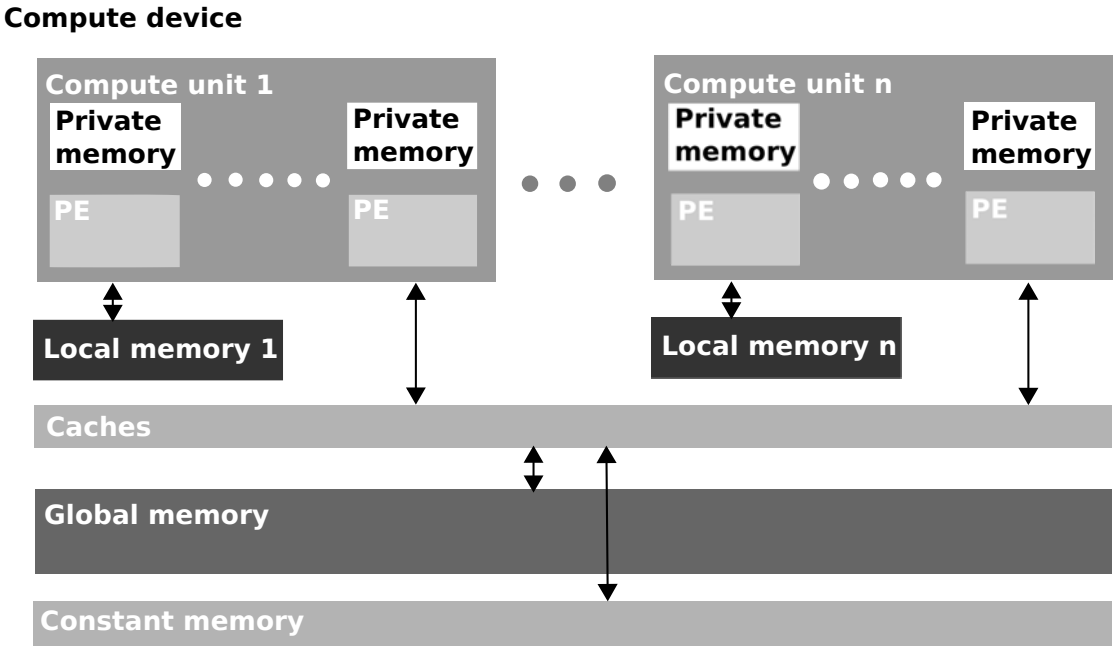



FIGURE 3.14: Illustration of the OpenCL compute and memory models.

called wavefronts and each has 64 work-items. Those in NVIDIA GPUs are the warps as defined by the CUDA programming model.
OpenCL defines four types of memory that devices may incorporate: a large high-latency *global memory*, a small low-latency read-only *constant memory*, a shared *local memory* accessible from multiple work-items within the same workgroup, and a *private memory* which is usually the register file accessible within each work-item. Local memory may be implemented using either high-latency global memory, or may be implemented with fast on-chip memory. Applications can query device attributes to determine the properties of the available compute units and memory systems, using them accordingly.

Before an application can compile OpenCL programs, allocate device memory, or launch kernels, it must first load an OpenCL *platform* (vendor dependant) and then create a *context* associated with one or more devices. The devices are picked by the programmer with respect to the application needs. Memory allocations are associated with a context rather than a specific device. Once a context is created, OpenCL programs can be compiled at runtime by passing the source code to OpenCL compilation functions as arrays of strings. After an OpenCL program is compiled, kernel objects can be extracted from the program. The kernel objects can then be launched on devices within the OpenCL context by enqueuing them into OpenCL *command queues* associated with the target device. OpenCL host-device memory transfers operations are also submitted to the device through the command queues. To illustrate the difference between a serial code and an OpenCL code, we present in figure 3.15 two implementations (in serial and in OpenCL) of the SAXPY algorithm. Note that the loop in the serial code is replaced by an SPMD code in OpenCL. For more information about OpenCL we refer the reader to [19] and to [29]. Note that for hardware accelerators, on accelerator clusters the mentioned programming models are usually used in conjunction with MPI in order to implement large scale accelerated applications.

```
void saxpy(int n, float a, float *x, float *y){
       for (int i = 0; i < n; ++i){
             y[i] = a*x[i] + y[i];
       }
}

__kernel void saxpy(float a,  __global float *x, __global float *y){
       int i = get_global_id(0);
       y[i] = a*x[i] + y[i];
}
```

FIGURE 3.15: A code snippet of the SAXPY algorithm implemented in serial (up) and in OpenCL (down). Only the OpenCL kernel is considered and the host code is not presented.

### 3.2.2 Directive-based compilers and language extensions

GPUs based systems have emerged as promising alternatives for high performance computing thanks to the high compute capability and high internal memory bandwidth of GPUs. However, their programming complexity poses significant challenges for developers. Several directive-based GPU programming models have been proposed such as OpenMPC [138], hiCUDA [108], PGI Accelerator [97], HMPP [83], OpenACC [13] and OpenMP 4.0 [14]. to provide a better productivity than existing ones. General directive-based programming systems usually consist of directives, library routines, and designated compilers. A set of directives are used to augment information available to the compilers, such as on mapping loops onto GPU and data sharing rules. Those directives are to be inserted into a piece of C, C++ or Fortran source code in order to automatically generate corresponding host and device codes (CUDA, OpenCL, assembly code ...).

Directive-based models provide different levels of abstraction, and require much less programming efforts. The most important advantage of using directive-based GPU programming models is that they provide very high-level abstraction on GPU programming, since the compiler hides most of the complex details specific to the underlying GPU architectures which helps programmers focus on the productivity of their programs and easily maintain legacy codes. However, a lower level of abstraction is not always counter productive, since it may allow more control over various optimizations and the features specific to the underlying GPUs to achieve optimal performance. An extension to HMPP [83], called HMPPcg, has been proposed in order to give more control to the user at a lower level of abstraction. It is to be noted that despite attractive, merly inserting directives into real world scientific applications often do not outperform the performance provided by dedicated programming models such as CUDA and OpenCL [205].

## 3.3 Power consumption in HPC and the power wall

Until recently, most HPC facilities have focused more on the search for performance rather than on power efficiency. With the cost of power increasing throughout the years (see figure 3.16), it is becoming crucial to also evaluate HPC systems on power consumption grounds. For that to do practical methods have to be established to measure

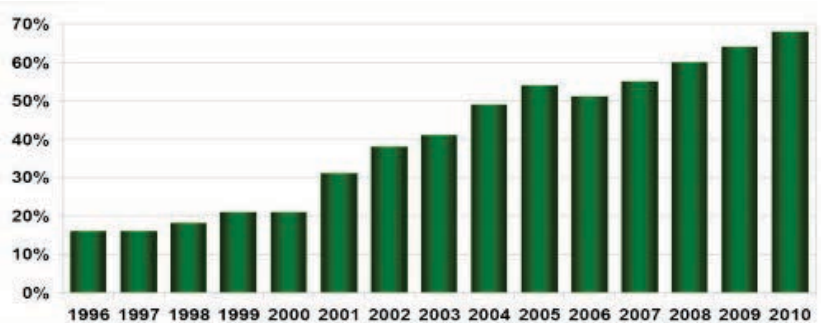



FIGURE 3.16: Spending of power as a ratio to server spending. Data provided by IDC (International Data Corporation) based on the average configuration of data-centers of 6000 nodes. From [68].

power consumptions of such systems. Today, one of main prerequisites to hit the exascale computing era is to tear down the "power wall". In this section we briefly underline the physical limits that have historically impacted the design of HPC facilities.

We have mentioned that single thread performance scaling was given by Moore's law. In conjunction with Moore's law, Dennard scaling, i.e. a scaling law named in 1974 after Robert H. Dennard, states that transistors power use is proportional to the die area as the transistors get smaller [161]. The relationship between processors frequency and power is emphasized by Martin [161] and defined as $P = fCV^2 + VI_{leakage}$, where $P$ is the power in Watts, $f$ the processor frequency, $C$ the total capacitance, $V$ the supply voltage and $I_{leakage}$ the leakage current. $VI_{leakage}$ is the static power dissipation and was usually neglected for device structures larger than 65nm [161] when evaluating the power consumption. In practice, this is no longer sustained since 2005 because of the exponential growth of the leakage current [161] which can't be neglected anymore as the vendors moved to sockets of size less than 40nm and below. Semiconductor processes are about to reach a physical limit, in terms of overheating, of what silicon can withstand with current cooling techniques, consequently the processors frequencies are stalled (see figure 3.2). Processors technology can no longer be counted upon for advances in terms of frequency, instead the focus must shift to energy efficiency. This limitation is relatively alleviated by the evolution the multi-core CPUs throughout the late years, as CPU cores are lowered in terms of frequency and are more an more multiplied on the silicon dies.

Today, processor designers are facing a new challenge when trying to design future exascale systems that are required to deliver 50 $GFlop/s/W$ [212]. In order to reach this target, energy efficiency must improve by a factor of over 20× given that current multi-core CPUs offer between 1 to 2 $GFlop/s/W$ (current GPUs offer between 3 to 5 $GFlop/s/W$). Villa et al. [212] predicted that processor technology improvement alone will only provide approximately 4.3× of the required improvement in energy efficiency, that an additional 1.9× factor can be extracted from circuit improvements, enabling operation at lower $V$, and that a further 2.5× contribution to energy efficiency must come from the design of system architectures (memory, disks, interconnect etc.).
Although power hungry (between 230 and 300 W of maximum power consumption as opposed to approximately 150 W TDP[2] for high-end multi-core CPUs) modern GPUs, with their low frequency and high density in terms of processing cores (more compute powerful than CPUs), are proved to be more power efficient than CPUs for a broad range of applications. Several researchers have conducted comparisons between the

[2]Thermal Design Power

power consumption of GPUs with that of CPUs and other processors such as Cell or FPGA [53, 92, 114, 170, 211, 224]. For massively parallel applications GPUs have been found to be more energy efficient than CPUs. Besides, APUs, with a TDP ranging between 60 W and 95 W (for the Desktop and Server lineups) and a single precision peak performance hitting the 1 Tflop/s, can be an attractive solution to be considered to implement energy efficient scientific applications as its estimated theoretical power efficiency is approximatively 10 $GFlop/s/W$. Note that these expectations are conditional and depend on how far are the sustained power efficiencies (measured) from the theoretical numbers.

# Chapter 4

# Overview of accelerated seismic applications

**Contents**



In this chapter a selection of seismic exploration related applications is given. We focus on the optimization carried on those applications by means of High Performance Computing. We start in section 4.1, with a building block of wave equation solvers: stencil computations. Then, we continue with a summary of accelerated implementations of the Reverse Time Migration, in section 4.2. These implementations are driven by the introduction of more physics in the wave equation solver, by advances in High Performance Computing and by a series of technical challenges that characterize the RTM workflow, i.e. a high computation cost, a tremendous memory requirement and extensive I/O operations. We close the chapter in section 4.3 with a quick presentation of some state-of-the-art applications whose workflow is similar to that of the RTM or which are based on wave equation like solvers.

## 4.1 Stencil computations

Stencil computations (SC) are a building block and a common kernel to a broad range of scientific applications. More importantly, they are at the heart of seismic modeling and RTM applications. Although this class of computation is characterized by regular and predictable memory access patterns, it has a low flop/byte ratio which requires to precisely tune the data layout and optimize the memory accesses according to the different hardware architectures. Therefore, extensive efforts have been made to optimize SC on homogeneous and heterogeneous architectures. In this section we give an overview of state-of-the-art implementations of SC.

On the CPU based node level, some popular techniques are often used, including cache oblivious [87], temporal blocking [160], SIMD vectorization and register blocking [81], to reduce the different cache levels misses and ameliorate the performance of SC on CPU. A multi-level SC parallelization approach is proposed in [80]. It combines inter-node by domain decomposition, intra-chip parallelism through multithreading and data-level parallelism by making use of SIMD vectorization. Datta et al. [73] introduce the cache blocking and the time skewing techniques for SC optimization. The authors used these techniques to effectively maximize cache resources by tiling in both the spatial and temporal dimensions. In [81], the authors added to these techniques software prefetching (by using intrinsic functions provided by compilers) to reduce the memory accesses overhead and increase temporal locality in SC. Other researches that are focusing on the same optimization techniques for SC can be found in [149, 164, 180, 203, 222]. To learn more about how these techniques are practically used in SC we refer to Farjallah [84] where a thorough description about the time skewing and cache oblivious algorithms is given. Some works are focusing in performance modeling of SC, such as in [133, 137] and in [201] where an ECM (Execution-Cache-Memory) performance model for SC is presented.
On the CPU multi-node level, MPI based SC implementations were proposed. For instance, in [54], the authors implemented SC based on a 3D domain decomposition, using MPI, on an Intel Xeon 5355 CPU cluster composed of 128 nodes. In addition to the optimization techniques such as the temporal blocking, cache oblivious and vectorization that were applied locally to each MPI sub-domain, the authors relied on overlapping the MPI communication with computation in order to achieve good strong scaling. The authors reported that up to 32 nodes the strong scaling was linear and that the performance decreased for 64 and 128 nodes due to the increase of the network traffic that became very high compared to the SC computation time.

SC are also constantly adapted to accelerators and heterogeneous architectures. On the GPU based single-node level, the natural parallelism offered by the GPU hardware design, including the DLP while executing on the GPU SIMD arrays, and by the GPU thread schedulers, including TLP while overlapping the executions of the wave-fronts, is exploited. Besides, Volkov [217] showed that the ILP can be also employed, by loop unrolling and instruction interleaving, in order to increase the occupancy rather than by filling the GPU cores with more and more threads. Furthermore, Micikevicius [156] introduced, a short time after the emergence of the CUDA programming model, an advanced parallelization of the 3D finite difference computation (based on SC) on NVIDIA GPUs. He made use of a 2D grid of threads that traverses the 3D data volume slice-by-slice, relying on the GPU shared memory and on the data access redundancy (that characterizes SC) to maximize the throughput. He demonstrated a performance of 2.400 millions of output points per second on a single Tesla 10-series GPU (out of a maximum theoretical throughput of 3.000 millions). He also introduced the register blocking technique: a sliding memory window is privately used by each GPU thread to reduce the memory access penalties when traversing the slowest memory dimension. Additionally, in [120] the authors extended Datta et al. work for GPUs by proposing a multi-level SC parallelization framework targeting GPUs (single nodes). For that to do, the authors introduced the out-of-core technique to allow processing domains bigger than a single GPU memory capacity (each of their GPUs features 3 GB of GDDR5 SDRAM).
Given that SC are subject to a relatively little computation performed per grid point, one of the primary performance challenges is efficient data movement. There are two aspects to efficient data movement: data transfer between the host memory and device

memory through the PCI Express bus, and onboard data transfer between the device memory and the GPU cores. Hiding the latency of the latter is covered by the use of the GPU shared memory and the use of register blocking as discussed above. However, the bandwidth of the PCI Express bus is over an order of magnitude slower than the bandwidth between the GPU memory and the GPU cores, and its latency remains a bottleneck for SC [23]. In [147], the authors demonstrated that the PCI Express bus memory traffic incurred up to 23% of performance loss of SC on an NVIDIA GTX 590 GPU. The impact on the performance is found even more important with the out-of-core technique, which requires extensive memory traffic between the CPU and the GPU, as mentioned in [121]. Therefore special emphasis needs to be given to limiting and overlapping PCI traffic. In SC one approach is used to limit the PCI traffic by running as much stencil computations on the device as possible. This is referred to as the temporal blocking method [72]. Another method that is used in SC, is the asynchronous transfers which can be used to overlap PCI memory transfers with computation on the device (and host) [121].
On the GPU multi-node level, a variety of studies focused on coupling GPU programming models with MPI in order to leverage large scale GPU clusters for SC. Multi-GPU implementations require dividing the subdomain of each MPI process into its interior and boundary regions. Exchanging the boundary regions between the different GPUs incurs memory traffic on the PCI Express bus and MPI communications between the compute codes (usually performed by the CPU). In [150], the authors introduce Physis, an automatic translation framework that generates and MPI+CUDA SC code based on a used-written code. The framework automatically manages the MPI communications and the boundary exchanges including the memory transfers through the PCI Express bus. It also allows the programmer to overlap the PCI Express memory transfers with the computation of the interior regions. The authors showed that the framework ensured 60% of the performance of a hand-tuned SC code on 256 GPU nodes of the TSUBAME 2.0 cluster. Similar frameworks such as PARTANS and SkelCL, a framework that is based on the OpenCL programming model, are proposed in [147] and [43]. In [72], the temporal blocking method is extended to the multi-GPUs implementation of SC in order to reduce the MPI communication overhead. Furthermore, in [118] the authors propose an SC algorithm where they overlap both the PCI Express memory traffic and the MPI communication with computations on the GPU. They show a sustained performance of 2.4 $TFlop/s$ and a linear strong scaling on 64 nodes of the NCSA Lincoln Tesla C1060 based cluster. A similar approach is presented in [197], where the authors used the communication-computation overlap to achieve a single precision performance of 1.017 $PFlop/s$ on 4000 GPU nodes of the TSUBAME 2.0 cluster.

Finally, some researches such as in [225] compare the performance of SC on cutting edge CPU and GPU technologies (on a single node). A highly efficient parallel vectorized (using SSE and AVX) implementation was developed on Intel Sandy Bridge CPU (i7-2600K) and on AMD Bulldozer (FX-8150). On the GPU most of the techniques described above were applied on NVIDIA Tesla M2070 and GTX 480 GPUs. A comparative table, between the performance of four NVIDIA GPUs and four Intel CPUs, was presented where the authors show that the GPU implementation offers a 3× speedup compared to the CPU implementation in single precision.

To summarize, given that SC have a low compute intensity, efforts are generally invested to minimize the data access redundancy. On the GPU implementations the

PCI Express bus is often identified as the bottleneck that hinders achieving good performances. Techniques such as out-of-core, temporal blocking, and register blocking are often used to optimize SC performance on GPUs.

## 4.2 Reverse time migration

We try throughout this section to describe the common trends followed to efficiently implement RTM algorithms in a HPC context. First, we underline the evolution of the complexity of RTM algorithms during the last decades. Then, we detail the different strategies used to circumvent well known bottlenecks such as the massive memory demand. Finally, we enumerate a selection of the state-of-the-art RTM implementations and related works on mainstream CPUs and also on hardware accelerators.

### 4.2.1 Evolution of RTM algorithms

Historically, the evolution of seismic imaging algorithms was dictated by the advances of the hardware design and compute capabilities. Indeed, the development of high performance facilities and storage capacities allow to design and implement more and more advanced RTM algorithms.

- More physics is introduced into the numerical solvers as complex media such as anisotropic and elastic are considered. This can drastically increase the complexity of the imaging algorithms and the compute power requirements. Figure 4.1, extracted from [48], depicts the compute resource requirements as a function of the complexity, in terms of computation, of the imaging algorithms over the recent years. Resource needs increase exponentially as the explored media is more subject of geophysical complexity. The figure is also an illustration of the trends in seismic imaging from asymptotic imaging to the visco-elastic full wave inversion (FWI), i.e. an algorithm used for velocity estimation based on information extracted from seismic traces.

- RTM where the simulation frequency, i.e. the seismic source frequency, ranges from $30Hz$ to $50Hz$ and above, which is considered as a high frequency, is nowadays feasible. This produces high resolution subsurface images with neater seismic attributes which leads to a better reservoir definition. Figure 4.2 presents three different images obtained after running RTM with three different frequencies. It shows the impact of the high frequencies on images resolution and the estimated needs on compute power.

- The large compute power available today implies the evolution of the seismic data used as input for imaging. Indeed, rich azimuth data recording methods are more often used such as WATS (*Wide Azimuth Trailed Streamers*) or coil shooting for full azimuth coverage [101]. Those extend the conventional *Narrow Azimuth Towed Streamers* surveys used in the oil and gas industry and require more compute power.

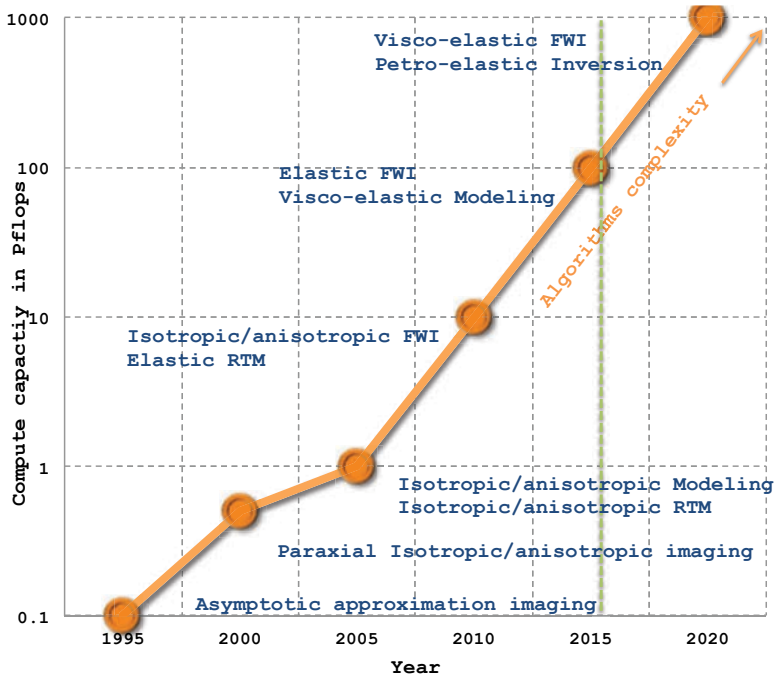
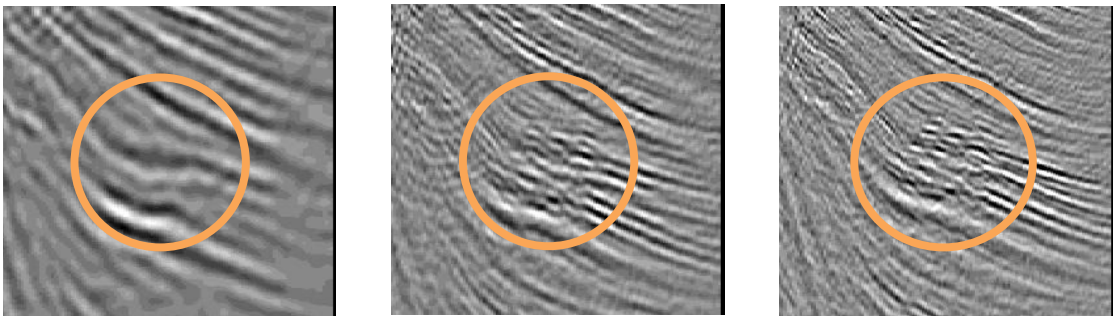



FIGURE 4.1: The evolution over the last two decades of the seismic imaging algorithms with respect to the technology and compute power advances throughout time.

### 4.2.2 Wave-field reconstruction methods

The high-level workflow of the Reverse Time Migration was schematized by the flowchart presented in the figure 2.12 in section 2.2. To apply the imaging condition, RTM needs the source wavefield and the receiver wavefield at the same time, e.g. at each time-step from the initial time-step until the maximum recording time. Given that the source and receiver wavefields are advanced along opposite time directions, the propagation history (*snapshots*) of the source wavefield needs to be saved. This means that we need to save these values for each time step which results in tremendous usage of memory



(a) RTM with an average frequency of $18Hz$, needs at least 100 Tflops of compute capacity

(b) RTM with a high frequency of $35Hz$, needs at least 1.5 Pflops of compute capacity

(c) RTM with a very high frequency of $55Hz$, needs at least 10 Pflops of compute capacity

FIGURE 4.2: Impact of the RTM frequency on the image resolution and on the high performance resources requirement. Example performed on a 3D model of size $1700 \times 860 \times 600$. Source Total S.A.

space [142]. The wavefield snapshots are usually too big for the volatile memory and for the storage capacity of the present hardware: take an example of a 3D grid model of $1000 \times 1000 \times 500$, the number of time-steps is usually in the order of 10000, which means that to save the propagation snapshots we need about 20 $TB$ of memory or hard disk space. As a consequence, effective algorithms are adopted to reconstruct wavefields before the imaging condition stage, in such a way that large migrations could be performed within minimal memory configurations. Anderson et al. [31] summarize the common trends to alleviate the memory requirements of RTM. Dussaud et al. [82] also describe these approaches and give their computation and storage complexities. Hongwei et al. [112] give pros and cons of each method and introduce new ones that are hybrid combinations of the conventional methods.
In the rest of this section, we quickly review six of the different algorithms used to reconstruct the receiver wavefield. The discussed solutions tend to find the right trade-off between storage and computation time, while making compromises that are satisfactorily for imaging. We do not present them in a particular order, rather we approximate the computation complexity of each one of them, along with the storage complexity and the impact on the final images quality.

#### 4.2.2.1 Re-computation of the forward wavefield

At each time-step during the backward modeling, the source wavefield is re-propagated forward from $t = 0$ until the current time-step is reached. Then the imaging condition for the current time-step is computed by correlating the source and receiver wavefield. In terms of computation, this method has a theoretical complexity of $\sum_{t=0}^{N-1}(N-t) = \mathcal{O}(N^2)$ [207] which is an unacceptable computational burden. No storage of the source wavefield is needed during the forward modeling. However, since the source and receiver wavefields are correlated at every time-step, the migrated image is at its highest quality.

#### 4.2.2.2 Storing all the forward wavefield

The source wavefield is saved after each time-step during the forward modeling. During the backward sweep the wavefield snapshots are loaded from memory or disk and are correlated with the receiver wavefield values in order to produce the final image. This approach requires prohibitive storage space and extensive I/O operations (the storage complexity is $\mathcal{O}(N)$) and thus is not appreciated in an industrial context where real datasets are used. The computation complexity of this option is only $\mathcal{O}(N)$ [207] but it requires much larger amounts of storage. Similarly to the previous algorithm, this method ensures high quality images.

#### 4.2.2.3 Selective wavefield storage (linear checkpointing)

During the forward modeling, the source wavefield is stored every $k^{th}$ time-step with $k > 1$. The choice of the value of $k$ is based on the Nyquist sampling theorem. During the backward propagation, the needed forward value is loaded every $k^{th}$ time-step and the imaging condition is performed at the same frequency. At the end of the day the missing parts of the final image are filled by interpolation. This method is adopted by a number of commercial RTM implementations. The computation complexity of

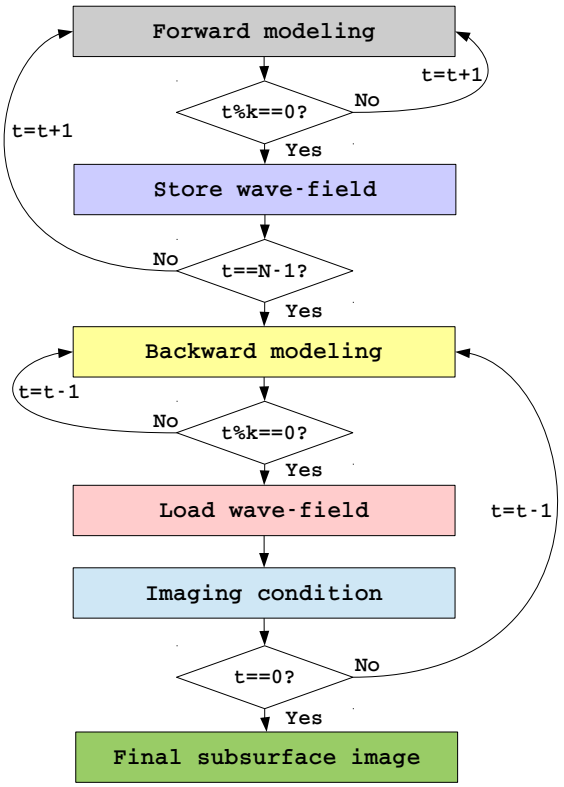



FIGURE 4.3: The flowchart of the linear checkpointing strategy for RTM. $t$ is the time-step index and $k$ is the number of time-steps that separates two successive store operations of the source wavefield.

this method is the same as the previous one ($\mathcal{O}(N)$) but offers a $k-$fold reduction in memory usage ($\mathcal{O}(N)/k$)[207] at the expense of image accuracy. Sun et al. [220] give more details about this method. Given that this method is the most used in the industrial and commercial seismic solutions, we illustrate its workflow in the flowchart 4.3.

#### 4.2.2.4 Checkpointing

The idea behind checkpointing in RTM is based on the adjoint state method [176]. The imaging condition is seen as an objective function whose adjoint needs to be computed using a time-reversal method proposed by Griewank [95, 96] and ameliorated in [30, 31]. Griewank's method has application to automatic differentiation of computer algorithms, Symes [207] used it to advance the optimized "checkpointing" method for RTM. During the forward modeling, some time points are set as checkpoints. During the backward propagation the source wavefield values are reconstructed by forward modeling from the nearest checkpoint in time. The receiver wavefield is retro-propagated as described in the previous methods. Several source wavefield values are computed several times in this method, but some internal buffers can be used to reduce the "re-computation ratio" (the number of times the same wavefield value is computed). The trade off between the number of checkpoints and the number of buffers should be carefully chosen for the checkpointing to be optimal, as the recomputation ratio is too high if the number of checkpoints is not enough, while the memory demand is too high if too many checkpoints are set. The computation complexity depends on the number of the checkpoints ($c$) and

on the number of the intermediate buffers ($b$). The storage complexity of this approach is $\mathcal{O}(N)/c$. However, it is to be noted that the high quality of images is preserved since no interpolation is involved.

#### 4.2.2.5 Boundaries storage

Clapp [62] suggested another approach to reconstruct the source wavefield during the backward modeling. His approach concerns the computational grid with damping zones (boundary conditions). Damping is required to remove the artifacts due to reflections on the grid boundaries (see section 2.3.3.3). During the forward propagation, only the slices that separate the damping zones from the rest of the compute grid are saved. During the backward modeling, the source and receiver wavefields are progressed in the same direction. The source wavefield is thus reconstructed by back-propagation after restoring the boundary slices from memory or disk. This approach reduces significantly the storage space needed even if it implies more computation complexity. This method is further investigated by Dussaud et al. in [82] and by Hongwei et al. in [112].

#### 4.2.2.6 Random boundary condition

Another approach to circumvent the I/O bottleneck of the imaging condition is the Random Boundary Condition (RBC) introduced by Clapp [61]. RBC increases the computation complexity but suppresses the need to store the source wavefield during the forward modeling. Clapp proposes another boundary condition that distorts the reflections in order to minimize artifacts rather than damping the energy whenever a wavefield hits the boundaries. During the forward modeling the source wavefield is propagated to the maximum record time and is pseudo-randomized on the grid edges. During the backward step, the source and the receiver wavefields are back-propagated simultaneously. Fletcher et al. [85] propose a similar approach with time-varying boundary conditions.

Apart from these wavefield reconstruction methods, there are many researchers trying to solve the I/O problem for RTM differently. For instance, Guan et al. propose a multi-step RTM [98] algorithm, where the velocity model is divided into regions in depth with respect to its geological complexity (water layer, shallow sediments etc.). RTM generates an image for each region separately and then the final image is aggregated. This multi-step approach reduce the memory requirement since the computation and storage complexity are driven by the minimum velocity of each region.
Data compression is another solution, and can be used with all the previous methods in order to accelerate the I/O operations. Sun et al. [220] introduce a strategy based on a loss-less compression algorithm that reduces storage significantly at a little computational cost.

### 4.2.3 RTM on multi-cores and hardware accelerators

Reverse time migration is at the heart of seismic imaging algorithms, a workflow that is known to be time consuming and to generate a tremendous amount of temporary data during computation. Over the recent years, the HPC ecosystem has become so diversified that it is difficult to choose the right hardware that may deliver the most

cost-optimal and high performance seismic imaging solution.
Along with the solutions based on multi-core CPUs, a proliferation of alternatives has emerged. Some of them leverage the hardware accelerators such as GPUs and FPGAs (Field Programmable Gate Arrays), others exploit the emerging Intel Xeon Phi technology to prepare the RTM for the exascale computing era [165]. Besides, many metrics can be used to evaluate the performance of these applications, namely the floating point performance, the cost in terms of hardware, the power consumption etc.
The parallelization of the RTM application starts at the shot level. The computation of shots being independent, multiple shots are usually processed simultaneously and the final image is obtained by the aggregation of the results of each one of them (in the case of pre-stack migrations). Researches thus often focus on optimizing the workflow conducted in one shot. Within a single shot, if the computation holds in a single node, then the shared memory programming models like OpenMP for example are used. Otherwise, domain decompositions are usually applied on distributed memory machines (often using MPI).

#### 4.2.3.1 RTM on multi-core CPUs

The evolution of multi-core CPUs and the rise of novel technical features, such as the use of wide SSE/AVX vectors, the use of instruction level parallelism (ILP) as well as the hardware and software prefetchings, push researchers to constantly redesign RTM algorithms. Efforts are put at the different levels of mainstream CPU clusters architecture: at the network (inter-node) level, at the node (intra-node) level and at the core/thread level by leveraging SIMD capabilities, i.e. vectorization.
On the single-node level, a TBB [117] based implementation of a pre-stack Reverse Time Migration, on a 8-CPU shared memory computer, is presented in [119]. The authors rely on the re-computation of the forward wavefield to perform the imaging condition. Their implementation offers a speedup of 4× compared to a sequential implementation.
On the multi-node level, some CPU implementations rely on a large scale design on distributed machines. In [213], the authors propose a distributed 3D acoustic finite difference modeling algorithm using a 1D domain decomposition approach and implemented it using the PVM message-passing library. They demonstrate parallel efficiency up to 94% on an IBM SP-2 cluster. In [175], the authors study the same algorithm and propose an MPI implementation where they compare the domain decomposition strategies. They proved that checkerboard partitioning (3D domain decomposition) gives the best performance and that it has the most suitable memory access pattern for such studies. In [172], the authors introduce a counter intuitive approach to optimize a 3D isotropic RTM code on a Blue Gene/P (BGP) supercomputer, based on over partitioning the computational domain. Over partitioning, which means that the global domain is partitioned into very small sub-domains (1000 times smaller than for standard RTM), helps storing the wavefield snapshots on the main memory of the compute nodes, suppressing the need for hard disks. The high speed network (5.22 $TB/s$ per rack) that features BGP attenuates the overhead incurred by communications. The authors also present optimizations including load-balancing, cache tiling and SIMD. They demonstrate throughput of up to 40 *Billions* stencil updates per second per node along with strong scalability. In [145], the authors extended the research conducted on BGP to the next cluster generation. They advanced a multi-level parallel RTM algorithm on a Blue Gene/Q machine that peaks at 204.8 $TFlops$ in single precision. They achieved a 14.93× speedup over the performance previously obtained on BGP machines [171, 172].

In [57], the authors implement the 3D seismic modeling problem and 3D RTM, using the Fourier method to solve the wave equation and MPI to deploy the applications on a Linux cluster. The implementations make use of all-to-all MPI non-blocking communications. The authors demonstrate that a performance improvement of up to 40% can be achieved by using non-blocking all-to-all communications and by introducing the MPI communication-computation overlapping capability. In [178] the authors report their experience of optimizing a seismic acoustic modeling application on the multi-core architecture of PARAM Yuva II [15]. Their application was developed using a hybrid OpenMP +MPI programming model. They used MPI for domain decomposition across parallel nodes. At the node level the performance was improved using OpenMP rather than MPI as they showed that OpenMP offered a better performance on the node level.

#### 4.2.3.2 RTM on GPUs

In search for more performance, application developers have turned to hardware accelerators to implement seismic applications. Thanks to their huge compute power and more importantly to their high memory bandwidth which is typically an order of magnitude higher than that of high-end CPUs, various RTM implementations on GPUs have emerged during the last years.
For GPUs, the challenge is often to optimize the small memory space available on the GPU devices (compared to the size of the datasets used in RTM) and to mitigate the overhead due to data transfers through the PCI Express bus when saving wavefield snapshots on the intra-node level, and when exchanging the boundary regions on the inter-node level. As a matter of fact, in [166] the authors conducted strong scaling experiments on a large scale GPU implementation of a 3D acoustic RTM in the TSUBAME-2.0 [210] cluster. They showed that the speedup in strong scaling was not proportional to the number of GPUs ($N$) but rather is proportional to $N^{2/3}$ because of the PCI Express overhead generated by the data snapshotting. In [86] an overview of an implementation of RTM on NVIDIA Tesla C1060 GPUs using the CUDA programming model is presented. A multi-GPU variant is also presented in order to process a set of Rich-Azimuth data. The authors advance a 4× speedup by comparing four GPUs against eight CPU cores. The authors also underline a computational overhead in moving data to and from GPU devices. In [111], the authors present a proof of concept of a pre-stack RTM on GPUs. The GPU performance is shown to be an order of magnitude higher than that of a traditional CPU. In [142], the authors also developed a pre-stack RTM algorithm on the NVIDIA Tesla C2050 GPU. The FDM was used to solve the wave equation and a TTI media is considered. The authors adopted a random boundary condition in order to circumvent the I/O bottleneck of RTM. In their approach RBC reduced the CPU-GPU traffic via the PCI Express by almost 50% at the expense of extra computations. Only the 2D case was studied, for the 3D case a hybrid pseudo spectral/finite difference approach was targeted as a future work. A similar work was achived by the authors of [130], but the wave equation was fully solved by using pseudo spectral numerical methods. In [23], the 3D acoustic seismic modeling and RTM are implemented in CUDA and deployed on a GPU cluster (5 NVIDIA Tesla S1070). The implementation is based on a finite difference approach on a regular mesh for both 2D and 3D cases using single precision computations. The authors showed a 30× (resp. 10×) performance speedup compared to a CPU cluster (10 Intel Xeon 5405 CPUs) for modeling (resp. for RTM). This research was extended by Abedlkhalek in [22] where the author conducted strong scaling tests of the seismic modeling (equivalent to the forward modeling, the first step

of the RTM algorithm) on 16 NVIDIA Tesla S1070 GPUs, and showed that the MPI communications (plus the PCI Express bus overhead generated to retrieve the boundary regions from the GPU memory to the CPU memory) hindered the achieving of a linear scaling. In [155], the authors accelerate a 3D finite difference wave propagation in a heterogeneous elastic medium on 4 NVIDIA GPUs. Despite the complexity of putting together the elastic wave equation solver, the authors demonstrated a speedup between 20× and 60× compared to a serial CPU implementation. In [162], the authors present performance optimizations for TTI RTM algorithm on hybrid CPU+GPU (NVIDIA M2090) based architectures and demonstrate around 4× performance gain over CPU only runs. The imaging condition is handled by the CPU while the wave propagation is performed by the GPU. Similarly, in [168] the authors present a hybrid CPU+GPU approach to implement the Kirchoff migration algorithm. The authors used a cluster of 64 nodes each of which has an NVIDIA C1060 GPU and a Xeon 5410 CPU and demonstrated a speedup of 20× compared to a CPU only version. Other related works can also be found in [219].
Finally, some researches are focusing on leveraging GPUs using high level programming models such as directive based ones (OpenACC, etc.). Sayan et al. [93] share their experience about high-level directive based GPU programming models in a seismic context. They implement building blocks of the RTM algorithm in OpenACC. The authors conclusions suggest that the OpenACC implementation offers a 1.7× speedup compared to a highly optimized OpenMP CPU code. However, it is 1.5× slower than a hand-tuned CUDA code. Besides, in a study [40], conducted by CAPS Entreprise jointly with Total, Bihan et al. made use of the HMPP [83] workbench to accelerate an MPI+Fortran implementation of RTM on an NVIDIA Tesla S1070 server with four GPUs and two bi-socket Intel Hapertown CPUs (each socket has four cores). HMPP was only used to manage the memory transfers between the CPUs and GPUs, namely to partially overlap the PCI Express memory traffic with GPU computation. The algorithm core functions were hand-written and hand-tuned in CUDA an plugged into the HMPP workbench. The authors advanced that one GPU is 3.3× faster than eight CPU cores, and that the performance of the four GPUs is 2.1× higher than the sixteen CPU cores. They also reported that the accelerated implementation on GPUs scales less efficiently than that on the CPU cores, due to the extra cost of the communications between the GPUs that need to go through the hosts CPUs.

#### 4.2.3.3 RTM on other accelerators

GPUs are not the only accelerators that are used in an exploration geophysics context. Researchers rely also on co-processors such as the Intel Knight Corner (Xeon Phi). In [84], Farjallah conducted a co-design study applied to the Reverse Time Migration in isotropic and anisotropic media. A cluster of Intel Knight Corners (*Stampede* hosted in TACC) was used to adapt the RTM algorithm to the many-core architecture and evaluate its performance. The author used a hybrid MPI+OpenMP programming model. Other works focused on porting the RTM workflow on Xeon Phi namely in [34] and in [178].

FPGAs (e.g., Xilinx [20], Altera [17], etc.) are different accelerators based on a reconfigrable hardware [109]. Given that the FPGA substrate can be configured to perform highly parallel tasks with a high degree of customization in a very low power envelope, FPGAs have become an attractive option for seismic applications. However, programming with FPGAs can be challenging and thus high level programming models (that support FPGAs) are needed to exploit this hardware design. We can refer to

a selection of works that focused on optimizing RTM and other seismic applications on FPGAs, namely [51, 56, 79, 88, 185, 195]. The common trends when it comes to implementing seismic applications on FPGA, is to use software pipelining to increase the throughput and fixed-point data representation to reduce the memory usage. Many authors reported a speedup between 20× and 30× of FDTD (Finite Difference Time Domain) applications, compared to a 3 $GHz$ commodity CPU.

Another hardware design that attracted the O&G industry is the Cell/B.E. architecture, proposed by IBM, Sony and Toshiba [124]. For example, in [32] the authors present a mapping of the computational kernel of RTM on the IBM Cell/B.E. processor. Their implementation achieved close-to-optimal performance, and the kernel (proved to be memory-bound) achieves a 98% utilization of the peak memory bandwidth. The Cell/B.E. implementation outperforms a traditional processor (PowerPC 970MP) in terms of performance (with an 8× speedup) and in terms of energy efficiency (with a 5.3× increase in the GFlop/s/W delivered).

In a different fashion, some researchers conducted more general studies to show how a key application in the O&G industry can be effectively mapped to different accelerator architectures, and to analyze performance and drawbacks. Generally, comparisons in terms of architectural design, power consumption and programmability are presented. In [33], the authors give a comparative study of different implementations of a 3D acoustic RTM over homogeneous processors (multi-core platforms) and over a selection of hardware accelerators: a Cell/B.E. processor, a GPU and an FPGA. They demonstrate that the accelerators outperform traditional multi-cores by one order of magnitude. However to achieve this a great development effort is required, mainly because the programming environments are too specialized. The authors recommend the need for automatic tools and high level compilers for accelerators in order to allow a more feasible scientific programming on these platforms. Clapp et al. [63] identify four potential implementation bottlenecks that characterize the second-order acoustic RTM algorithm.
First, they claim that optimizations and algorithmic techniques are concentrated on reducing the total number of operations and figuring out how to minimize the need to read from or to write to disks. They enumerate optimization techniques used for their RTM CPU implementation such as cache-oblivious approaches and seismic properties compression.
Then, they give some implementation details on streaming computing devices such as GPUs and FPGAs. According to the authors GPUs are suitable for small problems since their implementation suffers from multiple limitations. The amount of memory, throughout the different levels of the GPU memory hierarchy, is limited which constraints the size of the stencils and makes techniques such as data compression hard to put together. Large domains have to be spread through multi GPUs creating a bottleneck either over PCI, especially when storing the wavefield snapshots, or the network. Complex boundary conditions are less practical because of the SIMD programming model. The correlation phase (imaging condition) is also constrained by the memory bandwidth.
Finally, they present optimizations techniques on FPGAs such as data streaming on each direction through the 3D volume, multiple steps streaming, hardware support for reconfigurable number format and bit width where they demonstrate that 24-bit fixed point numbers achieve acceptable accuracy for RTM. But they underline that FPGA programming is complicated even although in recent years various efforts has been put to develop high-level compilers for FPGA programming.

## 4.3 Close to seismics workflows

Finite-difference techniques in the time domain (FDTD) applies also to another variety of numerical simulations in different scientific fields, such as computational fluid dynamics, astrophysics, electromagnetics and earth gravity. For instance, in [69] the authors propose a finite difference GPU implementation to simulate the calculation of the forward modeling of gravitational fields. Maxwell's equation is solved in [126] using finite difference approximation. Maxwell's equation solvers are used to solve the transient electromagnetic problems that provide valuable information about the multidimensional conductivity of the subsurface [67]. These methods are also used for oil discoveries. Navier-Stokes equations [52] are also solved by finite difference and is extensively used in computational fluid dynamics. In [107] the authors use a finite difference scheme to perform a room acoustic simulation, implemented on an NVIDIA Tesla K20 GPU. Most of these implementations may benefit from the researches conducted on optimizing the RTM workflow, and similarly these works and many others usually use the same physics or similar solver to those used by the seismic applications and by the RTM workflow particularly. Their conclusions, optimization techniques and performance studies on GPUs are helpful to our study.

# Chapter 5

# Thesis position and contributions

**Contents**



We presented in chapter 4 a selection of the state-of-the-art seismic applications and in chapter 3 an overview of the HPC facilities that are often leveraged to accelerate such applications and enhance their performance. In this chapter, we specify the context of this thesis within the HPC ecosystem applied to seismic and geophysics and present the contributions. Lastly, we give a technical description of the surveyed hardware and the numerical configuration of the seismic material used in this thesis.

## 5.1 Position of the study

Processing a seismic survey of few tens of squared kilometers generates tens of terabytes of temporary seismic data in order to build the seismic traces, and requires days of computer processing on mainstream CPU clusters. Improving the performance of seismic applications, RTM in particular, is synonymous to reducing the production cost and to adding more physics to the algorithm, therefore accelerating the RTM algorithm is a research topic that continuously attracts the HPC community and the industry. Researchers closely follow the HPC trends, survey the latest developments in terms of hardware design and programming models.
During the last decades, solutions for seismic based on hardware accelerators have emerged. One can find state-of-the-art RTM implementations that leverage GPUs, FPGAs, Cell processors and many others. Given the massively parallel nature of the RTM algorithm the GPU based solutions dominate the arena. With GPU platforms, the seismic surveys can be processed in a matter of hours. However, introducing such a “fancy” hardware design into the scientific community doesn’t come without cost.
Indeed, RTM implementations have been adapted to GPU programming models, such as CUDA, which requires important programming and software maintenance efforts.

Besides, GPUs have a limited size dedicated memory with respect to the memory requirements of the RTM and the seismic applications. To the best of our knowledge the largest amount of memory that features a single GPU is 12 $GB$ (NVIDIA Tesla K40) at the time of writing, whereas it is customary that a HPC compute node comprises up to 64 $GB$ of RAM memory which is more than 5× bigger than the maximum GPU memory. Consequently, scientists have to rely on techniques such as the out-of-core algorithms, or on using more and more GPU nodes in order to allow the RTM application to process big domains (tens of Terabytes) which often decreases the efficiency of RTM GPU implementations.
Moreover, albeit energy efficient as one can notice that the first ranked system in the Green 500 list of November 2014 relies on AMD FirePro S9150 GPUs, GPUs often draw about 3× higher power compared to high-end CPUs, when used for extensive computations. We recall that today, a GPU consumes between 230 W and 300 W (maximum power consumption) and needs to be driven by a CPU which itself usually consumes about 100 W TDP (or even more). We can then tell that the computing elements of a system that contains a high-end GPU draws about 400 W, which leads to the conclusion that GPU based systems are power hungry.
Furthermore, GPU implementations of RTM, as much as for the rest of seismic applications, require a specific memory management that involves memory transfers through the PCI Express bus and duplicating the memory arrays used by the RTM algorithm in the GPU memory. It had been shown that the PCI Express memory traffic can be, depending on the number of compute nodes and on the data snapshotting frequency, a bottleneck to the stencil computations and also to the large scale RTM GPU implementations [23, 63, 147]. Although numerous software solutions such as temporal blocking, overlapping the PCI Express memory traffic with computation, out-of-core etc. are proposed to address this issue, they require extensive programming efforts.

Very recently, the APU technology; a combination of a CPU and an integrated GPU proposed by AMD, has replaced the PCI Express bus by a faster interconnect between the CPU and the GPU, allowing the latter to access the system memory. The APU features a unified memory space that can be addressable by the CPU cores and by the integrated GPU cores which allows to support applications with extensive memory requirements, such as the RTM, to exploit the entire system memory available for computations. Besides, the chip is also considered as a low power accelerator since APUs only draw between 60 W TDP and 95 W TDP. On the one hand, this is mainly due to the low frequencies of integrated GPUs ($\sim$ 700 $MHz$ in average) compared to those of discrete GPUs ($\sim$ 1000 $MHz$), and on the other hand to the fact that the CPUs of APUs are chipped with a minimal configuration and with a small power envelope compared to high-end CPUs.
In addition, with its new memory model, the APU can be considered as an attractive hardware solution to the PCI Express bottleneck. However, APUs are almost an order of magnitude less compute powerful and have a lower memory bandwidth than discrete GPUs.

In this thesis we therefore study the feasibility and the interest of AMD APUs in a geophysics exploration context, with a special emphasis on the RTM application, from single precision performance, power efficiency and programming model perspectives.

## 5.2 Contributions

In this section, we briefly describe the contributions of this thesis as they are detailed in the next chapters of this document.

In chapter 6, we evaluate the APU in terms of architecture and memory model by means of a set of memory and applicative benchmarks. We have mentionned that with the APU, AMD has introduced a new memory model within the family of the hardware accelerators that allows the CPU and the integrated GPU to share data. We focus, in section 6.1, on demystifying this memory model and on studying how it evolves through the APU generations. We also study how exactly the memory objects can be exposed simultaneously to the CPU cores and to the GPU cores, especially for the early generations of APU where memory operations are subject to many limitations (limited in size, only a very small partition being shared between CPU and GPU). For that to do, we introduce different data placement strategies, with which we evaluate the memory bandwidth of the different memory locations of the APU: namely the GPU memory, the host-visible device memory and the device-visible host memory. We investigate when and how the Garlic and Onion buses are used to access memory and how do they affect applications performance on APUs.
Given that the APUs are less compute powerful and less memory efficient than discrete GPUs, we aim to ensure that the APU remains a competitive solution for high performance computing. To this purpose, we implement, in section 6.2, two highly optimized applicative benchmarks in OpenCL: a matrix multiply kernel and a 3D finite difference stencil kernel. The matrix multiply kernel is a compute bound application that we mainly use to benchmark the compute capabilities of the APU (as a matter of fact we only use the integrated GPU of the APU). The finite difference kernel is a memory bound kernel whose performance on the APU is mainly driven by the choice of the APU memory locations where data is stored. The choice of the finite difference stencil kernel is also motivated by the fact that it consists a building block of the seismic applications that we survey in the scope of this work. For both the applicative benchmarks, we adapt their implementations to high-end CPUs, to the successive AMD discrete GPUs and AMD APUs, that rely on different micro-architectures with different performance guidelines, and evaluate their performance accordingly. Being more compute powerful than high-end CPUs, we expect the APU to outperform the CPUs for both the matrix multiply and the finite difference stencil kernels. We evaluate the interest of the new memory model of the APU, as we consider in our study the memory transfers overhead incurred by copying the matrices back and forth between the main memory and the GPU memory (in the case of the matrix multiply kernel) and by the data snaphotting (in the case of the finite difference stencil kernel). In the discrete GPU implementations, this overhead is caused by the data traffic that goes through the PCI Express bus, however the overhead might be mitigated or even non-existent (if zero-copy memory objects are used) in the APU implementations. Therefore, in the case of the finite difference stencil OpenCL kernel, we focus on the impact of the data snaphotting on the sustained performance and underline how and under what conditions the APU can eventually help alleviate the PCI Express overhead reported in the discrete GPU implementations of such applications. We conduct a comparative study, based on the single precision performance, between the performance of the two benchmarks on a high-end AMD CPU (*AMD Phenom TM II x6 1055t Processor*), on two AMD GPU families (Cayman and Tahiti), and on three AMD APU generations (Llano, Trinity and Kaveri). We also show the impact of the data placement strategies on the performance of the matrix multiply

and the finite difference stencil kernels.
Besides, given the low power feature of the APU and the importance of the power wall in the HPC community, we also study, in section 6.3, the power efficiency of the two applicative benchmarks on the latest, at the time of writing, APU generation (Kaveri) and on the latest discrete GPU generation (Tahiti), by comparing their power efficiencies against that of the CPU.
Moreover, the theoretical performance of the integrated GPU of an APU represents the main (more than 80%) compute power of the chip. However, the lasting 20%, provided by the CPU cores, can be used for useful computations or tasks. We investigate, in section 6.4, the potential of the hybrid utilization of the APU by proposing a variant of the finite difference OpenCL kernel implementation that uses both the integrated GPU and the CPU for stencil computations. We propose a general hybrid strategy that can be used to spare stencil-based workloads between the CPU cores and the GPU cores within an APU. In addition we present two possible deployments of this strategy. The first one is task parallel oriented, where diverging control flows are routed to CPU cores as they are less expensive on CPUs than on GPUs (the execution of diverging wavefronts or warps are serialized on GPUs). The second is data parallel oriented, where the computational burden is spread over the GPU and the CPU by dividing the domain with respect to the ratio of the theoretical performance of the integrated GPU to that of the CPU.
Furthermore, we try to define which programming model is more suitable, in terms of programming complexity and performance portability, for the APUs and for the discrete GPUs. We particularly focus, in section 6.5 on comparing OpenACC against OpenCL in terms of performance and ease of programmability. For this comparison we use, other than the OpenCL implementation, three OpenACC implementations of the finite difference stencil algorithm, with gradual complexities. The first implementation is a straightforward OpenACC version where we use exclusively OpenACC directives to accelerate the stencil computations workload. In the second one, we use the HMMPPcg extension in conjunction with the OpenACC directives in order to have more options in terms of GPU optimizations. In the final version, not only we use both OpenACC and HMMPPcg, but also we manually modify the initial code to introduce more optimizations that cannot be realized using the OpenACC standard, such as increasing the ILP and using the GPU local memory.

Based on the conclusions of chapter 6, we study, in chapter 7, the performance, power efficiency and the programmability of two seismic applications, namely the seismic modeling and the RTM, on the node level. We present the seismic modeling as a first step of the RTM algorithm and thus we focus only on the main computation, that is the wave equation solver which is based on 3D finite difference stencil computations. Then, we implement the RTM using the linear checkpointing method, described in section 4.2.2.3. This method is the most used in the industrial and commercial seismic solutions. Besides we think that it represents an acceptable trade-off between the computation and storage complexities. As opposed to the seismic modeling, we take into consideration in the study of RTM algorithm the extensive need for I/O operations, such as saving the source wavefield and reading the receiver wavefield, that are used for data snapshotting. Those require memory transfers through the PCI Express bus in the discrete GPU implementation. Therefore, we assess their impact on the RTM performance.
Furthermore, we compare the performance and the power efficiency of the seismic modeling and the RTM applications on a compute node based on a Kaveri APU against

those of a compute node built on top of an *AMD Phenom TM II x6 1055t Processor* and those on a Tahiti discrete GPU based compute node.

Finally, in chapter 8 we extend our study of the seismic modeling and RTM applications to the large scale implementations. We implement the two applications on a high-end CPU cluster, on a discrete GPU cluster and on an APU cluster. The specifications of these clusters are detailed in section 5.3.1. For the CPU cluster, we modify a flat[1] Fortran MPI implementation that is used, in production by the Total Advanced Computing Research Group, in order to meet our needs. Then, we propose an OpenCL+MPI implementation for the APU and discrete GPU clusters. For the APU cluster we consider studying the impact of using the zero-copy memory objects on the performance and on the scaling of the seismic applications. We also highlight the relevance of zero-copy memory objects to the RTM algorithm, since they help address the total memory available on a compute node and expose it to the integrated GPU cores, which reduces the number of nodes needed to process one seismic shot profile.
We enumerate the MPI communications related issues, such as load balancing and MPI synchronization problems, that arise in the context of the seismic applications. We rely on an explicit mechanism, based on OpenMP, that allows overlapping the MPI communications with computations and show its impact on the performance of the seismic modeling on the CPU cluster and more importantly on the APU and GPU clusters, where the CPUs are idle while the GPU cores are handling the computations. Filling the CPUs with useful tasks (MPI communications) concurrently with the GPU computations can enhance the performance of the seismic applications on the hardware accelerators based clusters.
We also introduce the asynchronous I/O operations into the RTM algorithm and illustrate its impact on the performance of the application on the surveyed hardware. In the GPU and APU implementations, we emphasize how the asynchronous I/O can be delegated to the CPU while the GPU is updating the next time iteration, and on how it may improve the performance on the APU and GPU clusters.
Finally, in order to evaluate the performance of the seismic modeling and RTM, we consider the strong scaling and the weak scaling scenarios. We also conduct a comparaison, based on the strong scaling performance numbers, of the three clusters with respect to three metrics: the first metric is performance where we compare the performance numbers on the same number of nodes on the CPU, APU and GPU clusters; the second is power efficiency where we choose the number of nodes that approximately draw the same amount of TDP power on each cluster (we did not have the appropriate tools to accurately measure the power consumption of each cluster), and the third one is the "production efficiency" where we compare the performance multiplied by the total number of shots that a cluster, with a given number of nodes, can process.
Throughout this chapter, we investigate whether the RTM implementation on the APU cluster can be a solution that addresses the memory limitation, the high power consumption and the PCI Express issue that characterizes the discrete GPU implementation of the RTM.

## 5.3 Hardware and seismic material configurations

In this section we present the specification of the surveyed hardware in the scope of this work, as well as the numerical configuration of the seismic materials used in our

[1] MPI is used even for the cores within the same socket rather than OpenMP.

| | |
|---|---|
| Owner | Total S.A |
| Name | *PANGEA* |
| Location | Pau, France |
| Vendor | SGI |
| Number of nodes | 6900 |
| Processors/node | 2×Intel Xeon CPU E5-2670 8C 2.6 GHz 2-way SMT |
| Peak performance | -Per node: 340.78 GFlop/s<br>-Total: 2296.32 TFlop/s |
| Memory/node | 64 GB |
| Interconnect | InfiniBand FDR |
| Local storage/node | 500 GB |
| File system | Lustre 3.0 |
| Software | Linux OS, Intel compiler version 14.0<br>Intel MPI version 4.3, LSF |

TABLE 5.1: The CPU cluster hardware specifications.

| | |
|---|---|
| Owner | Total S.A |
| Name | DIP |
| Location | Houston, USA |
| Vendor | Cray |
| Number of nodes | 80 |
| Processors/node | 1×NVIDIA Tesla K40s GPU 745 MHz PCI Express gen 3<br>1×Intel Xeon CPU E5-2680 v2 10C 2.8 GHz 2-way SMT |
| Peak performance | -Per node: 4291 GFlop/s (GPU) + 224 GFlop/s (CPU)<br>-Total: 352.7 TFlop/s |
| Memory/node | 32 GB + 12 GB GDDR5 |
| Interconnect | InfiniBand DDR |
| Local storage/node | 500 GB |
| File system | Lustre 2.0 |
| Software | Linux OS, Intel compiler version 14.0<br>Cray MPT version 6.3.1, Slurm, OpenCL 1.1 |

TABLE 5.2: The GPU cluster hardware specifications.

benchmarks and performance evaluations.

### 5.3.1 The hardware configuration

In this section we summarize the specifications of the hardware used to conduct all the performance studies in this work. The hardware include a high-end CPU cluster, an APU cluster and a discrete GPU cluster.
The clusters are mainly used to generate the performance results of the multi-node implementations of the seismic applications presented in chapter 8. Note that the specifications of the hardware used in chapters 6 and 7 are introduced in chapter 3. The table 5.1 presents the specifications of the CPU cluster used for the CPU implementations of the seismic applications. Table 5.2 illustrates the specifications of the discrete GPU cluster used for the deployment of the GPU implementations of the seismic applications. Finally, the table 5.3 summarizes a description of the technical characteristics

| | |
|---|---|
| Owner | Total S.A |
| Name | RDHPC |
| Location | Houston, USA |
| Vendor | Pinguin Computing |
| Number of nodes | 16 |
| Processors/node | 1×AMD A10-7850K Radeon R7 4C+8G APU |
| Peak performance | -Per node: 737.28 GFlop/s (GPU) + 118.4 GFlop/s (CPU)<br>-Total: 13.37 TFlop/s |
| Memory/node | 32 GB |
| Interconnect | InfiniBand DDR |
| Local storage/node | Diskless |
| File system | Lustre 2.0 |
| Software | Linux OS, Intel compiler version 14.0<br>Intel MPI version 4.3, Slurm, OpenCL 1.2 |

TABLE 5.3: The APU cluster hardware specifications.

of the APU cluster used in this work to evaluate the AMD APUs.

### 5.3.2 The numerical configurations of the seismic materials

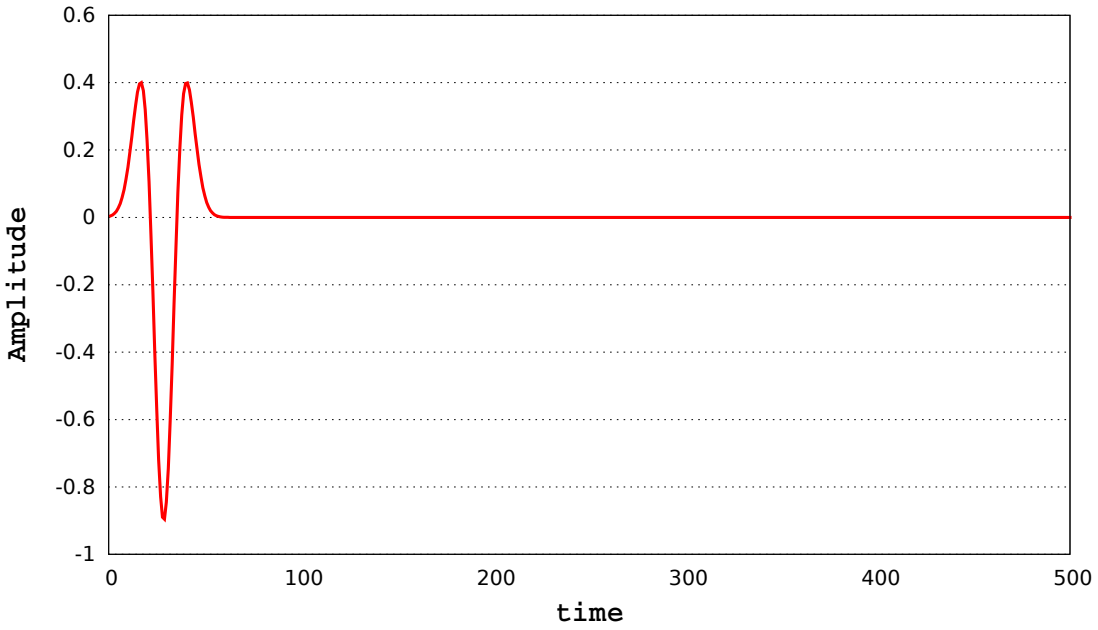
In this section we present the numerical configurations of the seismic materials that are used to drive the simulation of the wave propagation or as input data in the seismic applications workflows.

#### 5.3.2.1 The seismic source

We have mentioned in section 2.3.1 that the wave equation (2.27) requires a second term, i.e. a function $s(t)$ over the time $t$, which simulates a seismic source that periodically injects vibrations on the subsurface. In this study, we chose to add the *Ricker* wavelet to the wave equation (2.27) as a second term. The Ricker wavelet is a variant of the second derivative of the Gaussian function with respect to time. The analytic expression of the amplitude $A$ of the wavelet we use is $A = -(1 - 2\pi^2 f^2 t^2)e^{-\pi^2 f^2 t^2}$, with $f$ the peak frequency and $t$ the time index. In figure 5.1 we depict the evolution of the seismic source function over a 500 time-step interval. The figure is accompanied with a summary of the numerical parameters of the seismic source function.

#### 5.3.2.2 The velocity model and the compute grids

In the scope of this work we use the synthetic velocity model *3D SEG/EAGE salt model*. It is used, throughout this document, to simulate the wave propagation in an isotropic medium with a constant density. Figure 5.2 is a graphic representation of this model which we refer to as $\mathcal{V}$ in the rest of the document for the sake of simplicity. Note that we use the SEP format [200] and that we use the same grid orientation as conventionally used in seismic acquisition geometries [45]. The velocity model is used as the representation of the earth reflectivity in the process of seismic modeling and of seismic migration as input data for the wave equation solver that we are considering in



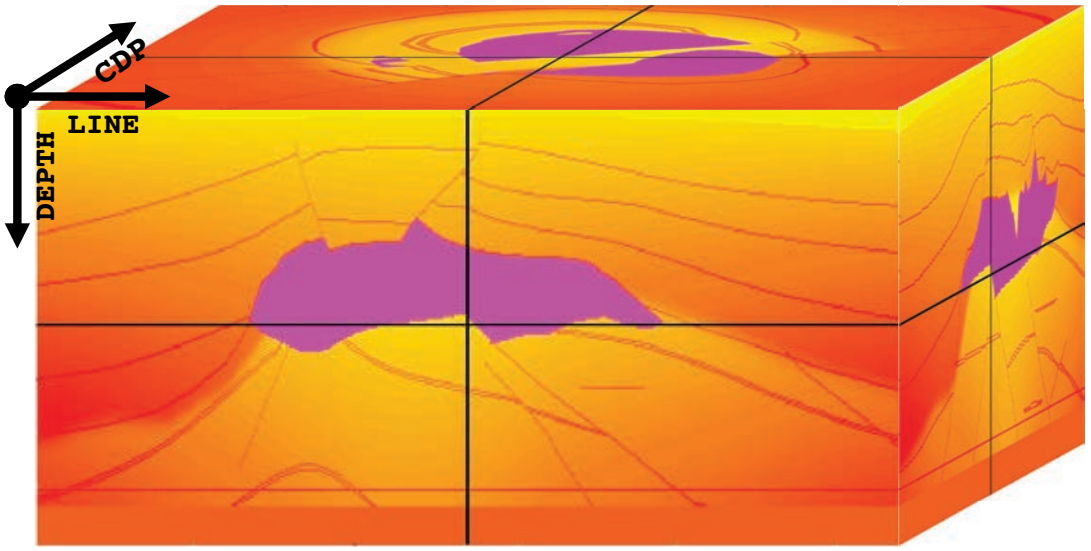

| *parameter* | *value* | *description* |
|---|---|---|
| type | *Ricker* | the name of the source function used in this work |
| frequency | 20 Hz | the frequency of the seismic perturbation |
| amplitude | 1 | the maximum amplitude of the source function |

FIGURE 5.1: The configuration and the evolution with respect to time of the **Ricker wavelet** used as a seismic source in this study. Only a 500 time-step interval is presented.

| *reference* | *dx* | *dy* | *dz* | *#grid points* | *size* | *#nodes* |
|---|---|---|---|---|---|---|
| $\Omega_{8\times8\times8}$ | 8 m | 8 m | 8 m | $1112 \times 3443 \times 534$ | 7870 MB | CPU cluster: 1-64 |
| | | | | | | GPU cluster: 8,16 |
| | | | | | | APU cluster: 8,16 |
| | | | | | | APU cluster (with zer-copy): 1-16 |

TABLE 5.4: Numerical parameters of the **compute grid**, used in the **seismic modeling** application, with respect to the **strong** scaling scenario.

this study. This model was built by the SEG research committee, and created as part of the Advanced Computational Technology Initiative. The figure is accompanied with a table that summarizes the pertinent SEP parameters that describe the geometry of $\mathcal{V}$. $\mathcal{V}$ is approximately 8.5 $km$ wide by 27 $km$ long by 4 $km$ deep. The discretization parameters, the number of grid points and the memory size of the compute grids used for the seismic modeling application in the multi-node strong scaling scenario are detailed in table 5.4. To help read this table, and the others following, we summarize the used notations and their signification in table 5.6. Those of the RTM in the strong scaling scenario are presented in table 5.5. Note that the number of grid points, in each direction, of a given compute grid is determined after dividing the length of the velocity model (in meters) in the corresponding dimension by the corresponding spatial discretization step (in meters) along the same direction. Moreover, we recall that the smaller are the spatial discretization steps the bigger is the compute grid and, due to the CFL




| *parameter* | *value* | *description* |
|---|---|---|
| n1 | 676 | number of grid points along the first dimension (CDP) |
| n2 | 676 | number of grid points along the second dimension (LINE) |
| n3 | 201 | number of grid points along the third dimension (DEPTH) |
| d1 | 12.5 m | length of the grid step along the first dimension (CDP) |
| d2 | 40.0 m | length of the grid step along the second dimension (LINE) |
| d3 | 20.0 m | length of the grid step along the third dimension (DEPTH) |

FIGURE 5.2: $\mathcal{V}$, the **velocity grid** of the problem under study: *3D SEG/EAGE salt model*. LINE is the *inline* axis where the streamers are numbered (in meters). CDP is the *cross-line* axis where the receivers in each streamer are aligned (in meters). DEPTH designs the depth of signals (in meters).

| *reference* | *dx* | *dy* | *dz* | *#grid points* | *size* | *#nodes* |
|---|---|---|---|---|---|---|
| $\Omega_{9\times9\times9}$ | 9 m | 9 m | 9 m | $997 \times 3060 \times 474$ | 5517 MB | CPU cluster: 1-64 |
| | | | | | | GPU cluster: 8,16 |
| | | | | | | APU cluster: 8,16 |
| | | | | | | APU cluster (with zero-copy): 1-16 |

TABLE 5.5: Numerical parameters of the **compute grid**, used in the seismic **migration application (RTM)**, with respect to the **strong** scaling scenario.

| | |
|---|---|
| *dx* | spatial discretization step along the $X$ direction. |
| *dy* | spatial discretization step along the $Y$ direction. |
| *dz* | spatial discretization step along the $Z$ direction. |

TABLE 5.6: Summary of the notations used to define the compute grids.

| *reference* | *dx* | *dy* | *dz* | *#grid points* | *size* | *#nodes* |
|---|---|---|---|---|---|---|
| $\Omega_{16\times16\times16}$ | 16 m | 16 m | 16 m | $556 \times 1721 \times 267$ | 983 MB | CPU cluster: 1 |
| | | | | | | APU/GPU clusters: 1 |
| $\Omega_{16\times16\times8}$ | 16 m | 16 m | 8 m | $556 \times 1721 \times 534$ | 1967 MB | CPU cluster: 2 |
| | | | | | | APU/GPU clusters: 2 |
| $\Omega_{16\times8\times8}$ | 16 m | 8 m | 8 m | $556 \times 3443 \times 534$ | 3935 MB | CPU cluster: 4 |
| | | | | | | APU/GPU clusters: 4 |
| $\Omega_{8\times8\times8}$ | 8 m | 8 m | 8 m | $1112 \times 3443 \times 534$ | 7870 MB | CPU cluster: 8 |
| | | | | | | APU/GPU clusters: 8 |
| $\Omega_{8\times8\times4}$ | 8 m | 8 m | 4 m | $1112 \times 3443 \times 1068$ | 15740 MB | CPU cluster: 16 |
| | | | | | | APU/GPU clusters: 16 |
| $\Omega_{8\times4\times4}$ | 8 m | 4 m | 4 m | $1112 \times 6886 \times 1068$ | 31480 MB | CPU cluster: 32 |
| | | | | | | APU/GPU clusters: $\emptyset$ |
| $\Omega_{4\times4\times4}$ | 4 m | 4 m | 4 m | $2224 \times 6886 \times 1068$ | 62660 MB | CPU cluster: 64 |
| | | | | | | APU/GPU clusters: $\emptyset$ |

TABLE 5.7: Numerical parameters of the **compute grids**, used in the **seismic modeling** and in the **migration application (RTM)**, with respect to the **weak** scaling scenario. The number of nodes in the APU and GPU cluster goes only up to 16 which is the maximum capacity of the APU cluster.

condition 2.3.3.2, the longer is the simulation. In our study, we manually set the total number of numerical iterations to 1000 in all the test cases and the frequency of the data snapshotting to 10. Besides, the memory footprint of each compute grid (the *size* column in the tables) corresponds to the size of only one instance of the compute grid, while in the implementations of the seismic applications multiple instances of each compute grid are used.

For the weak scaling scenario the compute grids numerical specifications are depicted in table 5.7. Note that for the APU and GPU clusters we limit the number of used nodes to 16 which is the maximum capacity of the APU cluster (RDHPC) that we have in hands (see table 5.3). For the sake of clarity and to help reference the compute grids in the rest of the document we give a name to each numerical configuration of a compute grid. For example $\Omega_{16\times8\times8}$ refers to the compute grid that is built with $dx = 16$, $dy = 8$ and $dz = 8$. When comparing CPU, APU and GPU on a single node in chapter 7, we make use of $\Omega_{16\times16\times16}$ for seismic modeling and for seismic migration, given that it represents the biggest compute grid that can fit on a CPU node, on a GPU node as well as on an APU node. Whereas when comparing the efficiency of multi-node implementations in chapter 8, we use $\Omega_{8\times8\times8}$ for the seismic modeling in the strong scaling scenario, and $\Omega_{9\times9\times9}$ for the RTM in the strong scaling as well. The compute grid $\Omega_{8\times8\times8}$ (resp. $\Omega_{9\times9\times9}$) corresponds to the biggest configuration that a CPU node can hold in the case of the seismic modeling application (resp. in the case of the RTM).

# Part II

# Seismic applications on novel hybrid architectures

# Chapter 6

# Evaluation of the Accelerated Processing Unit (APU)

**Contents**



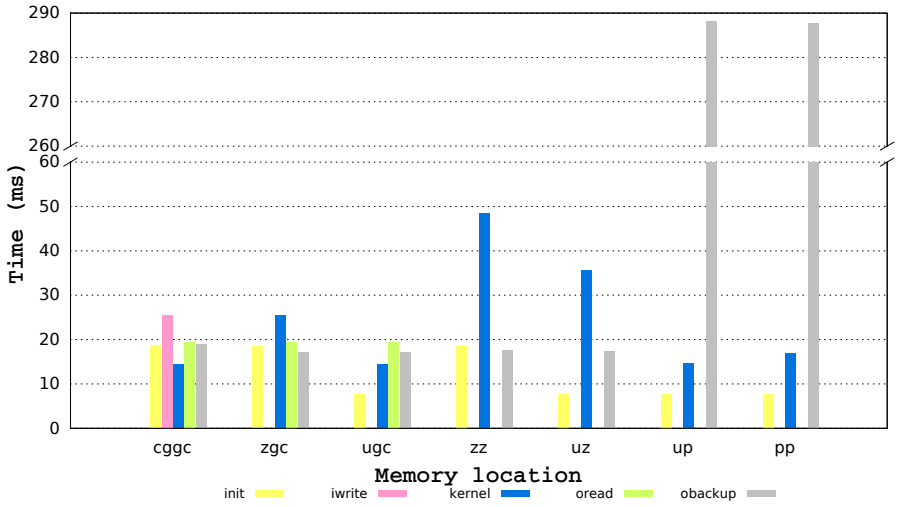
In this chapter we evaluate the APU technology by means of memory, applicative and power consumption benchmarks. We also evaluate the CPU-GPU hybrid utilization of the APU and leverage the OpenACC directive-based programming model to implement the applicative benchmarks and evaluate their performance on APUs and discrete GPUs. Albeit the newness of the AMD APU architecture, some related works to this chapter can be found in [71], were an empirical study is conducted in order to characterize the efficacy of the AMD APU by means of a set of benchmarks (FFT, Molecular Dynamics, Scan and Reduction). The authors inferred that the APU have reduced the overhead incurred by the PCI Express data transfers for discrete GPUs by as much as six-fold. They also showed that the APU can be attractive for memory bound benchmarks, namely Scan and Reduction, where it delivers 3× higher performance than that of discrete GPUs.

First, we particularly focus, in section 6.1, on the impact of the different APU memory partitions on applications performance by presenting the different *data placement strategies*. Then, we evaluate in section 6.2 the performance of a compute bound application (matrix multiply) and a memory bound one (3D stencil) on APUs by conducting a performance comparison against AMD discrete GPUs and an AMD CPU. This section summarizes the results that were published in: *"H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte and I. Said. Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil. 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2013"* where the work on the first two APU generations, i.e Llano and Trinity, was published. After the publication, the work was extended to the third APU generation which is Kaveri. Besides, we briefly emphasize in section 6.3 the low power consumption of APUs by measuring the power efficiency of the applicative benchmarks on CPU, on discrete GPUs and on APUs. Furthermore, the section 6.4 presents the potential of hybrid CPU-GPU utilization. This section was subject to the following publication: *"P. Eberhart, I. Said, P. Fortin, H. Calandra. Hybrid strategy for stencil computations on the APU. 1st International Workshop on High-Performance Stencil Computations, 2014"*. Finally, we evaluate in section 6.5 the performance of APUs and of discrete GPUs when using a directive-based programming model, i.e. OpenACC coupled with HMPPcg in the scope of this work, and compare it against the corresponding OpenCL performances.

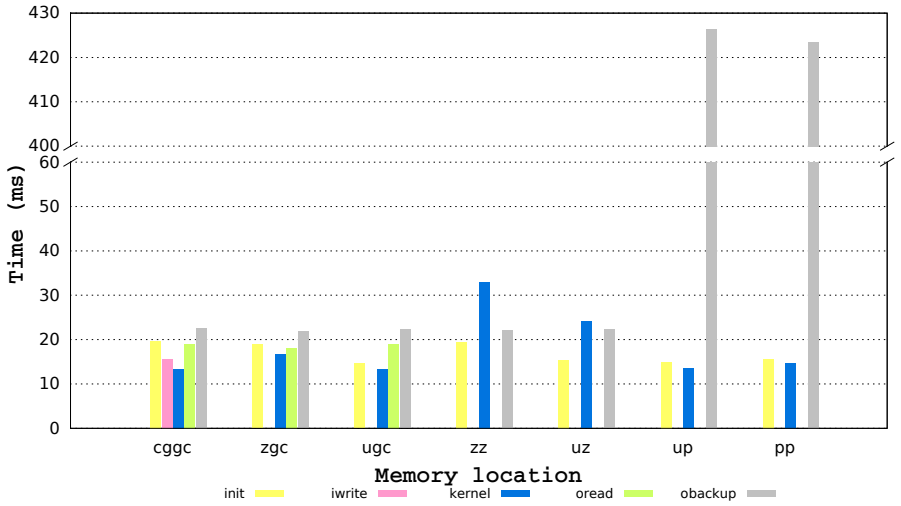
## 6.1 Data placement strategies

We have mentionned in section 3.1.3 that the APU memory system is different from that of a discrete GPU. On the one hand, the CPU (host) and the GPU (device) are fused in the same socket and the PCI Express interconnection between them is removed, but on another hand, the memory address space is not unified as the integrated GPU has its own dedicated memory partition which is a sub-partition of the system memory, as shown in figure 3.8. Not only data can be explicitly copied from the system memory to the integrated GPU partition and vice versa but also zero-copy memory objects can be shared between the CPU and the integrated GPU [29] by using the "host-visible device memory" or the "device-visible host memory" memory partitions (see figure 3.8). In addition, GPU read-only zero-copy memory objects are stored in the USWC (Uncacheable, Speculative Write Combine) memory, an uncached memory (part of the device-visible host memory) in which data is not stored into the CPU caches. This memory partition is subject to CPU contiguous write operations using the *write combine* buffers (WC in figure 3.8) to increase memory throughput. Besides, subsequent GPU reads from this memory are fast as the Garlic bus is used. We refer to the different memory locations of the APU as a lower case letter: `c` refers to the regular cacheable CPU memory (always pinned here for efficient CPU-GPU data transfer), `z` to the device-visible host memory employed as cacheable zero-copy memory objects, `u` to the device-visible host memory employed as uncached zero-copy memory objects (USWC), `g` to the regular GPU memory and `p` to the GPU persistent memory. We summarize these notations in table 6.1.

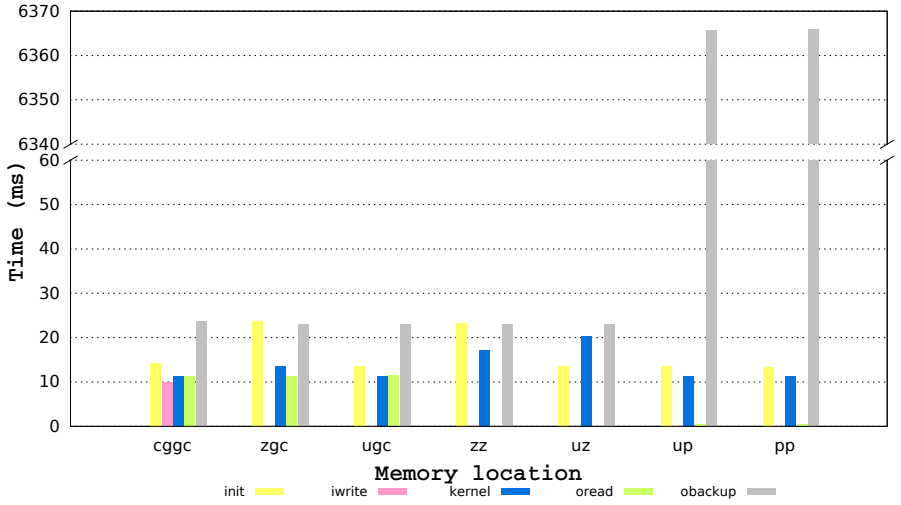
In order to test the performance of the read and write accesses to these buffers, we developed an OpenCL benchmark, a data placement benchmark, that makes the integrated GPU copy data from an input buffer stored in one APU memory location to an output buffer that lies in another APU memory location. For both the input and output




(a) Llano.




(b) Trinity.




(c) Kaveri.

FIGURE 6.1: The **data placement** benchmark times. The size of the used buffers is **128 MB**. The tests are performed on three APU generations: **Llano**, **Trinity** and **Kaveri**.

| | |
|---|---|
| c | Cacheable CPU memory |
| z | Device-visible cacheable memory |
| u | Device-visible uncached memory |
| g | GPU memory |
| p | GPU persistent memory |

TABLE 6.1: A summary of the APU memory location symbols.

buffers, cg (respectively gc) denotes an explicit data copy from the CPU partition to the GPU partition g (resp. from the GPU partition g to the CPU partition c), whereas z, u and p refer to the corresponding zero-copy buffer. For example, zgc describes the following data placement strategy: the input buffer is in z memory location, the output buffer is first created in the GPU memory (g) and then, is explicitly copied to the CPU memory (c). We tried different data access strategies: cggc, zgc, ugc, zz, uz, up and pp. We show the results in figure 6.1 for the three APU generations surveyed in this work: Llano, Trinity and Kaveri. We recall that the technical specification of the surveyed hardware is summarized in table 3.3.
In the figure, *init* is the input buffer initialization (using a CPU parallel memcpy operation) time, *iwrite* is the input buffer transfer time (if needed) to the GPU memory, *kernel* is the execution time of the OpenCL kernel, *oread* is the output buffer transfer time (if needed) back to the CPU and *obackup* is the time of an extra copy from the output buffer to a temporary buffer in the CPU memory in order to measure the time of reading from the memory location in which the output buffer is saved. In some cases, a *map* operation (resp. an *unmap* operation) is required before *iwrite* or *oread* (resp. after *iwrite* or *oread*). These operations have negligible times compared to the other operations and therefore are not presented in our results. We use system wallclock for timing this benchmark, as well as for the rest of the tests presented in this section. All the tests are run multiple times (up to 40) after devices "warm up". Note that the size of each buffer used in this benchmark is 128 $MB$. Besides, the tests on Llano and Trinity were conducted on a Windows operation system, with an AMD Catalyst OpenCL driver version 11.4, since the zero-copy memory buffers were supported in Windows only at the beginning of the project. On the contrary, the tests on Kaveri were ran on a Linux operating system with an AMD Catalyst OpenCL driver version 15.4.

We note that GPU reads from USWC are as fast as GPU reads from GPU memory, CPU writes to GPU persistent memory are fast but reads are very slow (see *obackup* in figure 6.1). Note that the *obackup* time depends mostly on the operating system and on the OpenCL driver which explains the different times measured on Llano, Trinity and Kaveri. This difference is of a minor interest for us as our main purpose is to highlight that CPU reads from the GPU persistent memory are very slow: they indeed bypass CPU caches. The figure also shows that contiguous CPU writes to USWC (u) offer the highest bandwidth for *init*. We particularly focus on the OpenCL kernel times which are represented by the blue bars in figure 6.1. In the one hand, we note that the kernel time varies with respect to the data placement strategies, namely the GPU memory accesses to z (via Onion with a sustained bandwidth between 6 $GB/s$ and 16 $GB/s$) are slower than accesses to u (via Garlic with a sustained bandwidth that ranges between 18 $GB/s$ and 23 $GB/s$) and g, except for the Kaveri APU where memory accesses to z are slightly faster than those to u. In the other hand, it also varies with respect to the APU generations: the kernel time when using the zero-copy memory buffers (zz) decreases from an APU generation to the next. With Llano the zz kernel time is 239% slower

than that of the explicit copy (`cggc`). With Trinity the ratio is reduced to 149% and with Kaveri it is further reduced down to 51%. This means that the bandwidth of the Onion bus is getting enhanced and is approaching that of the Garlic bus: within the first APU generation (Llano) the bandwidth of Onion is roughly 50% of Garlic bandwidth, while that of the most recent one at the time of writing (Kaveri) is 60% of the Garlic bandwidth.

To conclude, the data placement is to be considered as a relevant factor when it comes to applications performance on APU, especially for memory bounds ones. Relying on zero-copy memory objects implies the usage of cache coherent memory buses which substantially reduces the sustained bandwidth especially for the APU early generations but slightly enhanced with the Kaveri APUs where the bandwidth of zero-copy memory objects roughly represents 60% of the maximum bandwidth. However, zero-copy memory objects have the advantage to allocate a major part of the main memory that is addressable by the integrated GPU. Ultimately in this work we will inspect the impact of data placement strategies on the seismic applications performance, and in a more general note in the rest of the chapter we select the most relevant placement strategies: *cggc*, *ugc*, *uz* and *up* to evaluate their impact on a selection of applicative benchmarks (see section 6.2).

## 6.2 Applicative benchmarks

In this section we present two applicative benchmarks that we use to evaluate the performance of three APU generations, namely Llano, Trinity and Kaveri, two discrete GPU generations, Cayman and Tahiti, and one AMD CPU which is the *AMD Phenom TM II x6 1055t Processor*. We chose to use a single precision matrix multiplication OpenCL kernel and a single precision 3D finite difference stencil OpenCL kernel. We put a special focus on the possible memory transfers between the CPU and the GPU that are required by the two benchmarks. Those memory transfers have immediate impact on the APU and discrete GPUs performances, and can be used to highlight the main differences between the memory model of the two architectures. Therefore we include the CPU-GPU transfer times in the benchmark results in this section. For each benchmark, we provide a quick description of the implementation choices and optimizations. Our major optimizations include use of vector instructions, tiling in the local memory, software pipelining, register blocking and auto-tuning. The OpenCL kernels are highly tuned in order to provide a fair performance comparison between all tested devices. We also show some performance numbers on each tested device and we emphasize the impact of data placement strategies on APUs performance. Finally, we compare the performance of the integrated GPUs against those of the CPU and the discrete GPUs.

### 6.2.1 Matrix multiplication

This section shows the implementation details and performance numbers of a matrix multiplication OpenCL kernel on the surveyed hardware: an *AMD Phenom TM II x6 1055t Processor* CPU (see table 3.1 for the technical specifications), two discrete GPUs (Cayman and Tahiti whose technical specifications can be found in table 3.2) and three APUs (Llano, Trinity and Kaveri presented in table 3.3).

#### 6.2.1.1 Implementation details

We consider to compute single precision matrix multiplications as $C = C + A \times B$ where $A$, $B$, and $C$ are all square $N$x$N$ matrices (see [218] and [163]). This corresponds to the BLAS SGEMM routine. The floating point computation and data storage complexities of the matrix-matrix multiplication algorithm are respectively $O(N^3)$ and $O(N^2)$. This algorithm is known to be compute bound and its compute intensity is $O(N)$[1]. First, we implement a straightforward version in a data parallel fashion where each work-item computes $X$ ($X = 2$ or $X = 4$) elements of $C$. $X$ refers to the ILP (Instruction Level parallelism) [216] which may be a relevant performance factor in some implementations. This implementation is an OpenCL baseline for all architectures and will be referred to as *scalar*. Second, we explicitly vectorized the code using the OpenCL *float4* data structure in order to take advantage of the VLIW4 or VLIW5 based architectures (only the Cayman discrete GPU, and the Llano and Trinity APUs). We call this implementation *vectorized*. Note that in both *scalar* and *vectorized* the $C$ matrix is naturally partitioned due to the OpenCL execution model that gathers work-items in workgroups. Next, we apply a blocking algorithm that splits $A$, $B$ besides of $C$ matrices into square sub-matrices, which often has higher performance on systems with a memory hierarchy, on the *vectorized* implementation. Blocking matrix-matrix multiplications exploit more data reuse and achieve higher effective memory bandwidth. The matrices $A$ and $B$ are partitioned into blocks stored in the local memory and these blocks are laid out as OpenCL workgroups. This version is referred to as *local vectorized*. Besides, we implement another version, called *image*, using OpenCL image 2D memory objects. These are arranged with a cache friendly tiled layout format and can provide additional memory bandwidth in certain cases by using the texture caches on GPUs, coupled with texture sampling hardware for reading from these caches [46]. Finally, we chose to consider an additional implementation, labeled *local scalar*, for the GCN based hardware (the Tahiti discrete GPU and the Kaveri APU). In this implementation, we make use of the OpenCL local memory but without explicitly vectorizing the code as it is the case in the *local vectorized* version. We expect that the *local scalar* take advantage from the scalar design of the GCN micro-architecture based GPUs, and can highlight the architectural differences between the TeraScale graphics micro-architecture based GPUs and those designed on top of GCN.
For each implementation we use auto-tuning to find the best values for the following parameters: workgroup sizes, ILP factor, instruction scheduling, and local memory sizes (for *local vectorized* only). Note that each test is ran over a total of 100 times (or iterations) after issuing devices "warmup". Each performance result is the average of the total number of iterations. This benchmark protocol also applies to the rest of the tests presented in this chapter.

#### 6.2.1.2 Devices performance

We first consider timing the kernel execution only. We try square matrices with dimensions $N$x$N$ ($N$ varies between 64 and 4096).

Figure 6.2 summarizes the performance of the different OpenCL implementations on the CPU. We compare them against an OpenMP C code, compiled with *gcc version*

[1] On some GPUs however, this kernel can be memory bound because of their huge compute power (with respect to their internal memory bandwidth).

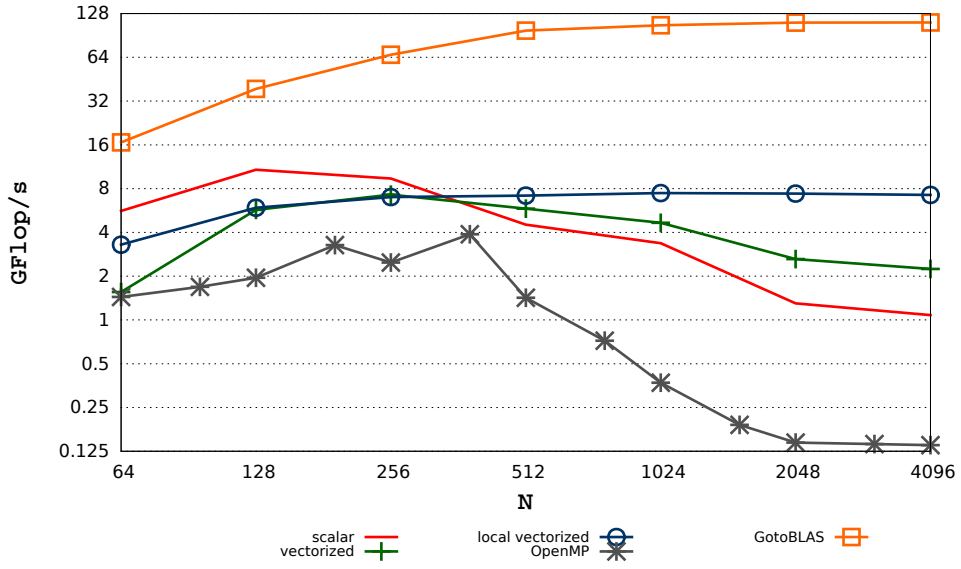



FIGURE 6.2: The performance numbers of the **matrix multiplication** OpenCL kernel, along with OpenMP and *GotoBLAS* implementations, on the ***AMD Phenom TM II x6 1055t Processor*** as a function of $N$, the dimension of the used **square matrices**.

*4.6.1* and a highly efficient *GotoBLAS* [2] based SGEMM implementation. The performance numbers (GFlop/s), of the CPU as well as those of the rest of architectures, are calculated based on the theoretical number of floating point operations issued by the applicative benchmark (matrix multiplication in this case and an $8^{th}$ order stencil in section 6.2.2).
We note that OpenCL is faster than OpenMP mainly because of the natural OpenCL blocking on the $C$ matrix. The *local vectorized* implementation is more efficient than the *vectorized* for large matrices thanks to the blocking on matrices $A$ and $B$. But the *GotoBLAS* clearly delivers the highest performance.

For our GPU implementations, we made use of the AMD Catalyst driver version 13.4, except for the Tahiti GPU for which we provide the performance numbers based on both the drivers 13.4 and 15.4, and for the integrated GPU of Kaveri where we upgraded the driver to version 15.4 (the most recent driver at the time of writing). We also used the AMD OpenCL Math Library (*clAmdBlas*) SGEMM implementation (version 1.6) to compare our implementations against it. Discrete GPU results are shown in figure 6.3 and figure 6.4. The first is about the Cayman performance, where the *scalar* version performs poorly on this vector architecture. The *vectorized* version is more efficient than the *local vectorized*, which is unexpected, but its performance is comparable to the clAmdBlas performance. Thanks to the texture memory, the *image* implementation is the fastest. Figure 6.4 is about Tahiti performance. We note that the *local vectorized* version gives better results (up to 1.740 TFlop/s) than the *vectorized* version and also than *clAmdBlas*. The OpenCL images and the texture caches do not provide a huge performance enhancement (the *image* version delivers 1.950 TFlop/s), such that in Cayman, as both reads and writes are systematically cached on Tahiti. After upgrading the driver to version 15.4 and adding the *local scalar* version, the performance numbers of the matrix multiplication OpenCL kernel on Tahiti are updated and presented in figure 6.5. We notice a 15% to 25% improvement of the performance

[2] http://www.tacc.utexas.edu/tacc-projects/gotoblas2

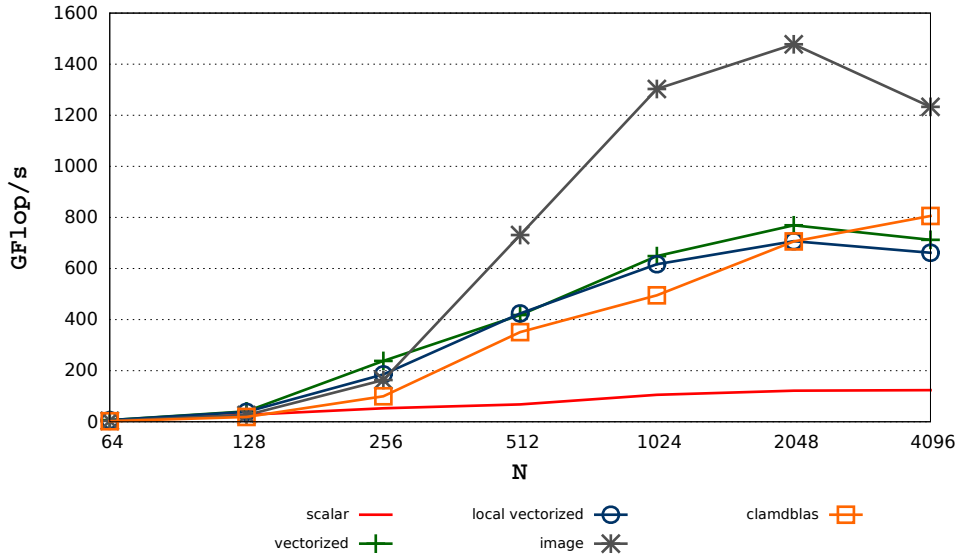



FIGURE 6.3: The performance numbers of the **matrix multiplication** OpenCL kernel on the **Cayman** discrete GPU as a function of $N$, the dimension of the used **square matrices**.

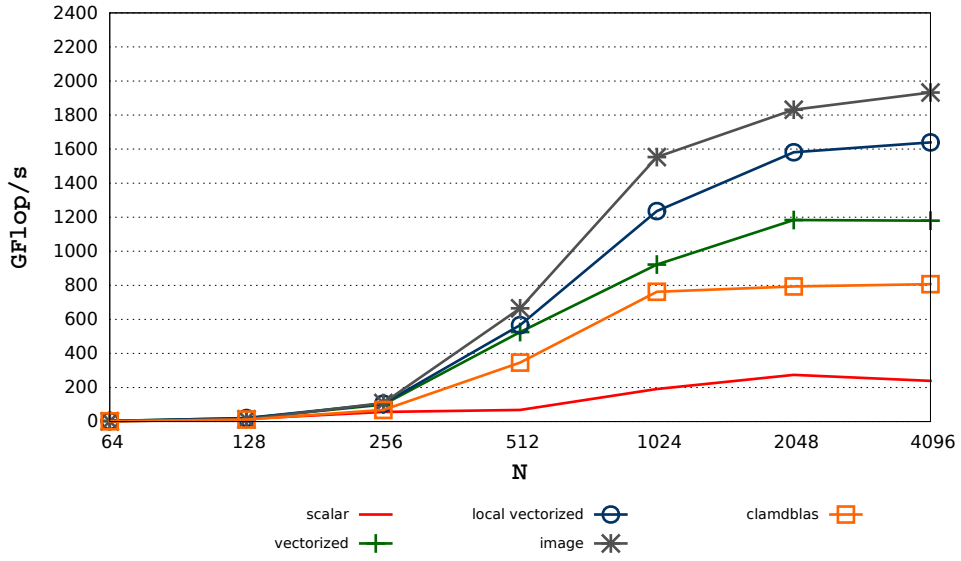



FIGURE 6.4: The performance numbers of the **matrix multiplication** OpenCL kernel on the **Tahiti** discrete GPU as a function of $N$, the dimension of the used **square matrices**.

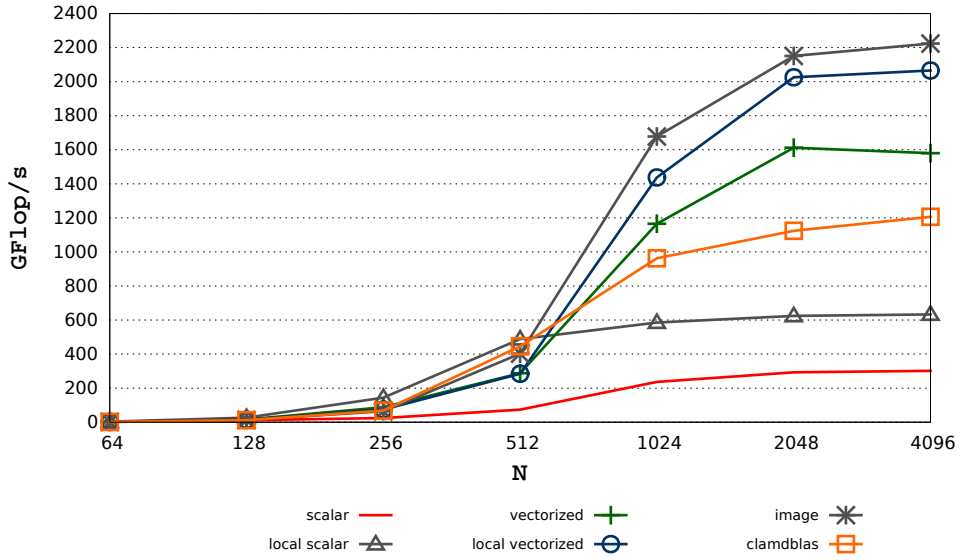



FIGURE 6.5: The updated performance numbers (after upgrading the **OpenCL driver to version 15.4**) of the **matrix multiplication** OpenCL kernel on the **Tahiti** GPU as a function of $N$, the dimension of the used **square matrices**.

of the *image*, *vectorized* and *local vectorized* versions. As a matter of fact, the *image* still delivers the best performance (2.2 TFlop/s) followed by the *local vectorized* version with 2.05 TFlop/s. The *scalar* version is subject to a higher performance enhancement percentage (50%) and the *local scalar* version delivers 2× higher performance than that of the *scalar* one. Even after upgrading the driver, the vector versions of the matrix multiplication OpenCL kernel still perform very well on the scalar design and offers better performance numbers than the scalar ones (by a factor higher than 3), while we are expecting the latter to perform better on a GCN based GPU. We were unable to explain these results. It is probably due to the fact that AMD OpenCL driver versions, that we are using, don't take enough advantage from the new hardware design of the early instances of GCN based GPUs.

Finally, figures 6.6, 6.7 and 6.8 illustrate the performance results of the integrated GPUs of Llano, Trinity and Kaveri respectively. The *scalar* version delivers a low performance on the three APU generations. While this result is expected for Llano given that its integrated GPU is based on a vectorized micro-architecture and thus the code needs to be explicitly vectorized, it is less expected to obtain such poor performance on Kaveri whose integrated GPU is based on the GCN micro-architecture. Similarly to the Cayman GPU, the performance of the *vectorized* version is comparable to the *clAmdBlas* performance. For both Llano and Trinity the *vectorized* version is more efficient than the *local vectorized* implementation. On the contrary, the *local vectorized* implementation performs up to 29% better than the *vectorized* version on the Kaveri APU. Furthermore, we have mentionned that we also consider the *local scalar* version for the Kaveri APU in order to benifit from, in addition to the scalar design of the Sea Islands GPUs, the local memory. As a matter of fact, the *local scalar* implementation has delivered even poorer performance than that delivered by the *scalar* version on both Llano and Trinity APUs, thus we chose not to add it to the plots in figures 6.6 and 6.7 for the sake of clarity. In the case of the Kaveri APU, the behavior of the *local scalar* is different as it gives up to 3× higher performance numbers than the *scalar* version as it can be seen in the figure 6.8 (used with the driver version 15.4). This may give some indications that the scalar

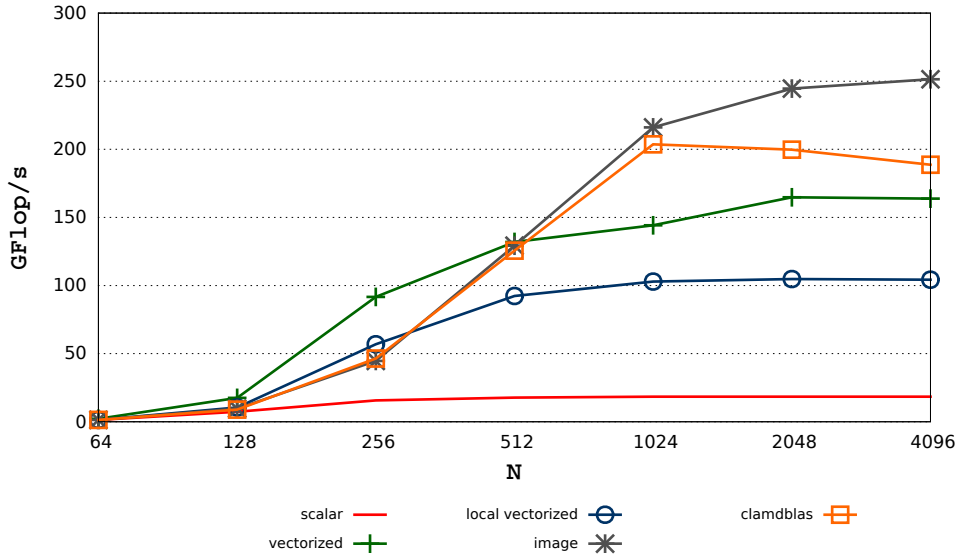



FIGURE 6.6: The performance numbers of the **matrix multiplication** OpenCL kernel on the **Llano** integrated GPU as a function of $N$, the dimension of the used **square matrices**.

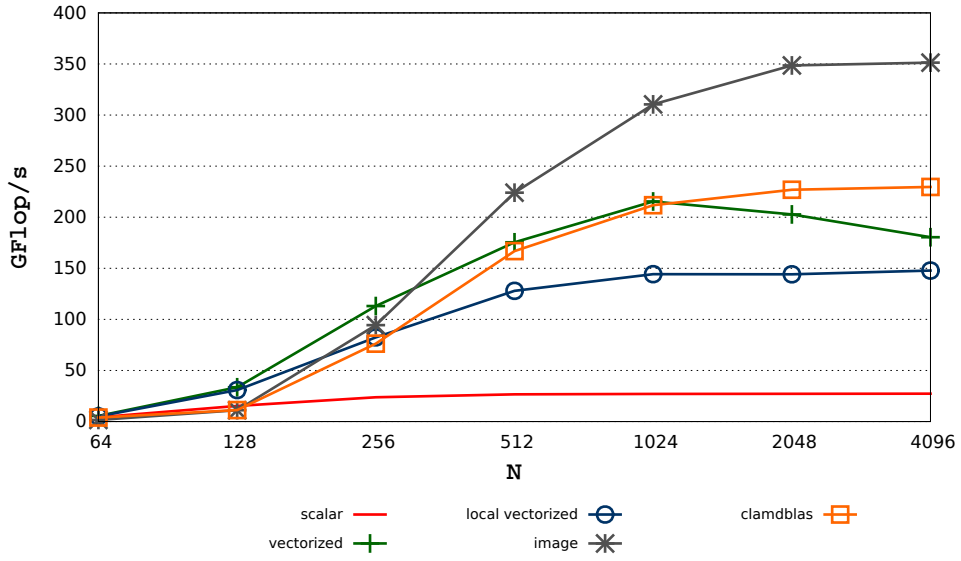



FIGURE 6.7: The performance numbers of the **matrix multiplication** OpenCL kernel on the **Trinity** integrated GPU as a function of $N$, the dimension of the used **square matrices**.

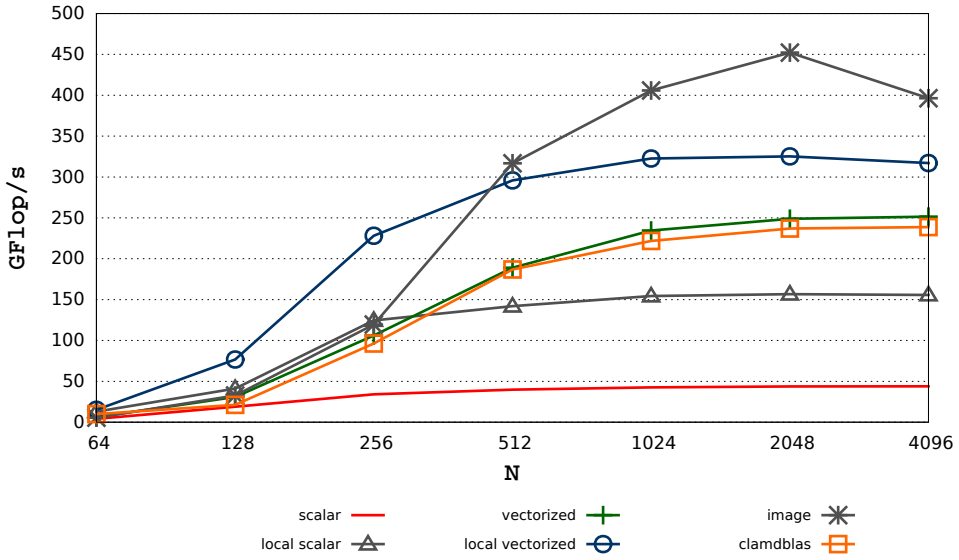



FIGURE 6.8: The performance numbers of the **matrix multiplication** OpenCL kernel on the **Kaveri** integrated GPU as a function of $N$, the dimension of the used **square matrices** (the **OpenCL driver version 15.4** is used).

memory access to the local memory in the Sea Islands GPU family (as well as in all the GCN based GPUs) are better handled than in the previous generations which explains the different behavior of the Kaveri APU compared to both Llano and Trinity.
We also notice that the use of OpenCL images enhances substantially the performance on Llano and Trinity, and relatively the performance on Kaveri. However, copying OpenCL images back and forth between CPU and GPU incurs a very large overhead which lowers the overall performance with communication times considered. For this reason, we do not use the *image* implementations in the rest of this section.

#### 6.2.1.3 Impact of data placement strategies on performance

In order to measure the impact of the data placement strategies on the matrix multiplication APU performance we try the four scenarios selected in Section 6.1: *cggc*, *ugc*, *uz* and *up* with both *vectorized* and *local vectorized* implementations. $A$ and $B$ correspond to the input buffers, whereas $C$ is an input/output buffer. We exclude the *image* from this survey because OpenCL image memory objects loose their intrinsic properties when saved in some memory locations within the APU such as USWC. Note that the possible communication times are included in the following performance numbers. Figures 6.9, 6.10 and 6.11 show the performance numbers of the two implementations coupled with the four data placement strategies on Llano, Trinity and Kaveri respectively. We conclude that in order to obtain the best $SGEMM$ performance on Llano and Trinity integrated GPUs, we have to use the *vectorized* implementation coupled with either the `cggc` or `uz` data placement strategy depending on the matrices sizes. When it comes to the performance on the Kaveri APU, the figure 6.11 indicates that the *local vectorized* implementation gives the best performance when the `cggc` strategy is applied. However, we noticed that the performance of the OpenCL kernel (computation only) of the benchmark, when the zero-copy memory objects are used, is comparable to that when explicit copies are employed (`cggc`). As a matter of fact the performance of `local`

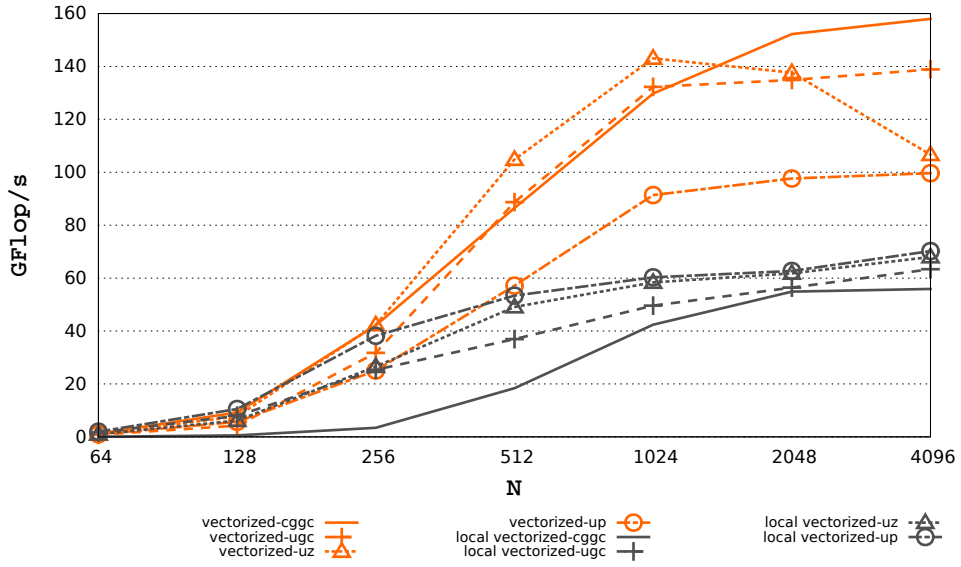



FIGURE 6.9: The impact of **data placement strategies** on the performance of the **matrix multiplication** kernel on the **Llano** APU.

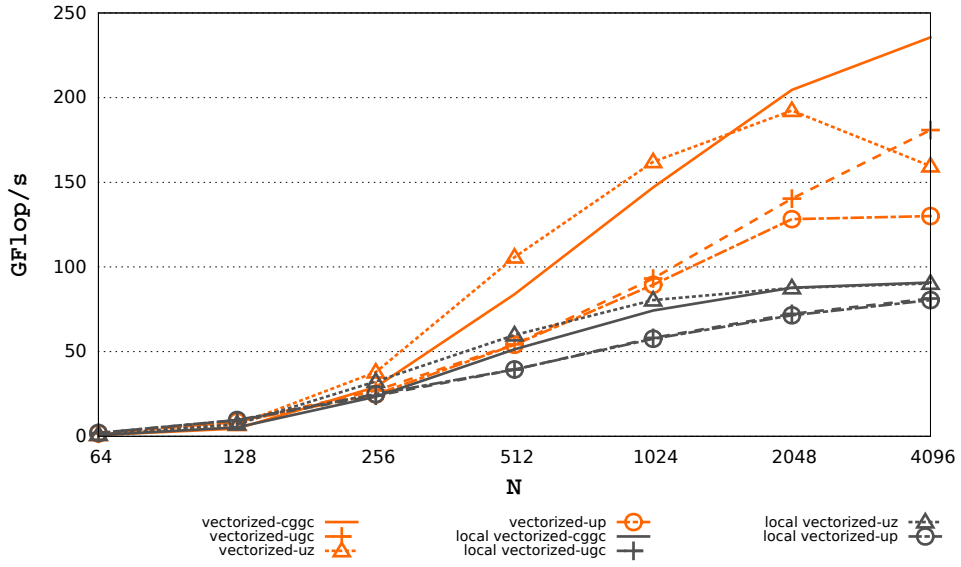



FIGURE 6.10: The impact of **data placement strategies** on the performance of the **matrix multiplication** kernel on the **Trinity** APU.

`vectorized-zz` is 3% lower than that of `local vectorized-cggc` on Kaveri, whereas on Llano the `vectorized-uz` performance is 36% lower than that of `vectorized-cggc`.

#### 6.2.1.4 Performance comparison

The purpose of running the matrix multiplication OpenCL kernel on discrete GPUs and on an AMD CPU is to evaluate the APU performance by conducting a comparaison between CPU, APUs and GPUs. Figure 6.12 illustrates a performance comparison of all the tested devices with the best implementation on each of a complete matrix multiplication (with possible communication times). *comp-only* refers to the performance

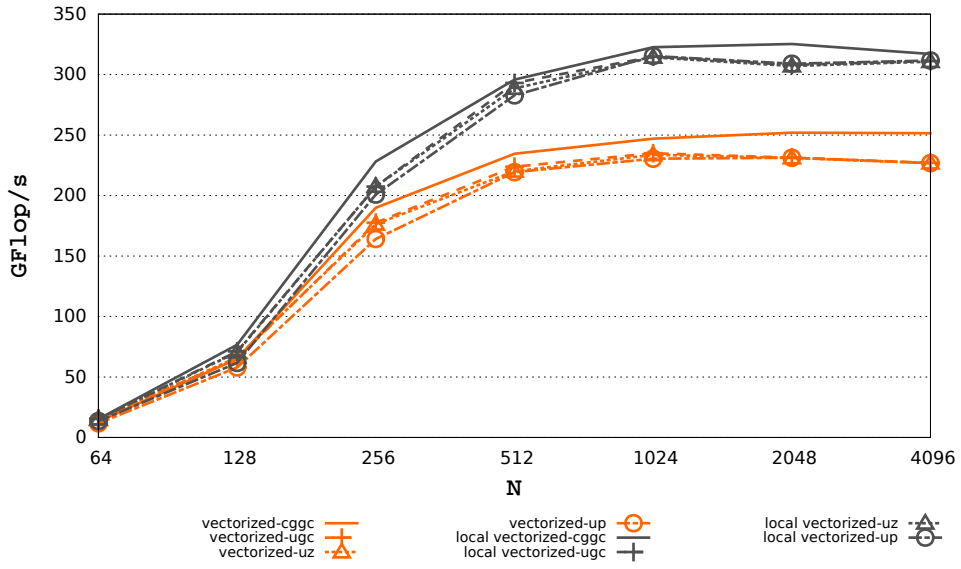



FIGURE 6.11: The impact of **data placement strategies** on the performance of the **matrix multiplication** kernel on the **Kaveri** APU.

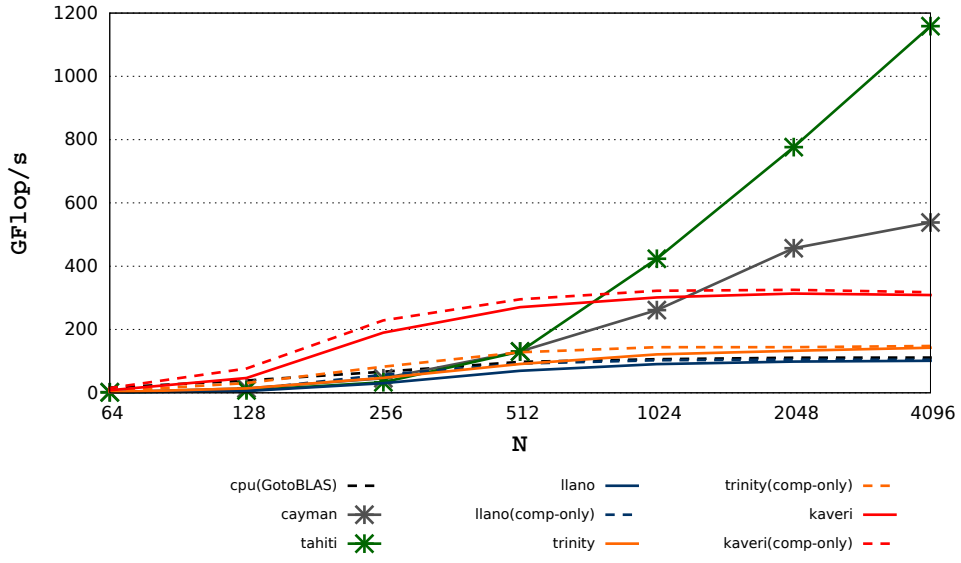



FIGURE 6.12: Performance **comparison** of the **matrix multiplication** best implementations on **CPU**, **GPUs** and **APUs** as a function of $N$, the dimension of the used square matrices. The OpenCL driver (**version 15.4**) used by the **GCN** based devices (the Kaveri APU and the Tahiti GPU) is newer than the driver used by the other devices (**version 13.4**).

of APUs without taking into consideration the communication times. Ideally, the upcoming AMD APUs (Carrizo and Zen APU) that feature a unified memory system will have a comparable performance to these results (*llano(comp-only)*, *trinity(comp-only)* and *kaveri(comp-only)* in figure 6.12). Before the Kaveri APU, the CPU was the most suitable choice for small matrices sizes (below 192), since communication times associated to the GPUs are important. It is no longer the case if we take into consideration the performance numbers obtained on the Kaveri APU, which outperform those of the CPU and that for all the matrices sizes. For medium-sized matrices (between 192 and 700), the Kaveri APU with a unified memory system (*kaveri(comp-only)*) may outperform both discrete GPUs and the CPU. Large matrices have to be sent to and processed by discrete GPUs.

### 6.2.2 Finite difference stencil

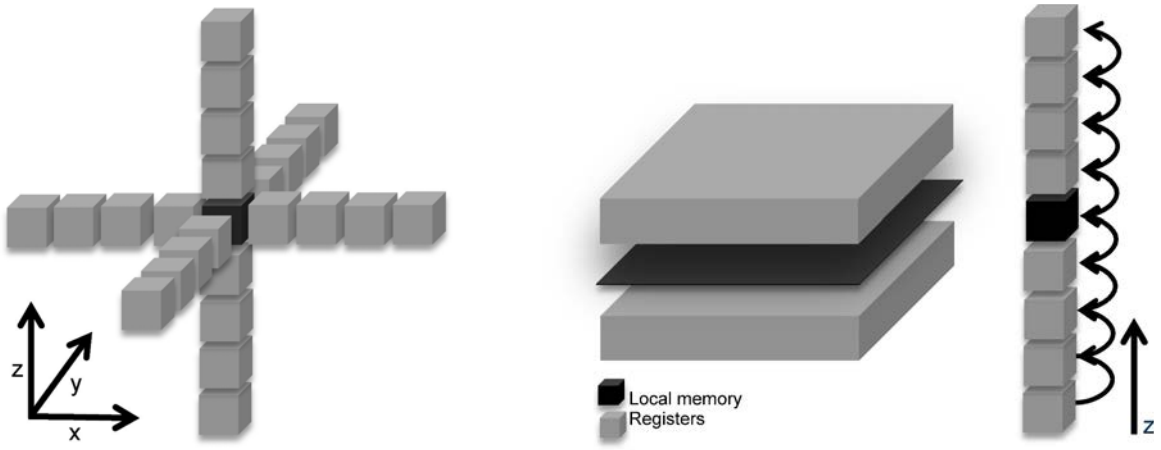
In this section, we describe the implementation details of a 3D finite difference stencil OpenCL kernel. We optimize and adapt the OpenCL kernel to the surveyed hardware with the help of auto-tuning, and present the performance results accordingly. Besides, we study the impact of data snapshotting on the performance of the different architectures, and the impact of the data placement strategies on the performance of the APUs. Finally, we conduct a comparison between the performance numbers of the 3D finite difference stencil kernel on CPUs, discrete GPUs and APUs.

#### 6.2.2.1 Implementation details

Finite difference stencil computations are the core of a wide range of physics simulations, e.g. RTM. They approximate the derivation operators in physics equations into linear combinations on a discrete domain. Due to their regularity, they are very well suited for efficient executions on the massively parallel architecture of GPUs and APUs.
A stencil computation of an element in a 3D $N$x$N$x$N$ regular mesh is a linear summation of its value and its neighboring values, along each dimension, weighted by specific coefficients. The building block, i.e. the differentiation operator, of our targeted seismic applications is a 3D $8^{th}$ order centered stencil as illustrated in figure 6.13a. We choose to evaluate such an operator whose workflow is presented in algorithm 6.1. The algorithm depicts the linear combination, which may correspond to a given time-step $t$ of a numerical simulation, applied to one grid point of a computational domain $u0$ situated at the position $(x, y, z)$. The floating point computation and data storage complexities of the stencil kernel are both $O(N^3)$. The compute intensity is thus $O(1)$ and this algorithm is memory bound. We apply a 2D work-item grid on the 3D domain and we first implement a *scalar* version in which each work-item computes $X$ columns of the domain ($X$ refers to the ILP and is determined via auto-tuning and is either equal to 2, 4 or 8) along the Z dimension which corresponds to the largest strided memory accesses [74]. All memory accesses are issued in global memory. Second, we explicitly vectorize the code and each work-item processes $X$ columns of *float4* elements along the $Z$ dimension. In most cases $X$ is equal to 2 or 4. This implementation is referred to as *vectorized*. Finally, we implement a blocking algorithm [156]. Data is fetched slice by slice from global memory and stored in local memory and each work-item sweeps along the Z dimension and blocks the grid values in the register file in order to reuse data (see figure 6.13b), and computes $X$ *float4* elements at a time. This implementation is

**Algorithm 6.1** The 3D $8^{th}$ order centered finite difference stencil operator applied to one grid point $u0(x, y, z)$ at a time-step $t$.

1: **function** fd_stencil_compute($u0$, $x$, $y$, $z$, $t$)
2: **return** $coef0 * u0(z, y, x)$ ▷ $coefn, n \in [0..4]$ are the stencil coefficients
3: $+ coef1 * (u0(z, y, x + 1) + u0(z, y, x - 1)$
4: $+u0(z, y + 1, x) + u0(z, y - 1, x)$
5: $+u0(z + 1, y, x) + u0(z - 1, y, x))$
6: $+ coef2 * (u0(z, y, x + 2) + u0(z, y, x - 2)$
7: $+u0(z, y + 2, x) + u0(z, y - 2, x)$
8: $+u0(z + 2, y, x) + u0(z - 2, y, x))$
9: $+ coef3 * (u0(z, y, x + 3) + u0(z, y, x - 3)$
10: $+u0(z, y + 3, x) + u0(z, y - 3, x)$
11: $+u0(z + 3, y, x) + u0(z - 3, y, x))$
12: $+ coef4 * (u0(z, y, x + 4) + u0(z, y, x - 4)$
13: $+u0(z, y + 4, x) + u0(z, y - 4, x)$
14: $+u0(z + 4, y, x) + u0(z - 4, y, x))$
15: **end function**




(a) Memory layout.

(b) Blocking algorithm.

FIGURE 6.13: The memory layout and the memory blocking layout of the 3D $8^{th}$ order finite difference stencil.

referenced as *local vectorized*. Similarly to the matrix multiplication kernel, we consider the *local scalar* version for the GCN based hardware, namely for the Tahiti GPU and for the Kaveri APU. Note that we also use auto-tuning to adapt at best the stencil implementations to each architecture.

#### 6.2.2.2 Devices performance

We first show the performance numbers of each tested device based on the kernel execution time only. We use 3D domains with $N \times N \times 32$ sizes ($N$ ranging between 64 and 1024). We use the AMD Catalyst OpenCL driver version 13.4 except for the Kaveri APU, where we use the driver version 15.4, and where we consider the *local scalar* implementation similarly to section 6.2.1. Moreover, we update the performance numbers of the Tahiti GPU using the driver version 15.4, and thus present its performance numbers

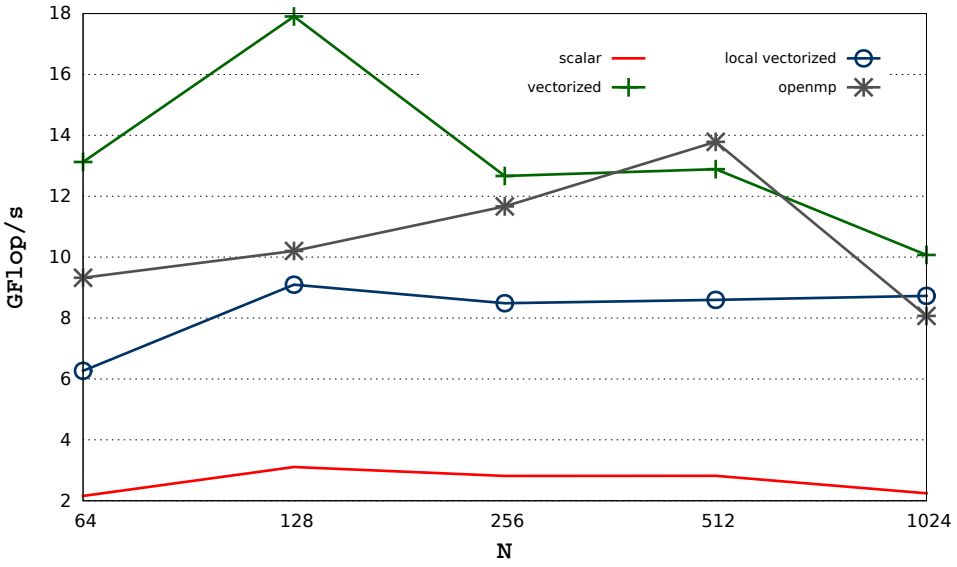



FIGURE 6.14: The performance numbers of the **3D finite difference stencil** OpenCL kernel, along with an OpenMP implementation, on the ***AMD Phenom TM II x6 1055t Processor*** as a function of the problem size **N**×**N**×**32**.

using both 13.4 and 15.4 driver versions. We also consider the *local scalar* version for Tahiti, given that in the following sections and in the rest of the document we restrict our study to the most recent hardware coupled with the most recent driver at the time of writing.

Figure 6.14 summarizes the performance of the different OpenCL implementations on the CPU. We compare them against an OpenMP Fortran 90 code (compiled and vectorized with Intel Fortran Compiler). Here again the OpenCL vector implementations outperform the OpenCL scalar version thanks to their use of SSE instructions. The *vectorized* implementation efficiently relies on CPU caches and delivers the best performance being faster than or as fast as the OpenMP implementation.

Figures 6.15, 6.16, 6.18, 6.19 and 6.20 illustrate the performance numbers of Cayman, Tahiti, Llano, Trinity and Kaveri respectively. Based on the performance results obtained using the driver version 13.4, we note that for discrete GPUs, as well as for the two integrated GPUs of Llano and Trinity, the *local vectorized* implementation is more efficient than the *vectorized* one thanks to the blocking in the local memory, which is an expected behavior. In addition, the *scalar* implementation gives low performance numbers on the GPUs based on the GCN micro-architecture (with a scalar design), i.e. Tahiti, where the performance of the *scalar* implementation is almost 2× slower than that of the *vectorized* and almost 4× slower than that of the *local vectorized*. The best performance of Tahiti, based on the figure 6.16, reaches up to 484 $GFlop/s$ and that of the Trinity APU reaches up to 50 $GFlop/s$.

The figure 6.17 represents the updated performance numbers obtained on Tahiti when using the driver version 15.4. We can instantly observe that the new driver helped benefit from the scalar design of the Tahiti GPU. The scalar versions either perform as good as the vectorized versions or outperform the vectorized implementations. As a matter of fact, the *local scalar* version gives the best performance numbers that reach 544 $GFlop/s$ which represents a performance enhancement of 12% compared to the numbers

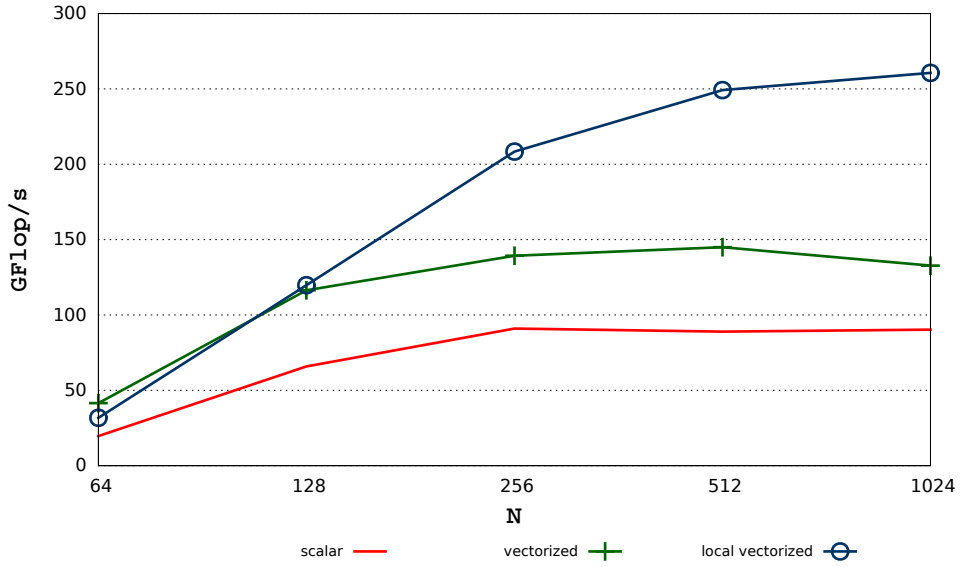



FIGURE 6.15: The performance numbers of the **3D finite difference stencil** OpenCL kernel on the **Cayman** GPU as a function of the problem size **N**×**N**×**32**.

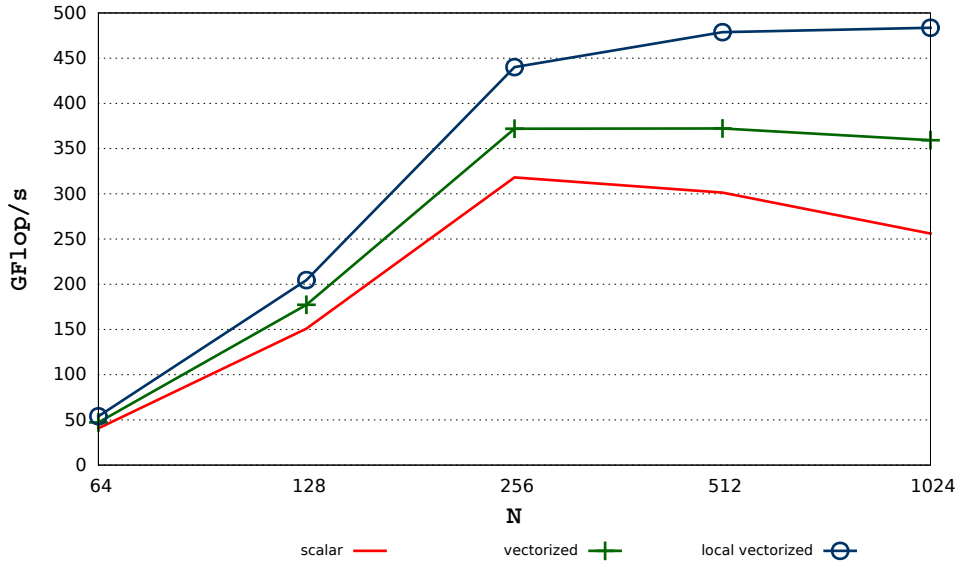



FIGURE 6.16: The performance numbers of the **3D finite difference stencil** OpenCL kernel on the **Tahiti** GPU as a function of the problem size **N**×**N**×**32**.

obtained with the driver version 13.4. In one hand, this leads to the conclusion that the vectorization is no longer needed to deliver the best OpenCL performance of the 3D finite difference kernel out of the Tahiti GPU. On the other, it shows that applications performance is sensitive to driver upgrades which implies that the drivers we are using in the scope of this work, are stable enough on GCN based GPUs.
When it comes to the performance numbers of the 3D stencil kernel on the Kaveri APU, presented in figure 6.20, we reached the same conclusions we have elaborated based on the performance results on Tahiti using the driver version 15.4. We can notice that in figure 6.20 the *local vectorized* delivers better performance numbers compared to the performance of the *vectorized* version thanks to the blocking algorithm of the *local vectorized* implementation. Besides, the scalar design of Sea Islands GPUs seems to be

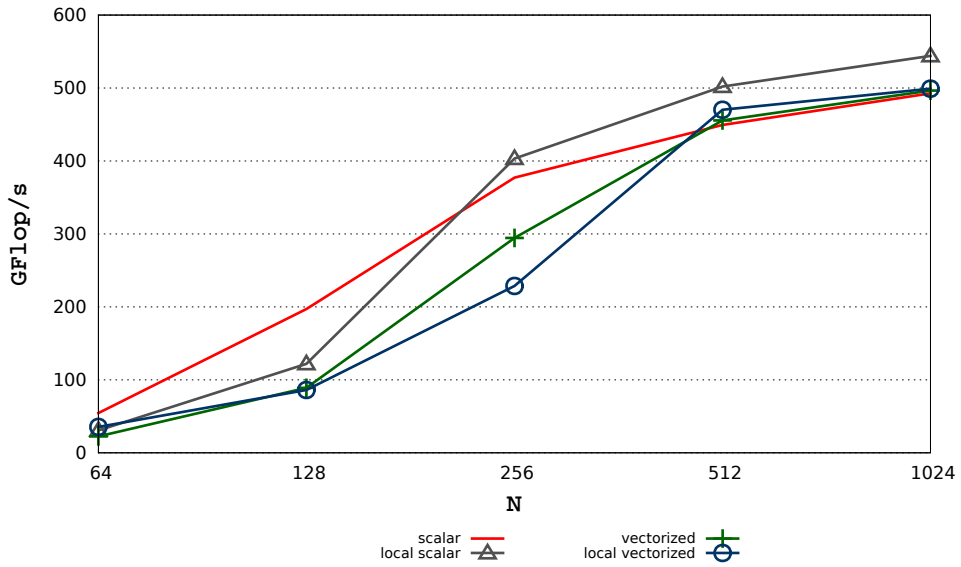



FIGURE 6.17: The updated performance numbers (after **driver upgrade to version 15.4**) of the **3D finite difference stencil** OpenCL kernel on the **Tahiti** GPU as a function of the problem size **N**×**N**×**32**.

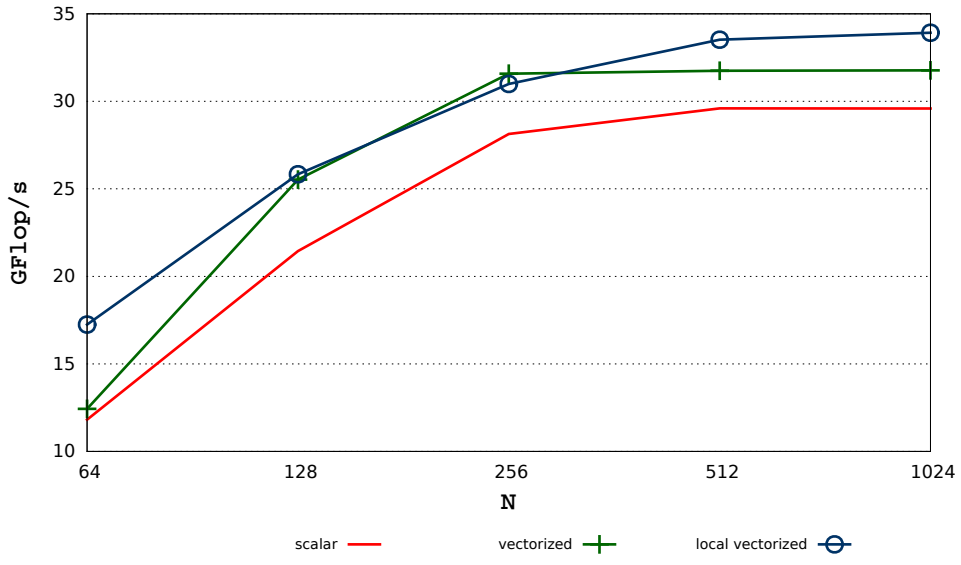



FIGURE 6.18: The performance numbers of the **3D finite difference stencil** OpenCL kernel on the **Llano** APU as a function of the problem size **N**×**N**×**32**.

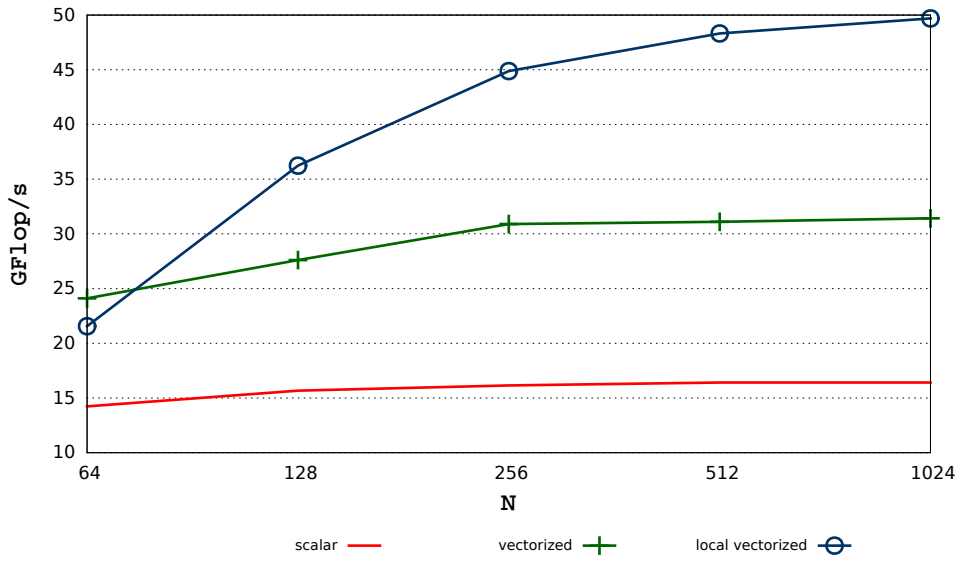



FIGURE 6.19: The performance numbers of the **3D finite difference stencil** OpenCL kernel on the **Trinity** APU as a function of the problem size **N**×**N**×**32**.

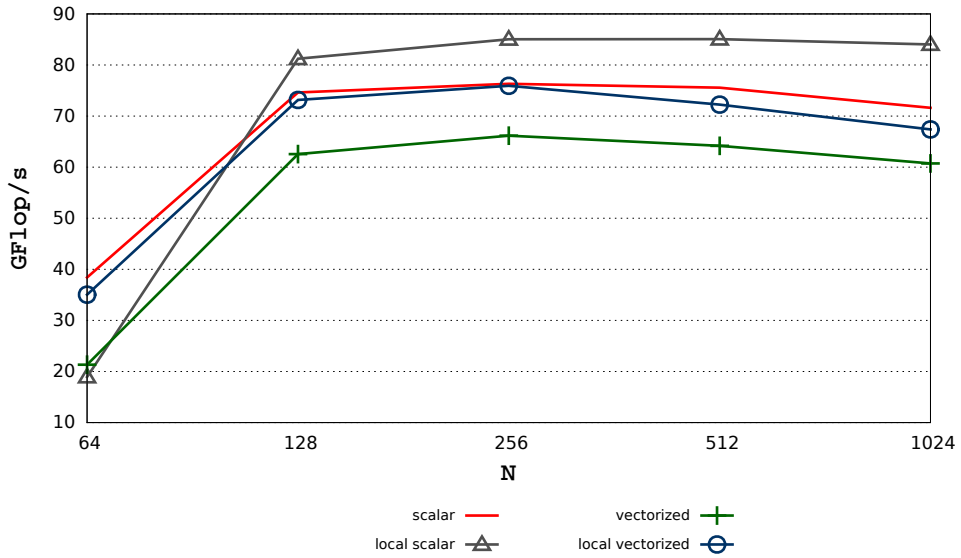



FIGURE 6.20: The performance numbers of the **3D finite difference stencil** OpenCL kernel on the **Kaveri** APU as a function of the problem size **N**×**N**×**32** (the **OpenCL driver version 15.4** is used).

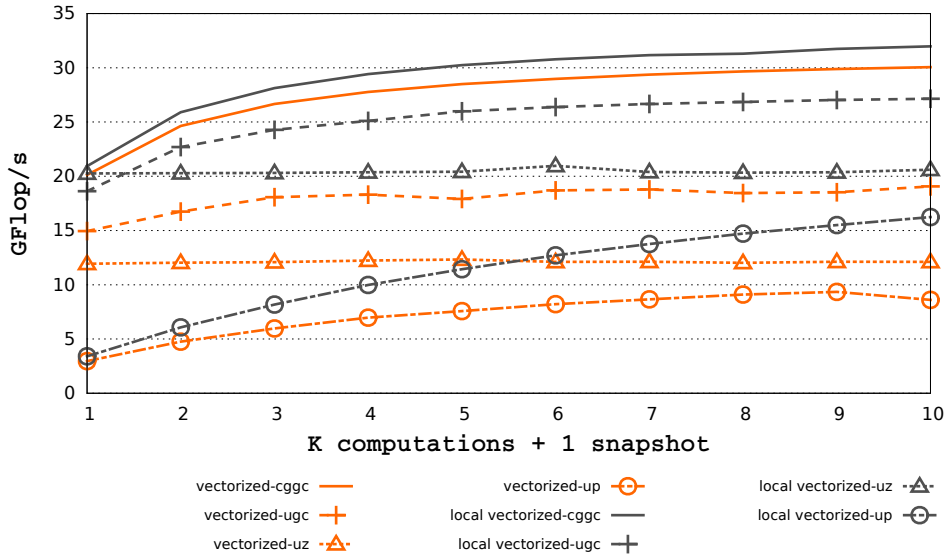



FIGURE 6.21: The impact of **data placement strategies** on the performance of the **3D finite difference stencil** kernel on the **Llano** APU, with respect to the frequency of **data snapshotting**.

more beneficial to the performance of the 3D stencil kernel. Indeed, both of the *scalar* and *local scalar* versions outperform the implementations that are based on explicit code vectorization, i.e. the *vectorized* and *local vectorized*. Furthermore, the *local scalar* implementation delivers higher performance numbers compared to the *scalar* version. The *local scalar* implementation gives the best 3D stencil performance numbers for medium and big problem sizes (with $N \geqslant 128$) that reach up to 85 $GFlop/s$. Finally, it is to be noted that OpenCL image memory objects provided very poor performance for large domains, as opposed to the matrix multiplication kernel. Therefore, we do not consider using them for the stencil kernel.

#### 6.2.2.3 Impact of data placement strategies on performance

Stencil computations are usually used in iterative methods. Between two subsequent iterations data need to be resident on the CPU memory in order to be used in further operations such as *imaging condition* in the RTM application. We call this process *data snapshotting* (see section 2.2). The frequency of data snapshotting can also be an important performance factor as the memory model of the APUs is different from that of the discrete GPUs. We run the best two implementations of the stencil OpenCL kernel on APUs, i.e *vectorized* and the *local vectorized* implementations for Llano and Trinity and the *scalar* and the *local scalar* implementations for Kaveri, while taking into consideration the data placement strategies selected in section 6.1 in one hand, and the frequency of data snapshotting in another. Note that we use one input buffer and one output buffer. In figures 6.21, 6.22 and 6.23, we show the performance results of the kernel ran on a $1024 \times 1024 \times 32$ grid, respectively on Llano, Trinity and Kaveri, as a function of the number of stencil computation passes performed before the snapshot (frequency of data snapshotting). Note that the communication times (possibly required to retrieve the snapshot on the CPU) are included in those numbers. We conclude that in order to obtain the best stencil performance on APUs, we have to use the *local*

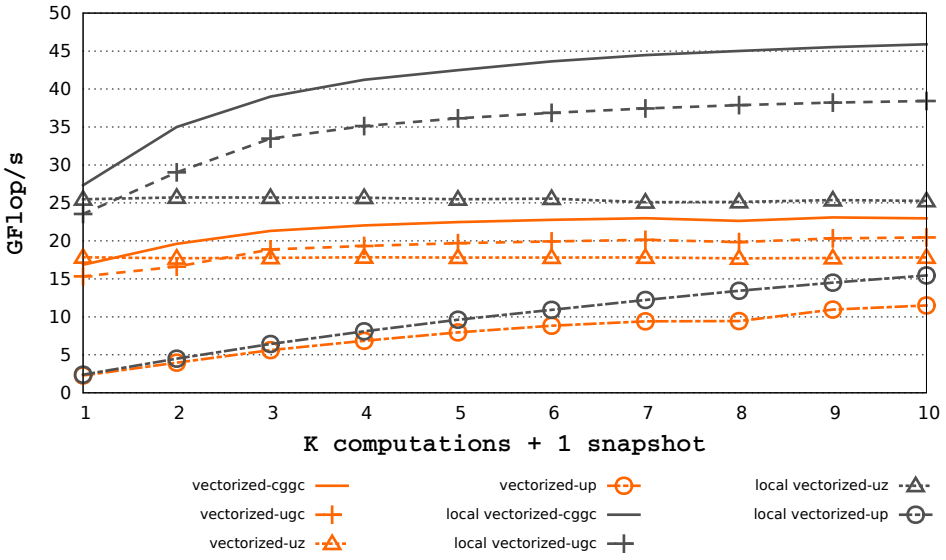



FIGURE 6.22: The impact of **data placement strategies** on the performance of the **3D finite difference stencil** kernel on the **Trinity** APU, with respect to the frequency of **data snapshotting**.

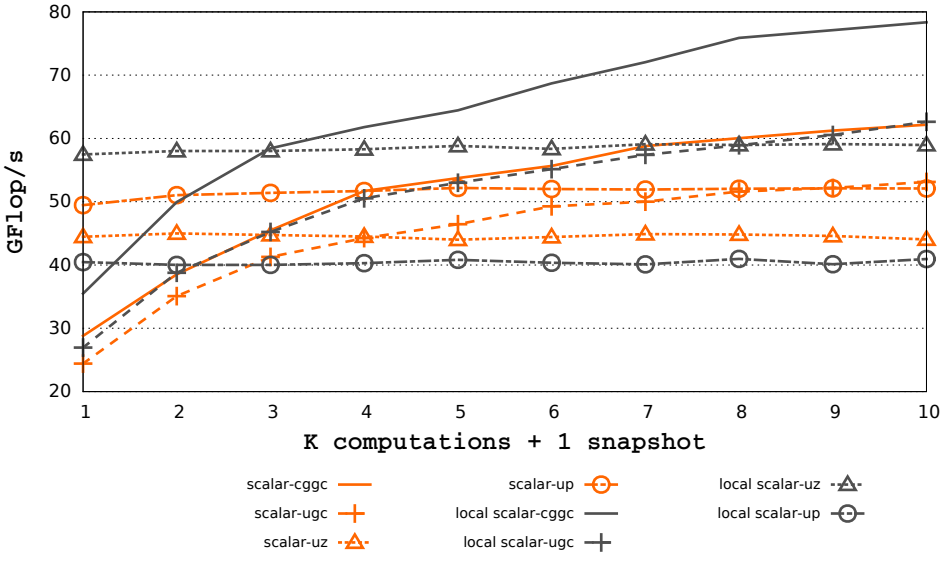



FIGURE 6.23: The impact of **data placement strategies** on the performance of the **3D finite difference stencil** kernel on the **Kaveri** APU, with respect to the frequency of **data snapshotting**.

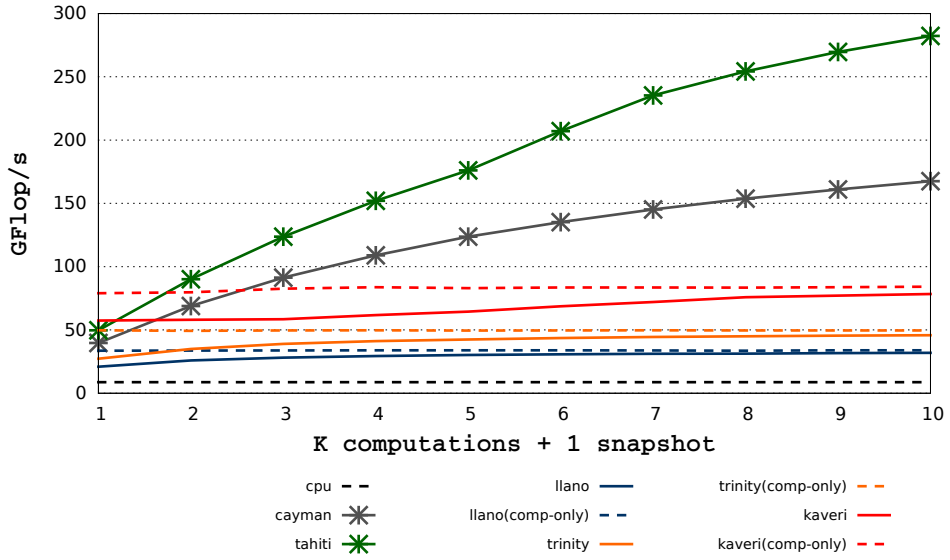



FIGURE 6.24: Performance **comparison** of the **3D finite difference stencil** best implementations on **CPU**, **GPUs** and **APUs** as a function of the data snapshotting frequency. The OpenCL driver (**version 15.4**) used by the **GCN** based devices (the Kaveri APU and the **Tahiti** GPU) is newer than the driver used by the other devices (**version 13.4**).

*vectorized* implementation coupled with the `cggc` data placement strategy for both Llano and Trinity APUs. However, the `uz` strategy (with zero-copy memory buffers) applied to the *local scalar* implementation on Kaveri gives higher performance numbers than the other data placement strategies when considering a snapshot retrieval after every 1 to 3 computations. Based on these numbers, the Kaveri APU is the first APU that allows a performance gain of the finite difference stencil kernel when zero-copy memory objects are used. Finally, it is important to mention that the data placement strategy `zz`, which allows on the Kaveri APU to map the entire system memory to the integrated GPU memory and also allows to use the zero-copy memory objects in both read and write accesses, delivers exactly the same performance numbers as the `uz` and thus it is not added to the figure 6.23 for the sake of clarity.

#### 6.2.2.4 Performance comparison

Finally, the figure 6.24 illustrates a performance comparison of all the tested devices as a function of the frequency of data snapshotting using the best implementation on each. According to the figure 6.23, the best performance numbers of the 3D finite difference stencil kernel on Kaveri, with respect to the data snapshotting frequency, are obtained by combining *local scalar-uz* (for the data snaphotting frequencies ranging from 1 to 3) and *local scalar-cggg* (for the data snaphotting frequencies ranging from 3 to 10). Therefore, the red solid line in figure 6.24 illustrates this combination. We recall that, the domain size of the used grid in this test is 1024x1024x32.
We notice that integrated GPUs outperform the CPU implementation for all the data snapshotting frequencies, and that the best performance is always delivered by the Tahiti discrete GPU, except for the case $K = 1$. Indeed, for a high rate of data snapshotting, more specifically for a snapshot retrieval after every stencil computation, the Kaveri APU outperforms the discrete GPUs. The upcoming APUs are expected to feature a

perfectly and fully unified memory, which would allow even higher performance (equal to the *kaveri (comp-only)* in the figure 6.24) for lower data snaphotting frequencies.

As a conclusion, we can advance that the APU can be an attractive solution for a the finite difference kernel with a high rate of data snapshotting, even though a high rate of data snapshotting does not necessarily correspond to that used in realistic stencil based seismic applications such as the RTM (the frequency is more between 10 and 18). Depending on the snapshotting frequency, the `uz` or the `zz` data placement strategies may deliver higher performance numbers than the straightforward `cggc` strategy. However, the `uz` placement strategy can not be used when implementing the RTM on APUs, since memory buffers involved in this strategy can not be accessed in both read and write modes, which is required by the RTM algorithm. Therefore, for the rest of the document, we will rather consider the `zz` strategy (along with `cggc`) which also allows the integrated GPU to allocate large read-write memory buffers which is more suitable to the seismic applications in general and to the RTM in particular.

## 6.3 Power consumption aware benchmarks

In this section we try to quantify the power efficiency of the AMD APUs and compare it against that of discrete GPUs and that of CPUs. For that to do, we chose to measure the power consumption of the two applicative benchmarks, i.e. the matrix multiply and of the 3D finite difference stencil, and compute the power efficiency (using the Performance Per Watt (PPW) metric) based on the best performance obtained on each platform, which are detailed in section 6.2. It is to be noted that we chose to restrict this evaluation to the most recent hardware generation of each architecture. Consequently, only the *AMD Phenom TM II x6 1055t Processor*, Tahiti and Kaveri are surveyed. Besides, we start this section with a tutorial where we describe our approach for power measurement, and where we enumerate state-of-art metrics used to evaluate the power efficiency.

### 6.3.1 Power measurement tutorial

Recently, the awareness about power consumption has sustainably raised. The Green500 list [6] ranks computers from the TOP500 list of supercomputers in terms of energy efficiency. The inaugural list was elaborated in 2007 and a new era of *green computing* has began. A detailed guide [91] is published in order to define the requirements a supercomputer should meet to be part of the Green500 ranking.
Several methods for measuring power usage on current architectures have been proposed [49, 100, 204]. They differ in the tools used for measuring power consumption (hardware based methodologies or software based methodologies) and in the various places where valid measurements can be collected (depending on the size of the HPC facility: a cluster, a rack, a cabinet, a workstation or a hardware component such as a CPU, an APU and a GPU).
Despite these efforts, in the HPC community, the measurement process remains an ad-hoc process and is slowly being standardized. Yet many questions can't be easily answered: how should the tested hardware be configured? What power consumption should be measured, that of the compute elements or that of the whole system? When

should the power be measured? For a certain portion or for the entire execution of an application?

#### 6.3.1.1 Metrics for power efficiency

Having defined how to measure power consumption of a HPC facility, the question remains how to use the collected data in order to conduct a fair comparison between the different technologies. Many metrics can be used to calculate a power efficiency and evaluate the power consumption of a given compute node.

- **Performance Per Watt ($PPW$):** measures the rate of computation that can be delivered by the compute node (in Flop/s) for every watt of power consumed.
- **Energy Delay Product ($ED^n$):** represents the energy consumed by the application multiplied by its execution time (to the power of $n$ with $n = 1, 2...$). According to [113], this is biased towards large supercomputers. A large $n$ value not only emphasizes the performance aspect of a HPC system, but it also exaggerates the performance gained from the massive parallelism. More specifically, the $ED^n$ metric with $n > 2$ increases exponentially with respect to the number of processors in a supercomputer.
- **Power Delay Product:** the product of the execution time with the dissipated power. This is appropriate for the evaluation of low-power systems. Indeed, since the power consumption numbers are relatively small, the execution time can be used as a multiplier to increase those numbers.

In the scope of this work we make use of the PPW metric to evaluate the power efficiency of the surveyed hardware.

#### 6.3.1.2 Proposed methodology

This tutorial serves as a practical guide for measuring the power of *a system as a whole*. That is, the measurement is not only restricted to the processor handling computations on a given computer, but encompasses all the computer including its memory, its hard-drives, etc. This choice is motivated by the fact that a discrete GPU is not standalone and needs a CPU to drive it during computations. This CPU consumes power and its power drawing should be taken into consideration. We refer to the section 3.3 for more details. We consider measuring the consumption of one compute node at a time, i.e. a computer that contains either a CPU, or a CPU and a discrete GPU or an APU, using a hardware based methodology. The collected data can then be extrapolated to estimate the consumption of a bigger HPC facility. Minimum equipment requirements are the followings.

- **The compute node to be tested:** this can be a cluster node or a workstation that contains a high performance CPU, a discrete GPU (with a commodity CPU) or an APU.

| Vendor | #Outlets | Price |
|---|---|---|
| *APC by Schneider Electric* | 10 | $1300 |
| *Server Technology* | 10 | $1200 |
| *Raritan* | 8 to 16 | $755 |
| *Watts Up?* | 1 | $235 |

TABLE 6.2: Power meters major vendors.

- **A digital power meter:** that is an interface that measures and monitors the current and voltage delivered to the compute node. As a matter of fact, power meters often report the *active power* (also referred to as *real power*) and *apparent power* of a given power supply unit (PSU). The active power is measured in Watts and is the power drawn by the resistive electrical components of a system doing useful work. The apparent power includes, on top of the real power, the power drawn by the reactive components (such as inductors and capacitors). The apparent power is measured in Volt-Amperes (VA). The ratio of real power to apparent power is defined as the *power factor*. The closer the power factor of a PSU is to 1, the lower cost (in terms of electric bills) it would incur. The power factor usually ranges between 0.8 and 0.9 on modern computer PSUs. Besides, it is crucial to make sure that the PSU that is installed on a given computer suits its main processor in terms of electric efficiency. The electric efficiency, of a PSU, is the ratio between the DC power it converts and the input AC power it pulls from of the wall outlet. In order to standardize the efficiency of PSUs, the 80 Plus certification [16], a convention used by the PSU manufacturers to validate that their PSUs were at least 80% efficient at 25% of full load, was created. Say, we would like to connect a 500 W PSU to an APU based computer that draws at most 60 W for a particular computation workload. Given that 25% of 500 W is 100 W, the PSU will be likely inefficient (its electric efficiency will be below 80%). However, a 200 W PSU would be much more efficient for this configuration.
  A digital power meter can be a simple power outlet or a Power Distribution Unit (PDU) designed to distribute and monitor power within a rack of compute nodes providing a remote access to the collected data. Common interfaces include RS-232, USB or a LAN network-controller accessible through a variety of protocols: Telnet, SSH, SNMP or HTTP. This allows an administrator to access a PDU from a remote terminal. Table 6.2 summarizes the list of the major power meters vendors. In the scope of this work a *Raritan PX (DPXR8A-16)* PDU is used, and only the active power is measured.

- **A power meter software:** used to process the collected data by the power meter.

- **A logging PC:** an isolated workstation exclusively used for sampling and logging the power measurement data.

- **Applicative benchmarks:** an appropriate workload that can be submitted to the compute node and keep it busy while recording the power measurement samples. For the sake of fairness, it is recommended to use the same execution environment for all the hardware components. It is also recommended to choose the most highly optimized application implementation (which may require different APIs or compilers) on each hardware configuration.

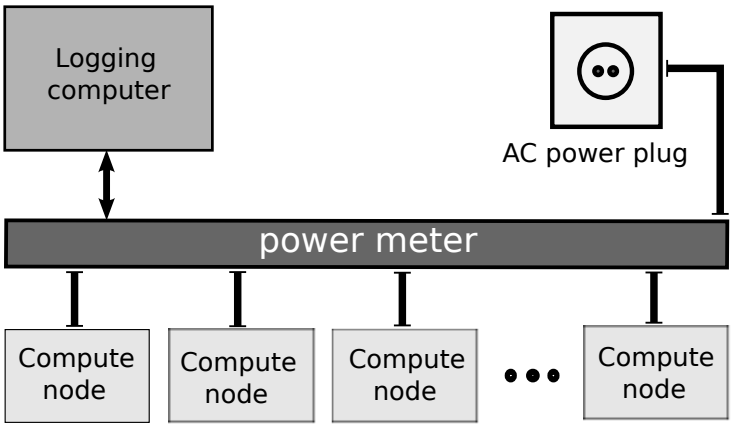



FIGURE 6.25: Connecting the power meter. The compute nodes are not necessarily homogeneous, they might be populated by CPUs, discrete GPUs or APUs.

The digital power meter should be, first, connected to the measured unit (compute node). Second, it can be connected to the logging workstation after choosing the communication protocol. Finally, the meter should be plugged to the AC power outlet. Figure 6.25 illustrates the connection between the different devices. Note that in the scope of this work, three compute nodes based on three different processors (a CPU, an APU and a discrete GPU) are connected to the power meter. Besides, it is to be noted that all the affected hardware should be powered off prior these operations. Following, we enumerate the power measurement steps that we recommend to follow before any power consumption aware benchmark.

1. Turn the power meter on.
2. Launch the logger software on the logging PC.
3. Start recording the power measurement samples: the logging workstation probes the meter in order to extract many measurement samples.
4. Immediately, launch the applicative benchmarks on the compute node. It is important to consider running several iterations in order to keep the node loaded and generate realistic measurements.
5. Stop recording the power measurement samples once the applicative benchmark has returned.
6. Save the performance numbers of the applicative benchmarks.
7. Calculate the unit average power (or any other mean operator such as RMS) from the recorded power measurement data. It is recommended to remove the first and the last samples as they may be inaccurate due to synchronization problems.

In Figure 6.26 we present a simplified model of a classic compute node. Hardware components are symbolically classified in two categories:

- ***computing components***, that is the processing units that can be present in a workstation and used for compute such as CPUs, APUs or GPUs. We refer to the power consumption of this category as $P_{computing}$;

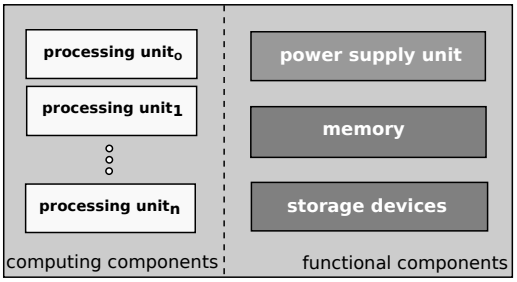
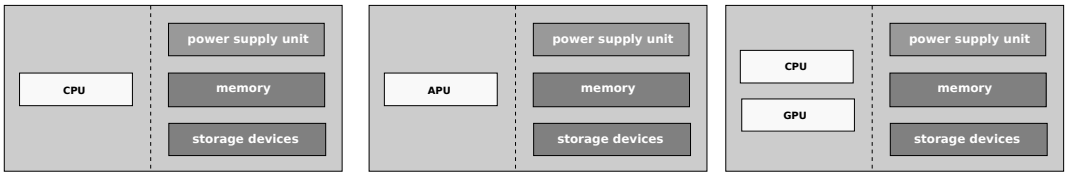



FIGURE 6.26: A simplified model of a compute node




(a) System 1. (b) System 2. (c) System 3.

FIGURE 6.27: The composition of the tested compute nodes.

- ***functional components***, are the other major components required for a system to function correctly (i.e power supply unit, memory, storage devices etc). We refer to the power consumption of this category as $P_{functional}$.

Thus the power consumption of a system $P_{system}$ can be defined as following:

$$P_{system} = P_{computing} + P_{functional} \tag{6.1}$$

Given that we only measure the $P_{system}$ and that we aim to accurately compare the $P_{computing}$ of the different hardware configurations, the affected compute nodes should, ideally, have similar *functional components*. This can be done by installing the same hard drives and amount of memory in each system. Note that the power consumption of systems that use discrete GPUs as co-processors should include both the power consumption of the CPU and that of the GPU power since GPUs are not standalone.

#### 6.3.1.3 Hardware configuration

We illustrate the hardware configuration of the three systems that we study in this section in figure 6.27. The *System 1* contains the *AMD Phenom TM II x6 1055t Processor* CPU, the *System 2* has the Kaveri APU as main processor (note that the memory is over-clocked in this system in order to obtain the best memory bandwidth possible) and the *System 3* is equipped with the Tahiti discrete GPU along with a commodity CPU (*AMD Phenom TM II x6 1055t Processor*). *System 1* and *System 3* even have the same CPU and the same mother board, while the mother board of *System 2* is different as the APU requires a special socket and chipset. Besides, the PSU of each system should be carefully picked with respect to the corresponding maximum power load. As a matter of fact, *System 1* is equipped with a 500 W PSU, that of *System 2* is 200 W and the *System 3* has a 750 W PSU, all of them are 80 Plus certified.

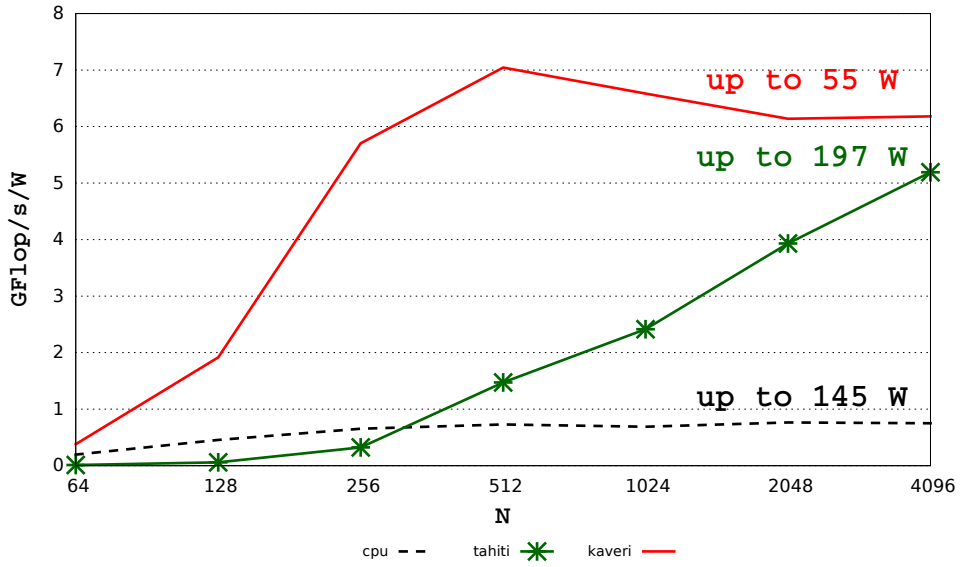



FIGURE 6.28: Performance comparison of the matrix multiply power efficiency (PPW) on the *AMD Phenom TM II x6 1055t Processor*, the Tahiti discrete GPU and the Kaveri APU as a function of $N$, the dimension of the squared matrices.

#### 6.3.1.4 Choice of applications and benchmarks

The applications should keep the processing units busy enough to report relevant measures. If one application is not time consuming, it is highly recommended to subsequently run it several times. In the scope of this work, all the memory transfers between CPU and GPU are considered when measuring the power consumption. In addition the I/O operations, such as data snapshotting for stencil computations, are also taken into considerations as they are relevant to the algorithms of the studied applications.

### 6.3.2 Power efficiency of the applicative benchmarks

We now focus on measuring the power efficiency of the matrix multiply and of the 3D finite difference stencil on the different hardware configurations detailed in figure 6.27. We follow the tutorial described in section 6.3.1 to measure the power consumption and the performance numbers of the best implementation of each benchmark on each hardware configuration. We then determine the power efficiency by computing the PPW of the matrix multiply and of the 3D finite difference stencil on CPU, APU and discrete GPU. The figure 6.28, represents the power efficiency of the matrix multiply application. First, we note that the *AMD Phenom TM II x6 1055t Processor* CPU offers a very low power efficiency (approximately 0.8 $GFlop/s/W$) compared to that of Tahiti and Kaveri, despite the excellent performance numbers delivered by the GotoBLAS implementation. This is due to the high power envelope measured on the System 1 (up to 145 $W$). Second, the figure also demonstrates that the AMD discrete GPU Tahiti delivers a high performance efficiency that reaches up to 5 $GFlop/s/W$. Finally, the red plot in the figure shows that the Kaveri APU surpasses the Tahiti GPU in terms of power efficiency which reaches 7 $GFlop/s/W$ for medium sized matrices. We note a withdrawal for big sized matrices, this is due to quick jump in terms of power consumption (from $42W$ up to $50W$) when the matrices are big ($N \geq 1024$). It is to be noted that the data placement

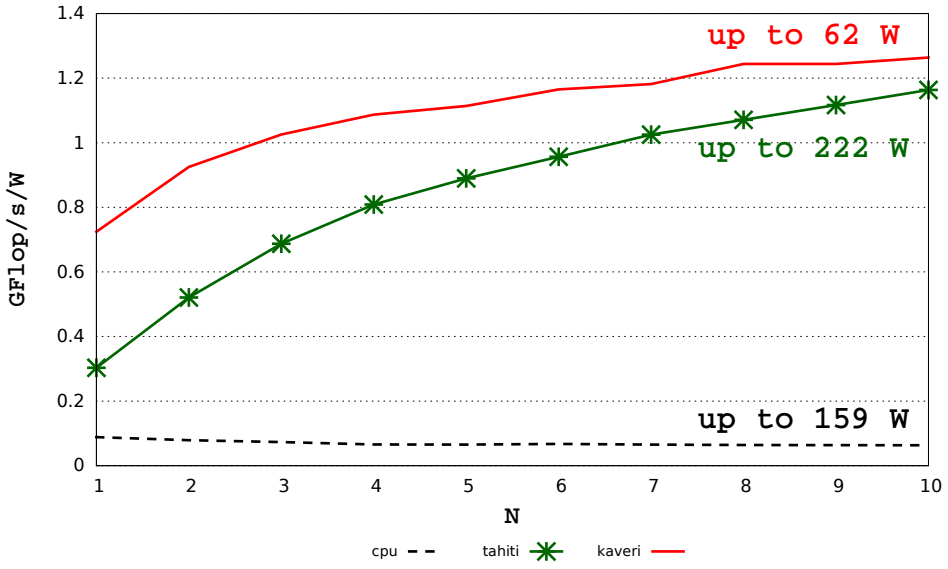



FIGURE 6.29: Performance comparison of the 3D finite difference power efficiency (PPW) on the *AMD Phenom TM II x6 1055t Processor*, the Tahiti discrete GPU and on Kaveri APU as a function of $N$, as a function of the snapshotting frequency.

strategy used here is `cggc` which was found, in section 6.2.1, to be the strategy that generally offers the best performance. However, we noticed that the power consumption of the APU is not affected by the choice of the data placement strategy.

Similarly, we present the power efficiency numbers of the 3D finite difference stencil application in figure 6.29. At first glance, we realize that the power efficiency of the 3D finite difference stencil is almost 4× lower than that of the matrix multiply benchmark. This is due to the fact that the 3D stencil application is memory bound which incurs lower performance numbers in the one hand, and higher power envelope (for example on Kaveri the 3D stencil application draws up to 62 W as opposed to 50 W for the matrix multiply application). This is due to the fact that a floating point operation consumes less than a memory access, especially when the memory location has a high overhead. The power efficiency of the *AMD Phenom TM II x6 1055t Processor* CPU is very poor (less than to 0.08 $GFlop/s/W$) compared to the power efficiency of the rest of the tested hardware. The power efficiency of the Tahiti discrete GPU merely reaches 1.1 $GFlop/s/W$. That of the Kaveri APU is slightly higher than that of the Tahiti GPU for all the data snapshotting frequencies and goes up to 1.25 $GFlop/s/W$.

As a conclusion, we demonstrated that despite the 3.3-fold difference in terms of sustained performance between the APUs and the discrete GPUs (see figure 6.24), the APUs are more power efficient than discrete GPUs for memory bound and compute bound workloads, and that for all the data snapshotting frequencies in the case of the 3D finite difference stencil kernel. The gain factor in terms of power consumption of the Kaveri APU compared to the discrete GPUs goes up to 3.59× (see figure 6.29). The upcoming APUs (such as Carrizo and the Zen APU) can be even more power efficient and may deliver higher PPW. Therefore, it is mandatory to take into consideration the low power consumption, seen as an important asset of the APU technology, while studying the performance of seismic applications on APUs, further in this document.

## 6.4 Hybrid utilization of the APU: finite difference stencil as an example

This section corresponds to a Master 2 thesis conducted by P. Eberhart in summer 2013.
The AMD APU has both CPU and GPU cores, and shared memory between them. Until now, we have only considered using the integrated GPU of the APU in order to implement the matrix multiplication and the 3D finite difference stencil OpenCL kernels. We have shown in section 6.2.2 the potential of APU integrated GPUs for the 3D finite difference stencil application, especially when considering the data snapshotting. But, apart from handling the GPU, the CPU cores are idle during the computations on the integrated GPUs. In this section, we extend the 3D finite difference OpenCL kernel and try to use both the CPU and the integrated GPU of the APU to gain further performance.
First, we propose a general hybrid strategy to spare stencil-based workloads between CPU and GPU cores within an APU. Then, we describe the changes that we had to conduct on the 3D finite difference kernel and show the performance numbers on the CPU of the APU and on the integrated GPU of the APU separately. Finally, we apply the hybrid strategy on the modified 3D finite difference kernel in order to exploit both the CPU and the integrated GPU, and we show the obtained performance numbers.

### 6.4.1 Hybrid strategy for the APU

In order to use the CPU and the GPU concurrently, we propose a hybrid strategy as to share an application workload between the APU CPU and the integrated GPU. Before developping such a strategy we recall some OpenCL pittfalls that should be considered in hybrid implementations.

The CPU can be seen as an OpenCL compute device along with the GPU. OpenCL kernels can be executed on CPU and on GPU, and synchronized through the runtime. On an AMD CPU, a thread will execute one by one each work-item of a given work-group until completion or encountering a barrier. As the work-items are executed one at time, there is no overhead on CPU due to diverging control flows among the work-items. Here, it has to be noticed that the AMD OpenCL SDK used at the time of writing (version 2.6) does not provide implicit vectorization among multiple work-items on CPU. To use the SSE instructions on the CPU, we therefore need to rely on explicit vectorization by means of vector types (`float4`). Moreover the CPU caches enable to offset the overhead of irregular memory accesses. Besides, local memory is redundant on CPU with the caches of each core, especially since a work-group is executed on a single core.

On AMD GPUs, wavefronts in which work-items execute different control flows will have their executions serialized over the different control flows. The partial SIMD execution of a GPU will force diverging control flows to be serialized (compute divergence). Furthermore, on a GPU, global memory accesses are optimal when work-items in a given wavefront access a contiguous and aligned memory area at the same time (*coalesced* memory accesses on NVIDIA GPUs). Conditional control flows would induce an overhead due to the divergence in memory accesses (memory divergence).

For finite difference stencils, as the costs of computation and memory divergence are lower on CPU, specific treatments on the domain boundaries could be executed on

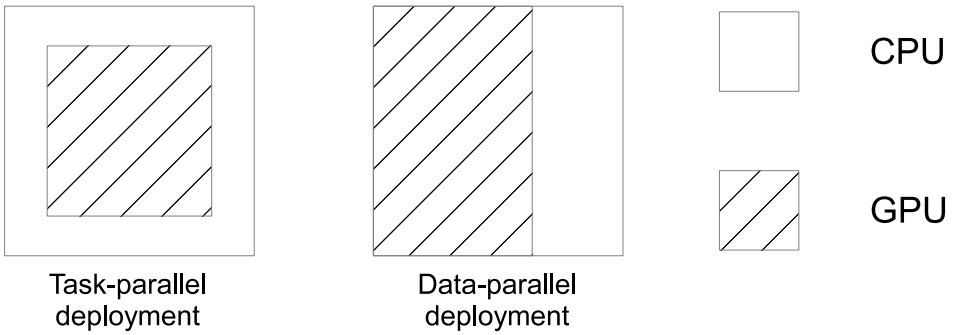



FIGURE 6.30: Example of data-parallel and task-parallel deployments on a 2D domain.

the CPU, while keeping the regular parallel execution within the domain on the GPU. For example, when simulating a wave propagation on an infinite or semi-infinite domain with stencils, the domain borders act as reflectors. To correct this, PMLs are usually used as boundary conditions (see section 2.2). We can imagine a scenario where the interior region of the domain is updated by the GPU and the PML regions by the CPU. In the rest of the chapter, we refer to splitting tasks between the CPU and the GPU as a *task-parallel* hybrid deployment. Alternatively, the CPU can simply be used for its additional compute power and memory bandwidth in a *data-parallel* deployment where we can divide the compute domain into two regions to be executed on the CPU and on the GPU, in a data-parallel fashion (see figure 6.30). To choose between these two alternatives, we propose the strategy presented in figure 6.31. First, we determine if the application is compute-bound or memory-bound. For this purpose, we compare the arithmetic intensity (ratio of the number of memory accesses over the number of arithmetic operations) and the hardware specifications of the integrated GPU. We then try to quantify the divergence, either through a profiler, or through code analysis or even thanks to programmer indications. For a memory-bound application, we only consider divergent memory accesses, whereas we look only at divergent computations in the compute-bound case. If the divergence is high enough according to empirical thresholds, we choose the task-parallel deployment. Otherwise, the data-parallel one will be preferred. Similar works can be found in [64], where the authors rely on the "computational density" metric in order to characterize whether a workload is better suited for the integrated GPUs of APUs or for discrete GPUs.
Our strategy could be implemented in a generic software platform. In this work, we will only apply and try to validate this strategy on the 3D finite difference stencil OpenCL kernel presented in section 6.2.2 on the Trinity APU (at the time of this work the Kaveri APU was not released yet).

The hybrid strategy applied to stencil computations requires to read from an input buffer and write to an output buffer, for both CPU and GPU. As the buffers will swap between input and output at every iteration, buffers need to be zero-copy and read-write enabled. $u$ memory, being read-only from the GPU, is not a valid choice, only $z$ and $p$ are both zero-copy and read-write enabled from CPU and GPU. It has been shown in section 6.2.2, that $p$ is significantly slower than $z$ for CPU reads. The input and output buffers will thus be allocated as $z$ buffers. Furthermore, it has been demonstrated that

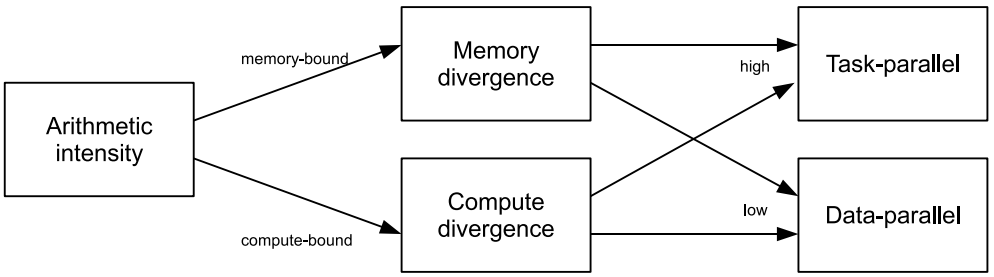



FIGURE 6.31: Hybrid strategy for the APU.

on the Trinity APU, the $z$ memory can be as interesting as the $g$ memory (see figure 6.22), only when snapshotting is performed at every iteration. Memory accesses via *Garlic* ($g$ memory) are indeed much more efficient than via *Onion* ($z$ memory) on the Trinity APU. On the Kaveri APU, we have shown in section 6.2.2 that the performance gap between $z$ and $g$ memory locations is reduced, which widens the interest of using $z$ memory and justifies our hybrid approach based on zero-copy buffers.

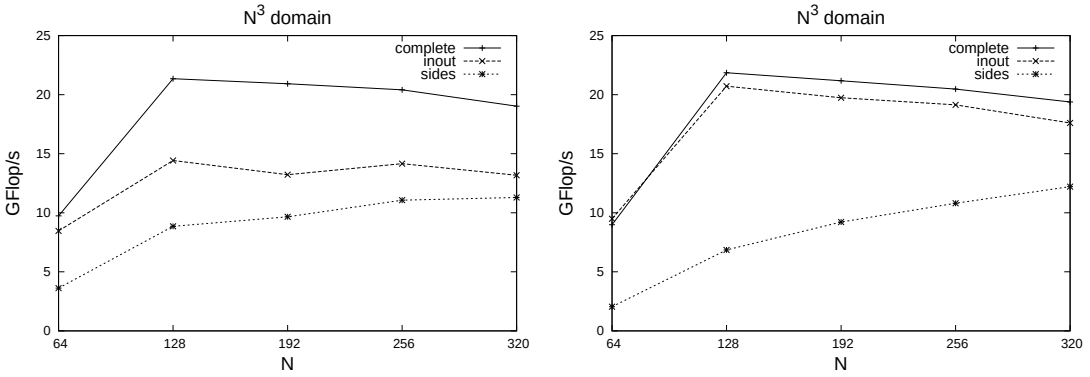
In order to study our strategy, we had to modify the 3D finite difference OpenCL kernel used in section 6.2.2. To study the effect of memory access divergence, we use here a basic stencil with a parametrizable size. This enables us to control the amount of divergence by changing its size. To ensure the best control over divergence we therefore do not use PMLs here. As opposed to the 3D finite difference stencil kernel studied in section 6.2.2, we do not use halos either (additional points on the borders of the domain containing only zero values) on the buffers, as they would cancel memory divergence. We will use a laplacian stencil (see figure 6.13a). For low divergence, we will use an $8^{th}$ order stencil, and a $64^{th}$ order stencil for high divergence.

### 6.4.2 Deployment on CPU or on integrated GPU

We first study CPU-only and GPU-only computations. We determine which ones are the most efficient in order to use them as the basis for our task-parallel and data-parallel hybrid approach. We investigate different deployment techniques following [209]:

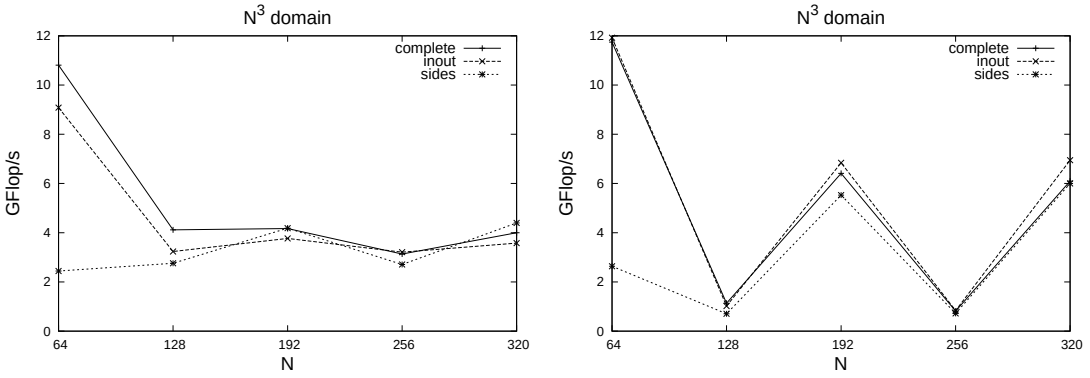
- **complete**, one single kernel executed on the 3D whole domain;
- **inout**, one kernel executed on the inside of the domain and another one encompassing all the borders;
- **sides**, one kernel executed on the inside of the domain and one for each of the six borders in 3D.

**complete** will be the basis for the data-parallel deployment, and the task-parallel deployment will be developed on the most performant between **inout** and **sides**. We consider the OpenCL kernel versions *scalar*, *vectorized* and *local vectorized* as described




(a) $8^{th}$ order stencil. (b) $64^{th}$ order stencil.

FIGURE 6.32: Performance results of stencil computations on the integrated GPU of Trinity.

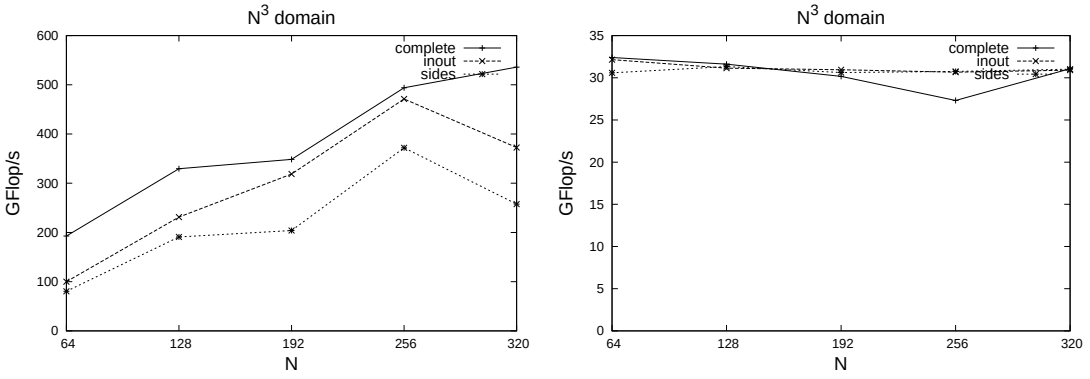



(a) $8^{th}$ order stencil. (b) $64^{th}$ order stencil.

FIGURE 6.33: Performance results of stencil computations on the CPU of Trinity.

in section 6.2.2. It has to be noticed that our kernel source codes are fully parametrizable. This enables us to define the size of the stencil and the size of the work-group at kernel compilation time by using the C pre-processor. We can then easily optimize the size of the work-groups for the Trinity APU via an exhaustive search. The domains are three-dimensional and of size $N^3$. Due to the existence of vectorized instructions on both the CPU and GPU, the *vectorized* kernel performs consistently better than the *scalar* one thanks to the SSE instructions and the GPU vector processing units (performance tests not presented). The *local vectorized* version does not offer performance gain over *vectorized* either. On the CPU, the redundancy of local memory with caches implies indeed an overhead. On the GPU, there is also no performance gain for *local vectorized* with a $8^{th}$ order stencil (note that the domain size ($N^3$) is different from that used in figure 6.19 ($N^2 \times 32$)). For a $64^{th}$ order stencil, the amount of local memory required decreases the occupancy, hence the low performance. We will therefore consider only the *vectorized* kernel through the rest of this section.

Figures 6.32a, 6.32b (resp. 6.33a and 6.33b), present performance results of our different deployments on GPU only (resp. on CPU only). This enables us to study the divergence impact on each architecture separately. On GPU, **sides** is topped by the other strategies, due to the cost of launching the multiple kernels and the low occupancy of the kernels on the borders. On a $8^{th}$ order stencil (see figure 6.32a), **complete** performs better than **inout**. When the size of the stencil is increased to 32

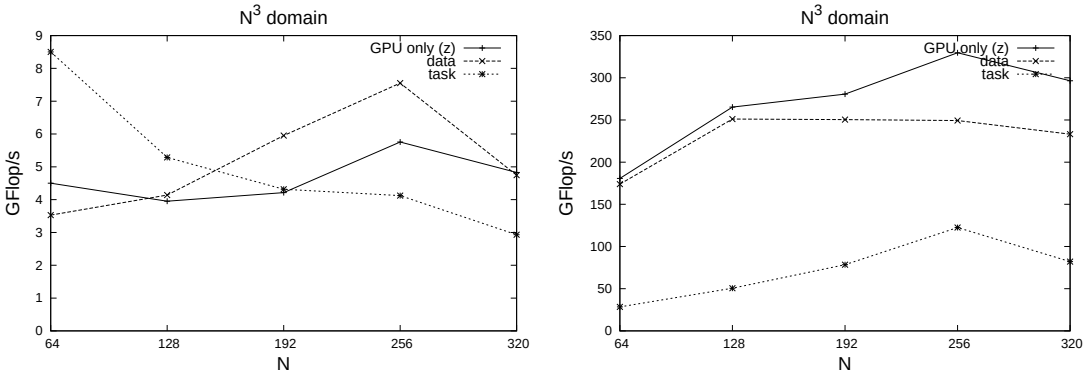



(a) On the GPU (b) On the CPU

FIGURE 6.34: Performance results of compute-bound stencil computations ($8^{th}$ order stencil).

(see figure 6.32b), memory divergence rises and **inout** almost catches up with **complete** but does not outperform it: this is due to the fact that the AMD GPUs are only slightly sensitive to memory divergence [29]. On CPU, performance drops very fast when the domain enlarges (see figures 6.33a and 6.33b), because a smaller domain, fitting almost entirely in cache, allows for a better memory bandwidth. There is nearly no difference between the performances of the three strategies, as they were designed to handle divergence and CPU hardware is not sensitive to divergence. Domains with sizes which are multiples of 128 offer low performance, probably due to the under-utilization of the interleaved memory banks. The effects are especially noticeable for the $64^{th}$ order stencil computations, which are subject to higher cache pressure.

We have observed so far the effect of memory divergence on stencil computations, but the little sensivity of AMD hardware to memory divergence has lessened its impact. To more clearly witness the impact of our strategy, we have also studied artificial compute-bound stencils with compute divergence. In this purpose, we first include compute divergence in our kernels by dividing the computation into seven distinct parts: the inside of the domain, and each of the six sides. To ensure that the compute divergence will not be made negligible by the memory-bound nature of our stencils, we artificially raise the arithmetic intensity. To do so, we compute each point of the domain several times and accumulate all the results, while keeping the values necessary for the computation in registers. In the memory-bound stencil, for each point in our domain, there were 26 memory accesses (25 reads and one write) for 36 arithmetic operations. For the compute-bound stencil, the number of memory accesses remains the same, whereas the number of arithmetic operations is multiplied by the number of iterations. The Trinity APU integrated GPU has a maximum memory bandwidth of 25.6 $GB/s$, meaning it can access 6.4 G floats per second. With a peak performance at 546 $GFlop/s$, its theoretical threshold between memory-bound and compute-bound applications is around 85 floating point operations per memory access. The arithmetic intensity of our previous basic stencil computation was 1.4, which is clearly memory-bound. With 128 iterations our new articifical stencil computation has an arithmetic intensity of 177 which is clearly compute-bound. For these compute-bound stencils, we also study our three deployment strategies and we set the $8^{th}$ order stencil. The relative performance of the three strategies are then similar to the previous ones on both the GPU and the CPU (see the figures 6.34a and 6.34b). On the GPU, *sides* is again performing worse than *inout*, *inout* being




(a) Memory-bound stencil (b) Compute-bound stencil

FIGURE 6.35: Performance results of hybrid deployments of stencil computations

closer to *complete*. On the CPU, there is no difference in the performance results of the three strategies, since the CPU is again not sensitive to divergence.

### 6.4.3 Hybrid deployment

Our two strategies for hybrid deployments, task-parallel and data-parallel, execute one or several kernels on both CPU and GPU. For the task-parallel deployment, we choose the **inout** strategy, as it consistently performs better than **sides**. We modify it to make the CPU execute a divergent kernel on the borders of the domain, while the GPU executes a regular kernel on the inside. For the data-parallel one, we divide the domain into two subdomains and execute the same kernel on them. We divide the domain along the Y axis, as cutting along the X axis could split up `float4` values, and as the Z axis is the axis work-items iterate along. We synchronize the execution of the kernels by using a blocking function, `clFinish`, to ensure that the two execution queues (CPU and GPU) are completed. According to the OpenCL specification, `clFinish` also ensures that memory writes are visible to both the CPU and GPU. As noted by [75], the OpenCL standard (version 1.2) does not specify a way to share a buffer for concurrent accesses by multiple devices (on distinct data within this buffer). However, the AMD implementation of OpenCL makes it possible by not deleting the reference to the physical location of the buffer when unmapping. In their case, simply unmapping the buffer from the device allowed them to use it simultaneously on the GPU, as a device, and on the CPU, as a host. In our case, we similarly write to the shared buffer concurrently by the CPU and the GPU, both used here as OpenCL devices, and we have consistently checked the correctness of our hybrid computations.
For the data-parallel deployment, we also need to determine the optimal ratio between the parts of the domain that will be computed by the CPU and by the GPU. At first, we had chosen the theoretical peak performance ratio of the CPU and the GPU, but the use of the profiler has shown an imbalance in the execution times. We have then chosen the ratio of the actual performance of the CPU-only and of the GPU-only vectorized (complete) kernels: these have been confirmed empirically as nearly optimal, by running several performance tests with various ratios. For the memory-bound case, we have thus a 0.8 ratio for the GPU, and 0.95 in the compute-bound case.
According to the AMD profiler, executing a kernel on the CPU used as a device preempts the CPU and hence prevents the CPU host from enqueuing GPU kernel executions. A

possible solution would have been the OpenCL device fission of the CPU, separating it into different devices, keeping one core for the host and executing the kernel on the remaining three. In our case, enqueuing the GPU kernel first was adequate, as we have to synchronize at each iteration and cannot enqueue more than one GPU kernel at a time.

We now compare the APU hybrid performance (for $8^{th}$ order stencils) with an execution on the APU integrated GPU only with buffers allocated in $z$ memory. For the memory-bound case (see figure 6.35a), we see a better performance in the task-parallel deployment on smaller domains, but this drops when the domain gets larger and the GPU occupancy rises. When the GPU is full enough, the CPU will not perform fast enough to keep up, and overall performance will decrease. The data-parallel deployment performs better when the domain gets larger, as the relative cost of synchronization gets lower when computation time increases. We obtain a 20% to 30% better performance compared to GPU only (with buffers in $z$ memory).
For the compute-bound case (see figure 6.35b), the task-parallel deployment is slowed down by the CPU's lower compute power. Even for the data-parallel deployment, the cost of synchronization is too high due to the imbalance in the ratio between the CPU and the GPU. This may be due to the lack of optimization of the source code for the CPU: whereas its compute power is 18% of the APU, its kernel performance is only 5% of that of the integrated GPU.
Finally, it has to be noticed that the GPU-only performance (with buffers in $z$ memory) is here lower than the GPU-only performance (with buffers in $g$ memory) presented in figures 6.32a and 6.34a. $g$ memory offers indeed a better bandwidth than $z$ memory on the Trinity APU. However, those previous results with $g$ memory do not take into account the cost of the snapshotting necessary to applications such as the RTM. In section 6.1, $g$ memory has been shown to perform currently better than $z$ memory even when the snapshotting frequency is high, but we already showed that with the Kaveri APU the bandwidth of such zero-copy ($z$) memory has been significantly improved.

In this section, we studied how to use both the CPU and the GPU of the APU for stencil computations. We provided a strategy for hybrid deployments of stencil computations that takes into account the compute-bound or memory-bound nature of the application and its amount of divergence. We proposed two possible deployments, a task-parallel one and a data-parallel one, and balanced the use of the CPU and the integrated GPU in order to exploit at best the APU. When considering a classic memory-bound stencil, we obtained an up to 30% performance gain compared to a GPU only deployment (with buffers in $z$ memory). Given that the theoretical single precision of the CPU of the Trinity APU represents 18% of the overall theoretical performance of the APU, one can infer that not only the hybrid approach has improved the performance of the stencil computations by using more resources, but also it accelerated the GPU computation since the branch divergence is reduced.
However, our strategy seems to be valid in the memory-bound case with a high rate of memory and compute divergence, but requires further investigation for the compute-bound case on the APU. More precisely, the OpenCL kernel for the CPU should better exploit the CPU caches as well as the CPU *prefetch* feature, and hence offer much better performance. In the future, we first could try to confirm the validity of our strategy on real world applications such as the RTM. We could also plan to apply this strategy on other hybrid hardware, such as the Intel multicore CPUs with integrated GPUs or the Tegra X1 GPU from NVIDIA with both GPU and ARM CPU on the same chip (such future work is not covered by the scope of this thesis).

To conclude, hybrid computations on APU can be reliable under several assumptions: a high rate of memory and computation divergence is required along with a highly optimized CPU code. Given the low compute power of the CPUs of the actual AMD APUs, and given that the finite difference stencil, that we use in the seismic applications, is of a low order ($8^{th}$ order) and is not subject to a high rate of divergence, we decided to dedicate the CPU resources of the APU rather to auxiliary tasks, other than computations, such as I/O operations and MPI communications in the case of the multi-node implementations of the seismic applications (namely for the RTM).

## 6.5 Directive based programming on the APU: finite difference stencil as an example

This section corresponds to a Master 1 internship conducted by the student R. Dang. We evaluate the performance and the programmability of the 3D finite difference benchmark using a directive-based approach. An OpenACC 3D finite difference kernel is proposed, where a list of implementations and optimization techniques are detailed. The implementations include three versions with a gradual increase in terms of programming complexity. The first is a plain OpenACC implementation, the second one is an implementation based on both OpenACC and the HMPPcg extension in order to generate optimized OpenCL kernels, and the third one makes use of OpenACC, HMMPcg and modifies the initial code in order to exploits the GPU local memory and increase the ILP factor. Besides, a performance comparison between the OpenCL performance numbers, more specifically those of the *local scalar* version as we only survey the GCN based GPUs, and those of the 3D finite difference OpenACC implementations on the Kaveri APU and on the Tahiti discrete GPU is presented.

### 6.5.1 OpenACC implementation details

We start this section with a quick description of our OpenACC implementation of the 3D finite difference stencil application. We also describe our optimization techniques while giving details about the OpenACC programming paradigm. For a thorough understanding we refer to the OpenACC literature that can be found in [13]. For this evaluation, we make use of the CAPS Compiler (version 3.3), provided by *CAPS Entreprise*, to generate OpenCL kernels from an initial implementation in *C* code of the 3D finite difference stencil application.

To begin with, the straightforward implementation of the 3D stencil kernel consists in delimiting the main loop nest, presented in the figure 6.36, with a "kernels" OpenACC construct as shown in figure 6.37a. The "parallel" OpenACC directive can be used as well in order to map the loop to an accelerator (see figure 6.37b). In the figure 6.37, the construct "gang" refers to a group of threads that operates over a number of processing elements as with workgroups in OpenCL. The "kernels" construct indicates that a region in the code may contain parallelism, and the compiler identifies the loop nests that can be converted into parallel kernels, in a CUDA or OpenCL fashion, that run on GPU. However, with the "parallel" construct the programmer explicitly identifies one loop nest as having parallelism, in an OpenMP fashion, and the compiler generates a parallel kernel based on the body of the construct. We chose to use the "kernels" construct.

```
for(z = 0; z < dim[2] ; ++z) {
   for(y = 0; y < dim[1]; ++y) {
      for(x = 0; x < dim[0]; ++x) {
         update_cell(uo, ui, x, y, z);
         ...
      }
   }
}
```

FIGURE 6.36: The main loop nest in the 3D finite difference stencil code. $x$, $y$ and $z$ are the grid indexes along the $X$, $Y$ and $Z$ axises respectively. *uo* is the 3D domain to be updated (inside the *update_cell*() routine) based on the inputs in *ui*.

```
#pragma acc kernels {
#pragma acc loop gang
for(y = 0; y < dim[1]; ++y) {
    #pragma acc loop gang
    for(x = 0; x < dim[0]; ++x) {
         for(z = 0; z < dim[2] ; ++z) {
             update_cell(uo, ui, x, y, z);
             ...
         }
    }
}
}
```

(a) Using the *kernels* construct.

```
#pragma acc parallel {
#pragma acc loop gang
for(y = 0; y < dim[1]; ++y) {
   #pragma acc loop gang
   for(x = 0; x < dim[0]; ++x) {
      for(z = 0; z < dim[2] ; ++z) {
         update_cell(uo, ui, x, y, z);
         ...
      }
   }
}
}
```

(b) Using the *parallel* construct.

FIGURE 6.37: The initial OpenACC implementation of the 3D finite difference stencil main loop nest.

Note that the order of the loop over $X$, over $Y$ and that over $Z$ are reorganized. This allows each work-item to sweep over the $Z$ axis during computation as it is the case in the OpenCL implementation of the 3D finite difference stencil, detailed in section 6.2.2. Furthermore, in order to enhance the memory allocation and transfers between the host memory and the GPU memory, namely when performing data snapshotting, we added a user managed OpenACC "data" region (using the "data" construct). As a matter of fact, memory buffers are automatically allocated and then copied to the accelerator memory prior to any kernel execution by the OpenACC runtime. They are released at the end of the kernel execution. In our case we would like to keep the memory objects persistent in the GPU memory throughout the kernel executions of successive iterations. Thanks to the "data" clause the memory arrays are created only once and before the first iteration, and released after the last one. The updated OpenACC pseudo-code is partially presented in figure 6.38. Note that the data snapshotting is performed using the "update" directive. The "create" directive is used to allocate copies of the memory objects on the GPU memory when the data region is reached for the first time, the "present" data clause means that the memory objects are already present in the GPU memory which prevent creating them every time they are accessed, and the "delete" clause indicates that the memory objects on the GPU memory have to be released. Besides, in order to efficiently gridify the 3D finite difference stencil main loop nest, i.e. mapping the loop nest to grid of GPU threads, and determine the appropriate workgroup size, we made use of the "gang", "worker" and "vector" directives. The CAPS compiler translates the gang construct into the total number of workgroups, the vector construct into the size of the first dimension and the worker construct into the size of the second dimension of a workgroup. It is to be noted that this may change from an OpenACC compiler to another as the OpenACC standard does not give strict recommendations on how these directives should be interpreted. The figure 6.39 shows an example of the gridification of the 3D finite difference main loop nest into 400 workgroups, each

being of dimension $32 \times 8$. In the figure, the "independent" construct specifies that the loop iterations are data-independent and can be executed in parallel, overriding compiler dependence analysis. Note that similarly to the OpenCL implementations, the best workgroup dimensions are determined by auto-tuning.

The next step is to use the HMPPcg extension in order to have much more control on how the OpenCL kernels should be generated. To be able to use the HMMPcg clauses, one should first use the gridification mechanism provided by HMMPcg instead of that provided by the OpenACC standard. This can be done using the "gridify" construct which we found very useful to express 2D gridifications (as opposed to OpenACC which is not straightforward). The HMPPcg gridification parameters should be defined at compile time with the help of the options "-Xhmppcg -grid-block-size,SIZEXxSIZEY" where $SIZEX$ is the first dimension of a workgroup and $SIZEY$ is the second. An example of an implementation using the HMPPcg directives is presented in figure 6.40. We relied on the HMPPcg directive "unroll" which is intended to unroll the the inner-most loop, and thus increase the register exploitation and decrease the impact of the conditional test at the end of each iteration of the loop. With this technique, we aim to improve the performance. Note that in this version we keep using the OpenACC data clauses in order to handle the memory transfers.

```
#pragma acc enter data create(ui[0:size], uo[0:size])
...
for(it = 0; it < N; ++it) {
    #pragma acc kernels present(ui[0:size], uo[0:size]) {
    #pragma acc loop gang
    for(y = 0; y < dim[1]; ++y) {
        #pragma acc loop gang
        for(x = 0; x < dim[0]; ++x) {
             for(z = 0; z < dim[2] ; ++z) {
                 update_cell(uo, ui, x, y, z);
                 ...
             }
        }
    }
    }
    ...
    #pragma acc update host(uo[0:size])
    ...
}
#pragma acc exit data delete(ui[0:size], uo[0:size])
```

FIGURE 6.38: The modified OpenACC implementation of the 3D finite difference stencil main loop nest in order to enhance memory transfers between the host and the GPU memory. *size* is the size of the domain, *it* is the iteration index and $N$ is the number of iterations.

```
#pragma acc kernels {
#pragma acc loop independent, gang(400), worker(8)
for(y = 0; y < dim[1]; ++y) {
    #pragma acc loop independent, vector(32)
    for(x = 0; x < dim[0]; ++x) {
         for(z = 0; z < dim[2] ; ++z) {
             update_cell(uo, ui, x, y, z);
             ...
         }
    }
}
}
```

FIGURE 6.39: The gridification of the 3D finite difference stencil main loop nest using OpenACC constructs only.

Finally, in order to try to apply the same performance optimizations that we adopted in the OpenCL implementations we had to alter the initial code. First, we tried to make use of the local memory. For this purpose, the OpenACC standard provides the directive "cache". Unfortunately, after an early experiment the CAPS compiler seemed to ignore this construct. As an alternative, we relied on the HMPPcg directive "shared". Introducing the "shared" clause required the modification of the initial code. Some HMMPcg intrinsics, such as $RankInBlockX()$ and $RankInBlockY()$ should be added in order to distinguish the threads local coordinates to workgroups from the threads global coordinates. Besides, we had to explicitly add the code that copies data from the global memory to the local memory to the initial serial code. Thus, it goes without saying that this version affected the initial code and required the modifications of the latter in order to reproduce the same OpenCL *local scalar* kernel discussed in section 6.2.2. As a matter of fact we almost had to re-write the initial code. We present a code snippet of this implementation in figure 6.41. Besides, we also tried to increase the ILP. For that to do, we had to modify the initial code by breaking the innermost loop into two loops, having the second one sweeping over a temporary array whose size defines the ILP parameter, and which contains the intermediate results of the computation performed on multiple grid points at the same time.

To summarize, we started with a straightforward implementation where few OpenACC directives are added to the CPU code in order to express parallelism and map

```
#pragma hmppcg gridify(y,x),
for(y = 0; y < dim[1]; ++y) {
    for(x = 0; x < dim[0]; ++x) {
         #pragma hmppcg unroll(8)
         for(z = 0; z < dim[2] ; ++z) {
             update_cell(uo, ui, x, y, z);
             ...
         }
    }
}
```

FIGURE 6.40: The gridification of the 3D finite difference stencil main loop nest using HMPPcg.

```
#pragma hmppcg gridify(y,x), shared(buffer), private(id_x, id_y)
for(y=0; y<dim[1]; y++){
    for(x=0; x<dim[0]; x++){
        #pragma hmppcg unroll(8)
        #pragma hmppcg set id_x = RankInBlockX()
        #pragma hmppcg set id_y = RankInBlockY()
        for(z=0; z < dim[2]; z++){
            if(id_x < 4)
                L(buffer, id_x-4, id_y) = O(i, z, y, x-4);
            if(id_x >= size_bloc_x - 4)
                L(buffer, id_x+4, id_y) = O(i, z, y, x+4);
            if(id_y < 4)
                L(buffer, id_x, id_y-4) = O(i, z, y-4, x);
            if(id_y >= size_bloc_y - 4)
                L(buffer, id_x, id_y+4) = O(i, z, y+4, x);
            L(buffer, id_x, id_y) = O(i, z, y, x);
            #pragma hmppcg barrier
            ....
            update_cell(uo, ui, x, y, z)
        }
    }
}
```

FIGURE 6.41: The implementation of the 3D finite difference main loop nest using HMPPcg and modifying the initial code.

| | Kaveri ($GFlop/s$) | Tahiti ($GFlop/s$) |
|---|---|---|
| OpenACC only | 32.28 | 213.52 |
| OpenACC + HMPPcg | 42.13 | 254.55 |
| OpenACC + HMPPcg + code modification | 38.73 | 247.73 |

TABLE 6.3: Performance comparison of the different OpenACC implementations of the 3D finite difference application on Kaveri and Tahiti. The numbers correspond to the average performance of 100 iterations of stencil computations.

computations to the GPU. Then we presented a list of optimization techniques, with a gradual increase in terms of complexity. Some implementations have affected the initial code, which became very verbose to some extent, but they are meant to provide performance enhancement.

### 6.5.2 OpenACC performance numbers and comparison with OpenCL

In this section we evaluate the performance of the different OpenACC implementations of the 3D finite difference stencil application detailed in section 6.5.1. We compare the OpenACC performances with respect to the OpenCL performance numbers obtained in section 6.2.2. We chose to consider a domain size of $1024 \times 1024 \times 32$ as a use case for our evaluation. The performance numbers on Tahiti are obtained using the AMD Catalyst driver 14.4 and those on Kaveri are obtained using the AMD Catalyst 15.4. Each OpenACC implementation is auto-tuned in order to determine the best gridification parameters as well as the most appropriate loop unrolling level in order to deliver the best performance possible. Besides, given that we only survey the GCN based GPUs, the OpenCL vectorized implementations are not considered in this performance comparison. In table 6.3, we summarize the performance numbers of the three OpenACC implementations, detailed in section 6.5.1 on both the Tahiti discrete GPU and the Kaveri APU (integrated GPU only). The complexity, in terms of programmability, of the OpenACC implementations increases from top to bottom in table 6.3. We notice that the evolution of the performance on both Kaveri and Tahiti are quite similar. The OpenACC implementation coupled with the HMPPcg extension offers the best performance on both the APU and the discrete GPU. The modification of the initial code by adding the local memory did not provide any performance enhancement. Note that trying to increase the ILP didn't help either and provided even lower performance numbers than those presented in table 6.3. Not to mention that it almost required to rewrite the initial code (see figure 6.41).
In order to evaluate the performance of the OpenACC implementations on both Kaveri and Tahiti, we compare it against the performance of the best OpenCL implementations on both the APU and the discrete GPU, which is given by the *local scalar* version. We can see in table 6.4 that the OpenACC + HMPPcg implementation delivers approximately half the performance of the best OpenCL implementation on both Tahiti and Kaveri. As a matter of fact, the OpenCL implementation makes use of the local memory and includes other optimizations such as using the register blocking and increasing the ILP, while we did not manage to benefit from the local memory in the OpenACC implementation (see table 6.3). Besides, we were unable to reproduce the register blocking and neither were we able to take advantage from increasing the ILP.

To sum up, we have proposed three directive-based implementations of the finite difference stencil kernel, with an increasing level of programming complexity. We have

| | Kaveri ($GFlop/s$) | Tahiti ($GFlop/s$) |
|---|---|---|
| OpenACC | 42.13 | 254.55 |
| OpenCL | 85.06 | 544.09 |

TABLE 6.4: Performance comparison between the OpenACC best implementation and the best scalar OpenCL implementation of the 3D finite difference application on Kaveri and Tahiti. The numbers correspond to the average performance of 100 iterations of stencil computations.

evaluated them on the Kaveri APU and on the Tahiti discrete GPU. We have shown that the OpenACC standard does not provide enough directives to optimize the generated OpenCL kernels and to expose the hardware features to the programmer (for example to explicitly use the local memory). Consequently, the HMPPcg directives (we recall that HMPPcg is not a standard), have to be used in conjunction with OpenACC in order to reach the best performance. Besides, modifying the initial code in order to use the local memory with OpenACC and to increase the ILP did not provide any performance enhancement compared to the OpenACC + HMMPcg implementation, although it required a tremendous programming effort. Finally, we noticed that the OpenACC standard did not restrict enough how the directives should be interpreted, especially when defining the gridification parameters (with "gang", "worker" and "vector" directives), which would impact the performance portability of the GPU applications.

As a conclusion, given that the hardware accelerators programmability, that of APUs in particular, is driven by the preferred trade-off between performance enhancements and programming efforts, the obtained OpenACC performance numbers are encouraging thanks to the huge difference in terms of efforts compared to OpenCL (almost 15× less lines of code). However, given that the best floating point performance is our main concern when deploying seismic applications on APUs, we choose to rely on the OpenCL programming model in the rest of our study.

# Chapter 7

# Seismic applications on one compute node

**Contents**



In this chapter we focus on real world seismic applications that we study in the scope of this work, namely seismic modeling and seismic migration. We implement seismic modeling and seismic migration applications on CPU, on APU and on discrete GPU. A special emphasis is placed on optimizing the seismic application at the node level on the APU and on the discrete GPU using OpenCL, while the CPU implementation (also based on OpenCL) is more used for the sake of comparison.
First, we describe the algorithm of the seismic modeling and give implementation and optimization details on APUs and on GPUs. Besides, we elaborate a comparison at the node level between the different architectures (an *AMD Phenom TM II x6 1055t Processor*, a Tahiti GPU and a Kaveri APU). The comparison includes an OpenCL performance evaluation and a power efficiency study. Moreover, we implement the seismic modeling using a directive based approach with OpenACC and we compare its performance against that achieved with OpenCL. Based on that comparison, we assess the two programming models in terms of ease of programmability.
Second, we use the seismic modeling core algorithm to build the forward modeling and the backward modeling stages of the seismic migration, more specifically of the RTM (see section 2.2.2). Similarly to the seismic modeling application, we evaluate the performance and the power efficiency of the RTM at the node level and compare the results

of the different architectures. In addition, we compare the performance delivered by OpenACC against that obtained using OpenCL, with an emphasis on the programming efforts required by the two programming models.

## 7.1 Seismic modeling

In this section we describe the different steps of the seismic modeling algorithm. We only focus on the heart of the application, which is the wave propagation, that will be further used by the RTM application. Besides, we give a succinct overview of the OpenCL implementations on the surveyed hardware, and evaluate their performance results and their power efficiency. Finally, we also compare a proof-of-concept OpenACC implementation of the seismic modeling with the OpenCL implementations in terms of performance and programmability.

### 7.1.1 Description of the algorithm

The seismic modeling, or the forward modeling as presented in section 2.2, is a numerical simulation of a propagation of a source wavefield on a given medium (in our case the medium is isotropic). The propagation is governed by a wave equation solver and by the subsurface reflectively, often emulated by synthetic velocity models. By seismic modeling we actually approximate the solution of the wave equation (2.16), using the Laplace operator to compute the spatial derivatives and the Leap-Frog scheme to compute the derivatives with respect to time. We present in algorithm 7.1 a high level description of the seismic modeling algorithm. The computation involves two arrays of the wavefield $u0$ and $u1$ that are interleaved throughout the simulation iterations in order to compute the temporal derivatives. The source term of the equation (2.16) is introduced in the subroutine *add_seismic_source*(). We make use of the Ricker wavelet, illustrated in the figure 5.1, as a seismic source. The wave propagation is performed in the subroutine *update_wavefield*() where the stencil computations, described in section 6.2.2, are used. Figure 7.1 shows the structured grid, $\Omega_{16\times16\times16}$ (see section 5.3.2.2) used to compute the discretized wavefields on each time iteration. The gray grid is augmented with additional layers (the light blue layers) on the bottom and on the sides. These layers are artificial and are used for the perfectly matched layer (PML) boundary conditions applied to the wave equation solver. The grid also features a *free surface* or a *zero-stress* boundary condition (the purple layer) on the top. This is to simulate the flat free surface of the earth or sea in which the seismic energy is not reflected nor refracted. The gray region is updated using the subroutine *compute_core*() and the blue one is updated using *compute_pml*(). Usually, the source wavefield is then periodically stored on disk (subroutine *save_seismic_trace*()), with respect to a snapshotting frequency, in order to construct the seismic traces that may be further used for seismic migration or full wave inversion. However, in this section we only focus on the core algorithm of the seismic modeling, i.e. the *update_wavefield*() subroutine, as it is the building block of the RTM which is our main interest in this chapter. As a consequence, our evaluation of the seismic modeling does not involve data snapshotting (no I/O operations). In other words, the subroutine *update_wavefield*() is not considered in this section, we rather present it to give the reader a complete picture of the seismic modeling algorithm. We consider that the medium is isotropic and we make use of the *3D SEG/EAGE salt model* presented in figure 5.2 as a velocity model.

### 7.1.2 Accelerating the seismic modeling using OpenCL

When it comes to the OpenCL implementations, for the seismic modeling application we only consider the *scalar* version and the *local scalar* version, as described in section 6.2.2, since we target only GCN micro-architecture based AMD GPUs. However, we still need the *vectorized* version for the CPU implementation in order to benefit from the explicit vectorization (see figure 6.14). We apply a 2D work-item grid on the 3D domain and we rely on the same optimization techniques applied to the 3D finite difference stencil kernel. The seismic modeling kernel involves more memory accesses, compared to the stencil kernel, as more memory objects are used namely for the source term, the velocity model and for arrays related to the computation of the PML regions. Besides, updating the wavefield on the PML regions requires more floating point operations (higher compute intensity compared to the core regions). In addition, as shown in algorithm 7.1, the seismic modeling kernel is subject to a conditional statement in order to evaluate whether a grid point is situated in a PML region or not, which incurs two different computations accordingly. Depending on the width of each PML region (18 grid points for $\Omega_{16\times16\times16}$), this may cause more or less GPU divergence in the OpenCL implementations. Moreover, we implemented the subroutine *add_seismic_source*() in OpenCL, with the help of OpenCL tasks, because only few grid points (8 grid points) are concerned with this computation.
We considered running 1000 time iterations of the seismic modeling OpenCL kernel on

**Algorithm 7.1** High level description of the one-node seismic modeling algorithm.

```
1:  for t ∈ [0..T] do                          ▷ [0..T] is the simulation time-step interval
2:      if mod(t, 2) == 0 then                 ▷ t is the time-step index
3:          add_seismic_source(u0, t)          ▷ u0 is the wavefield array at time-step t
4:          update_wavefield(u1, u0, t)        ▷ u1 is the wavefield array at time-step t − 1
5:          save_seismic_trace(u1, t)          ▷ this involves I/O operations (not used in here)
6:      else
7:          add_seismic_source(u1, t)
8:          update_wavefield(u0, u1, t)        ▷ u0 and u1 are interleaved
9:          save_seismic_trace(u0, t)
10:     end if
11: end for
12:
13: procedure update_wavefield(u, v, t)
14:     for z ∈ [0..nz − 1] do                 ▷ z is the array coordinate in Z
15:         for y ∈ [0..ny − 1] do             ▷ y is the array coordinate in Y
16:             for x ∈ [0..nx − 1] do         ▷ x is the array coordinate in X
17:                 laplacian = fd_stencil_compute(v, x, y, z, t)   ▷ see algorithm 6.1
18:                 if (x, y, z) ∈ PML region then ▷ propagate the wave in PML regions
19:                     compute_pml(u, x, y, z, laplacian, t)
20:                 else                       ▷ propagate the wave in core regions
21:                     compute_core(u, x, y, z, laplacian, t)
22:                 end if
23:             end for
24:         end for
25:     end for
26: end procedure
```

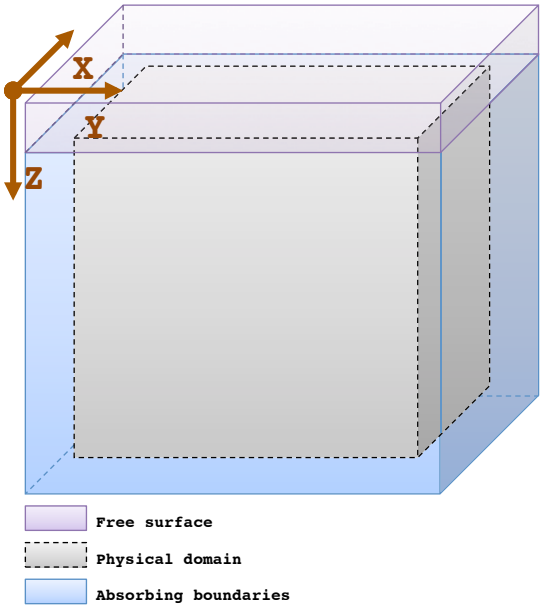



FIGURE 7.1: $\Omega_{16\times16\times16}$ : the computational domain of the problem under study.

the *AMD Phenom TM II x6 1055t Processor* CPU, the Kaveri APU and on the Tahiti GPU. We used the AMD OpenCL driver version 15.4 on both Kaveri and Tahiti, and we vary the different parameters such as the workgroup dimensions and the ILP in order to perform auto-tuning and find the appropriate combination that delivers the best performance on each architecture. Besides, our first OpenCL implementations use one kernel as to implement the seismic modeling application. We also considered a different OpenCL approach where we deploy multiple OpenCL kernels, one to update the gray area of the compute grid 7.1 and one kernel to update each PML layer, which makes six OpenCL kernels in total. This approach may reduce the impact of the computation divergence that may occur within the OpenCL workgroups deployed when the seismic modeling application is implemented in one OpenCL kernel.

The table 7.1 summarizes the performance numbers of the two different OpenCL approaches (one OpenCL kernel or multiple OpenCL kernels) on the Kaveri APU. Note that the performance numbers (GFlop/s) are mainly calculated based on the theoretical number of floating point operations issued when solving the wave equation (given that

| Kernel version | #Kernels | DPS | LX | LY | ILP | comp-only ($GFlop/s$) |
|---|---|---|---|---|---|---|
| scalar | one | cggc | 32 | 8 | 1 | 28.41 |
| local scalar | one | cggc | 32 | 8 | 1 | 26.11 |
| scalar | one | zz | 128 | 2 | 2 | 12.26 |
| local scalar | one | zz | 256 | 1 | 2 | 14.09 |
| **scalar** | **multiple** | **cggc** | **64** | **1** | **2** | **32.03** |
| local scalar | multiple | cggc | 32 | 2 | 1 | 26.14 |
| scalar | multiple | zz | 128 | 1 | 8 | 13.14 |
| **local scalar** | **multiple** | **zz** | **64** | **1** | **2** | **15.57** |

TABLE 7.1: Performance parameters and results of the **seismic modeling** OpenCL kernels on **Kaveri**. The numbers marked in **bold** are the best achieved performance numbers for each configuration.

other minor operations, such as those related to adding the seismic terms to the equation, are very minority). Consequently the number of flops required to update one grid point in the core region corresponds to $4+3*(3*p/2+1)$, $p=8$ being the stencil order (see equation (2.27)) and to $26+3*(3*p/2+1)$ if the grid point is in a PML region (see equation (2.32)). For each approach we present the performance numbers of the *scalar* and the *local scalar* versions both coupled with two different data placement strategies (DPS) `cggc` and `zz`. We recall that `zz` involves zero-copy memory objects (see section 6.1) which would allow to process bigger compute domains on APUs. Besides, the zero-copy memory would be beneficial if I/O operations are considered in the algorithm. However, this is not the case in our implementations as we only focus on the computations related to the wave propagation, i.e. the OpenCL kernels performance. But, the I/O operations (via data snapshotting) will be considered in our implementations of RTM in the next section. Thus, it is expected that the seismic modeling implementations based on the `zz` DPS would be less efficient on APUs, than those relying on `cggc`. In addition to the performance numbers, the best auto-tuning parameters combination is also given in the table 7.1. The column LX indicates the size of one OpenCL workgroup along the $X$ axis, the column LY indicates the size of one OpenCL workgroup along the $Y$ axis and the ILP column shows the level of instruction parallelism used in each case. We present the performance numbers of the computation of the wave simulation, i.e the performance of the subroutine $update\_wavefield()$ (the *comp-only* column in the table). First, the table shows that as opposed to the 3D finite difference stencil OpenCL kernel, the *scalar* version slightly outperforms, by 8%, the *local scalar* version when the `cggc` data placement strategy is used. This may be induced by the fact that the GPU local memory is only used to copy the wavefield arrays ($u0$ and $u1$) from the GPU global memory, while the other arrays, which are the velocity model array, the source array and the PML arrays (2 arrays), are accessed directly in the global memory. However, when the `zz` strategy is applied the *local scalar* version is more efficient as it is the case for the 3D finite difference stencil kernel. This is most probably due to the high latency of the $z$ memory as it is accessed by the GPU cores within the system memory via the Onion bus. This latency is alleviated when the wavefield arrays are copied to the local memory, which is closer to the GPU cores and thus has a much lower latency than the $z$ memory. Consequently, the wavefield updates are faster in the *local scalar* version.
Second, we notice that using multiple OpenCL kernels offers up to 12% of performance enhancement. As a matter of fact, processing the core regions is almost 3× faster than processing the PML regions and delivers an OpenCL performance that is comparable to that of the 3D finite difference stencil kernel. Besides, updating the PML arrays separately may reduce the GPU divergence which eventually enhances the multiple OpenCL kernels implementations. More importantly, the multiple OpenCL kernels approach is expected to be more efficient in the multi-node implementation since not all the six PML faces are required to be locally updated by each compute node.
Finally, using the zero-copy memory objects cut the performance almost to half. Indeed, the seismic modeling OpenCL kernels are memory bound and therefore their performance is mainly driven by the memory bandwidth. Given that the $z$ memory bandwidth is almost 1.6× lower than that of the $g$ memory (see section 6.1), it comes as no surprise that the ratio between the performance obtained using the `zz` DPS and the one with the `cggc` DPS ranges between 0.5 and 0.6.

Similarly, we present in table 7.2 the performance numbers of the seismic modeling application on the Tahiti GPU. Here again we notice that the *scalar* version delivers the best performance (the *scalar* implementation is 12% more efficient than the *local*

| Kernel version | #Kernels | LX | LY | ILP | comp-only ($GFlop/s$) |
|---|---|---|---|---|---|
| scalar | one | 64 | 4 | 2 | 140.53 |
| local scalar | one | 64 | 1 | 1 | 124.76 |
| **scalar** | **multiple** | **64** | **2** | **2** | **144.02** |
| local scalar | multiple | 256 | 1 | 1 | 117.10 |

TABLE 7.2: Performance parameters and results of the **seismic modeling** OpenCL kernels on **Tahiti**. The numbers marked in **bold** are the best achieved performance numbers.

*scalar*). Besides, the multiple kernel approach offers an enhancement of almost 4%, only for the *scalar* version.

As a conclusion, we have shown that the multiple OpenCL kernels can slightly enhance the performance at the node level (only with the *scalar* implementation for the discrete GPUs, and with both the *scalar* and *local scalar* implementations, coupled with the two data placement strategies: `cggc` and `zz`, for APUs). Therefore, the *scalar* version with the multiple OpenCL kernels is to be used for the multi-node implementation of the seismic modeling application (see chapter 8).

We present in figure 7.2, a performance comparison of the seismic modeling OpenCL kernel on the CPU, on the APU and on the discrete GPU based on the best numbers obtained on each architecture. When it comes to the performance of APU, we distinguish the case where the `cggc` DPS is used from that with the `zz` DPS. The Phenom CPU performance numbers, presented in the figure, are obtained after deploying the OpenCL kernel on the CPU (with explicit vectorization and without using the local memory). Note that we also deployed a flat MPI version on the CPU and achieved similar performance numbers as the OpenCL version. For the multi-node CPU implementation, we will rely on the flat MPI version only (see chapter 8). We recall that the

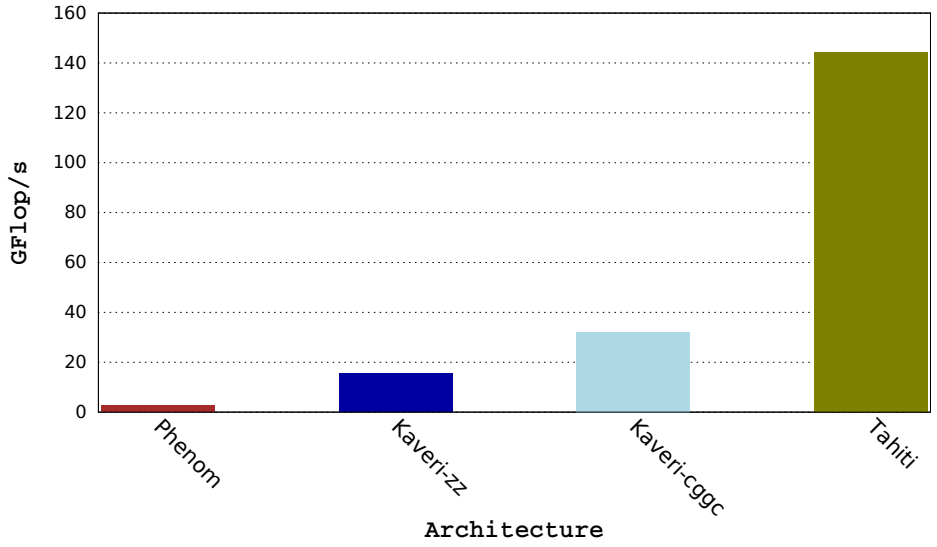



FIGURE 7.2: **Comparison** between the performance numbers of the **seismic modeling OpenCL** kernels obtained on the **Phenom CPU**, on the **Kaveri APU** and on the **Tahiti discrete GPU**. On the APU two data placement strategies are considered: **cggc** and **zz**.

performance results are obtained when considering only the computations related to the wave propagation. Therefore, while running the seismic modeling OpenCL kernels we did not consider data snapshotting (no I/O operations), nor did we consider memory transfers from CPU memory to GPU memory, i.e. PCI transfers for discrete GPU or explicit data copies from the `c` memory to the `g` memory for APUs.
The figure shows that the CPU performance numbers are small (roughly equal to 2.5 $GFlop/s$) which is almost 57× (resp. 13×) slower than the Tahiti GPU (resp. than the Kaveri GPU). Moreover, the Tahiti discrete GPU, which we recall is theoretically an order of magnitude more compute powerful than the Kaveri APU, performance results are 4.5× higher than the performance of the APU obtained when using the `cggc` DPS. Since we have mentioned that the `zz` DPS reduce the performance of the APU to half as compared to the `cggc` DPS (it is also illustrated in the figure, see Kaveri-cggc and Kaveri-zz), the Tahiti discrete GPU is 9.2× faster than the Kaveri APU when using the zero-copy memory objects.

### 7.1.3 Performance and power efficiency

In this section we add the power consumption factor to our performance study. We measured, following the methodology presented in section 6.3, the power consumption of the seismic modeling application on the different architectures and deduced the performance per Watt based on the OpenCL performance numbers presented in tables 7.1 and 7.2. The figure 7.3, shows a comparison between the power efficiency of the Phenom CPU, the Kaveri APU (while considering the two data placement strategies: `cggc` and `zz`) and the Tahiti GPU (for the discrete GPU the measured power includes the power consumption of the main CPU that is managing the GPU). Besides, the numbers on top of the histograms represent the measured power consumptions in Watts of each system. One can notice that given the low performance of the CPU and that it consumes about 130 W, the power efficiency of the seismic modeling on the Phenom CPU is very low. Besides, we notice that the choice of the data placement strategy on APUs does not impact the power consumption (we have measured 60 W of power consumption for both the `cggc` and `zz` strategies). Therefore, the ratio between the power efficiency of the seismic modeling on the Kaveri APU with the `cggc` DPS, and that using the `zz` DPS is the same ratio observed between the performances of the two strategies. Finally, the Tahiti discrete GPU offers a higher power efficiency compared to the Kaveri APU. This is a natural conclusion given that we only consider the performance of the OpenCL kernels and that the discrete GPU outperforms the APU by a factor of 4.5×. However, the gap between the power efficiencies (1.4×) of the two processors is smaller than that between their OpenCL performances.

### 7.1.4 OpenACC evaluation and comparison with OpenCL

In this section, we evaluate the OpenACC implementation of the seismic modeling application on the Tahiti GPU and on the Kaveri APU. We extended the OpenACC code used for the 3D finite difference stencil presented in section 6.5. We also consider the same three OpenACC variants "OpenACC only", "OpenACC + HMPPcg" and "OpenACC + HMPPcg + code modification". Note that the modification of the initial code, in the third version, involves increasing the ILP by explicitly rewriting the loops, and also using the GPU local memory. In table 7.3, we summarize the performance numbers

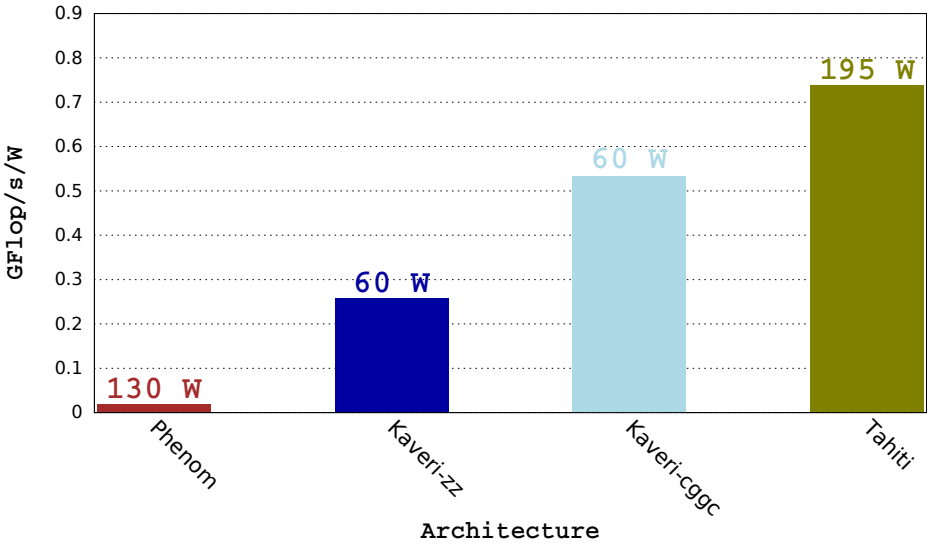



FIGURE 7.3: **Comparison** between the **performance per Watt** numbers of the **seismic modeling** application obtained on the **CPU**, **APU** and **GPU**.

of the three OpenACC versions (computation times only). The table also present the number of lines of code (LOC) that were added to the initial Fortran CPU code of the seismic modeling application, in order to adapt it to the GPUs and to the APUs. As it was the case for the OpenACC implementation of the 3D finite difference stencil, the best OpenACC performance of the seismic modeling application is obtained when using OpenACC coupled with HMPPcg and that for both Tahiti and Kaveri. Besides, increasing the ILP and relying on the local memory, not only did require an extensive effort to modify the initial code (74 LOC added) but also it did not help deliver higher performance than that of the "OpenACC + HMPPcg" version (only 29 LOC). In table 7.4, we present a comparison between the best OpenCL performance numbers and the best OpenACC numbers of the seismic modeling application. The table also shows the number of OpenCL LOC and OpenACC LOC added to the initial code of the seismic modeling application. For both Tahiti and Kaveri, OpenACC offered almost half the

| | Kaveri ($GFlop/s$) | Tahiti ($GFlop/s$) | #LOC |
|---|---|---|---|
| OpenACC only | 15.18 | 60.22 | 25 |
| OpenACC + HMPPcg | 17.61 | 77.77 | 29 |
| OpenACC + HMPPcg + code modification | 13.43 | 53.19 | 74 |

TABLE 7.3: Comparison of the different **OpenACC** implementations of the **seismic modeling** application on **Kaveri** and **Tahiti**, in terms of performance and of numbers of lines of code added to the initial CPU implementation.

| | Kaveri ($GFlop/s$) | Tahiti ($GFlop/s$) | #LOC |
|---|---|---|---|
| OpenACC | 17.61 | 77.77 | 29 |
| OpenCL | 32.03 | 144.02 | 762 |

TABLE 7.4: **Comparison** between the best **OpenACC** implementation and the best scalar **OpenCL** implementation of the **seismic modeling** application on Kaveri and Tahiti, in terms of performance and of numbers of lines of code added to the initial CPU implementation.

OpenCL performance. However, the OpenCL performance comes with adding 762 LOC to the initial code which is 26× the number of OpenACC LOC added.

## 7.2 Seismic migration

In this section, we extend our study to the seismic migration application. We give an overview of the RTM algorithm which involves data snapshotting. Besides, we evaluate the performance and the power efficiency of the application in OpenCL on the different architectures. Finally, we implement the RTM in OpenACC and compare its performance and ease of programmability against those of the OpenCL implementation.

### 7.2.1 Description of the algorithm

In this section we focus on the seismic migration also presented in section 2.2. The seismic migration can be seen as two successive sweeps of the seismic modeling. In the first sweep, referred to as the forward sweep or "FWD", the wave equation is solved exactly like in the seismic modeling application. In the second sweep, called the backward sweep or "BWD", the wave equation is solved backward in time. We present in algorithm 7.2 a high level description of the seismic migration workflow. Similarly to the seismic modeling, the computation involves two arrays of the wavefield $u0$ and $u1$ that are interleaved throughout the simulation iterations in order to compute the temporal derivatives. The forward sweep is similar to the seismic modeling workflow. However, during this workflow snapshots of the source wavefield are periodically stored on disk (this was not the case in the seismic modeling application). We recall that we use the "selective wavefield storage" method with a data snapshotting frequency of 10. During the backward sweep, the snapshots, saved during the forward sweep, are read from disks in the subroutine *read_seismic_snapshot*(). Then, we inject the seismic traces (those usually represent the data collected after a seismic survey) on the receivers positions. This is held in the subroutine *add_seismic_receivers*(). After propagating the receiver wavefield backward, the imaging condition is performed in the *imaging_condition*() subroutine (on the CPU) by correlating the source and receiver wavefields.

### 7.2.2 Accelerating the seismic migration using OpenCL

We considered the same testbed as that of the seismic modeling. The same versions and optimization parameters are also used for the seismic migration. However, in the case of the seismic migration we also consider the data snapshotting in the algorithm, which incurs I/O operations and may require memory transfers between CPU and GPU in the APU and discrete GPU implementations. It is to be noted that the computers that contain the surveyed processors (*AMD Phenom TM II x6 1055t Processor*, Kaveri and Tahiti), are connected to an NFS file system and run under a Linux OS (Ubuntu 14.04). We summarize the obtained performance numbers on Kaveri in table 7.5 and those on Tahiti in table 7.6. We differentiate the performance of the OpenCL kernels ("comp-only") from that of the whole application ("overall"). Besides, we distinguish the performance of the FWD from that of the BWD.

For the "comp-only" case, we came to the same conclusions observed for the seismic modeling application. As a matter of fact, the best performance numbers are obtained when considering the *scalar* implementation on both Tahiti and Kaveri with the *cggc* data placement strategy. Besides, the *local scalar* version is more beneficial to the APU when using the zero-copy memory objects. In addition, the multiple OpenCL kernels approach offers up to 12% of performance enhancement for the APU and only up to 3% for the discrete GPU. Moreover, when using zero-copy memory objects the performance of the seismic migration on the Kaveri APU is almost reduced to half, which is due to a two fold difference in terms of memory bandwidth between the `z` memory and the `g` memory on APUs. Furthermore, while the forward and backward performances are

**Algorithm 7.2** High level description of the one-node seismic migration (RTM) algorithm.

```
1: for t ∈ [0..T] do                                   ▷ the forward sweep (or FWD)
2:     if mod(t, 2) == 0 then                          ▷ t is the time-step index
3:         add_seismic_source(u0, t)
4:         update_wavefield(u1, u0, t)                 ▷ see algorithm 7.1
5:         save_seismic_snapshot(u1, t)                ▷ this involves I/O operations
6:     else
7:         add_seismic_source(u1, t)
8:         update_wavefield(u0, u1, t)
9:         save_seismic_snapshot(u0, t)
10:    end if
11: end for
12:
13: for t ∈ [T..0] do                                  ▷ the backward sweep (or BWD)
14:    read_seismic_snapshot(tmp, t)                   ▷ tmp is used to read snapshots
15:    if mod(t, 2) == 0 then
16:        add_seismic_receivers(u0, t)
17:        update_wavefield(u1, u0, t)
18:        imaging_condition(u1, tmp, t)
19:    else
20:        add_seismic_receivers(u1, t)
21:        update_wavefield(u0, u1, t)
22:        imaging_condition(u0, tmp)
23:    end if
24: end for
25:
26: function imaging_condition(u, tmp)
27:    img = 0                                         ▷ img is the final image
28:    for z ∈ [0..nz − 1] do                          ▷ z is the array coordinate in Z
29:        for y ∈ [0..ny − 1] do                      ▷ y is the array coordinate in Y
30:            for x ∈ [0..nx − 1] do                  ▷ x is the array coordinate in X
31:                img(z, y, x) += u(z, y, x)*tmp(z, y, x)
32:            end for
33:        end for
34:    end for
35: return img
36: end function
```

very close on the Kaveri APU, they are different on the Tahiti GPU (the bwd sweep performance numbers are often 8% lower than the performance numbers of the fwd sweep) which is unexpected.

For the "overall" case, one can notice that for the Kaveri APU the performance of the fwd sweep is often higher than that of the bwd sweep, especially when the `cggc` DPS is used. After investigation, we found out that the cost of the data snapshotting during the fwd sweep (write I/O operations) is higher than that during the bwd sweep (read I/O operations), which explains the performance difference between the two steps of the RTM algorithm. However, the fwd and bwd performances on the Tahiti GPU are very comparable, as opposed to the "comp-only" case. It is to be noted that the overall times of the seismic migration on discrete GPUs include the data snapshotting times (which incurs an overhead spent in transferring data back and forth between CPU and GPU via the PCI Express bus). It can be seen in table 7.6, that the performance of the "overall" is almost 4× lower than the performance of the OpenCL kernels. This clearly emphasizes the impact of the PCI Express bus and of the extensive I/O operations on the performance of the RTM on discrete GPUs. This difference is much more smaller in the case of the APU when using the `cggc` DPS, since the table 7.5 reports that the overall performance is, at worst, 2× lower than the performance of the OpenCL kernels of the seismic migration on APUs. Indeed, the explicit copy from CPU memory to GPU memory on APUs is equivalent to a regular copy from or to the system RAM which has a higher sustained bandwidth than that of the PCI Express bus. Besides, the performance difference is even much lower (only up to 23%) when the `zz` DPS is applied. As a matter of fact, this is due to that data copies from the CPU to the GPU are no longer needed in this case, since the GPU cores directly access the CPU memory through the Onion bus.

Finally, the figure 7.4, shows a CPU, APU and GPU performance comparison of the seismic migration. In the "comp-only" case, we notice that similarly to the seismic modeling, the CPU delivers a poor performance compared to that of the Kaveri APU and that of the Tahiti GPU. The figure also reports almost the same performance ratios

| Kernel type | #Kernels | DPS | LX | LY | ILP | comp-only (*GFlop/s*) | | overall (*GFlop/s*) | |
|---|---|---|---|---|---|---|---|---|---|
| scalar | One | *cggc* | 32 | 8 | 1 | fwd | 28.45 | fwd | 16.43 |
| | | | | | | bwd | 27.55 | bwd | 13.93 |
| local scalar | One | *cggc* | 32 | 8 | 1 | fwd | 26.84 | fwd | 15.22 |
| | | | | | | bwd | 26.33 | bwd | 13.66 |
| scalar | One | *zz* | 128 | 2 | 2 | fwd | 12.60 | fwd | 10.01 |
| | | | | | | bwd | 12.81 | bwd | 10.00 |
| local scalar | One | *zz* | 256 | 1 | 2 | fwd | 14.56 | fwd | 11.26 |
| | | | | | | bwd | 14.13 | bwd | 10.42 |
| **scalar** | **Multiple** | **cggc** | **64** | **1** | **2** | **fwd** | **32.45** | **fwd** | **16.89** |
| | | | | | | **bwd** | **31.39** | **bwd** | **14.98** |
| local scalar | Multiple | *cggc* | 32 | 2 | 1 | fwd | 26.81 | fwd | 15.40 |
| | | | | | | bwd | 26.02 | bwd | 13.34 |
| scalar | Multiple | *zz* | 128 | 1 | 8 | fwd | 13.99 | fwd | 11.03 |
| | | | | | | bwd | 13.84 | bwd | 10.60 |
| **local scalar** | **Multiple** | **zz** | **64** | **1** | **2** | **fwd** | **15.25** | **fwd** | **11.61** |
| | | | | | | **bwd** | **15.18** | **bwd** | **11.21** |

TABLE 7.5: Performance parameters and numbers of the **seismic migration** application on **Kaveri**. The numbers marked in **bold** are the best achieved performance numbers for each configuration.

| Kernel type | #Kernels | LX | LY | ILP | comp-only (*GFlop/s*) | | overall (*GFlop/s*) | |
|---|---|---|---|---|---|---|---|---|
| scalar | One | 64 | 4 | 2 | fwd | 140.59 | fwd | 29.60 |
| | | | | | bwd | 135.08 | bwd | 29.95 |
| local scalar | One | 64 | 1 | 1 | fwd | 124.78 | fwd | 28.22 |
| | | | | | bwd | 116.80 | bwd | 28.43 |
| **scalar** | **Multiple** | **64** | **2** | **2** | **fwd** | **144.52** | **fwd** | **29.97** |
| | | | | | **bwd** | **138.86** | **bwd** | **29.19** |
| local scalar | Multiple | 256 | 1 | 1 | fwd | 117.04 | fwd | 28.01 |
| | | | | | bwd | 107.92 | bwd | 27.06 |

TABLE 7.6: Performance parameters and results of the **seismic migration** application on **Tahiti**. The numbers marked in **bold** are the best achieved performance numbers.

than the seismic modeling. Indeed, the performance of Tahiti is 4.4× (resp. 9.3×) higher than that of the APU with the `cggc` DPS (resp. `zz` DPS) and given that only the computation times are considered. In the "overall" case and because of the impact of the PCI Express bus and I/O operations on the discrete GPUs performance, the ratio decreases down to 1.8× (resp. 2.5×) when considering the `cggc` DPS (resp. `zz` DPS) for the APU. Finally, we recall that the "overall" performance depends on the data snapshotting frequency used in the RTM algorithm. In our case we used a frequency of 10 which corresponds to the case $K = 10$ in the figure 6.24. As a matter of fact, similarly to that case we observed that the GPU outperforms the APU.

To conclude, we can advance that the multiple OpenCL kernels approach can help improve, as much as for the seismic modeling, the performance of the seismic migration at the node level. Therefore, we will rely on the same approach for the multi-node implementations. Besides, we have shown the impact of the PCI Express bus and of the I/O operations on the performance of the RTM on discrete GPUs, while we assessed the relevance of APUs on how they help mitigate this impact (when using both the `cggc` and the `zz` DPSs), even though their overall performance falls behind that of the discrete GPUs.

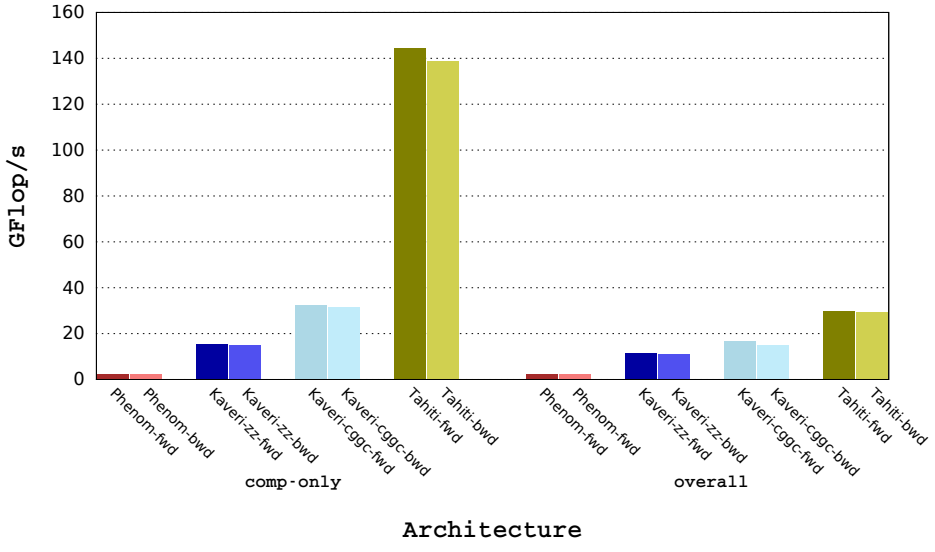



FIGURE 7.4: **Comparison** between the performance numbers of the **seismic migration** application obtained on the **CPU**, on the **APU** and on the **GPU**, with a data snapshotting **frequency** equal to **10**.

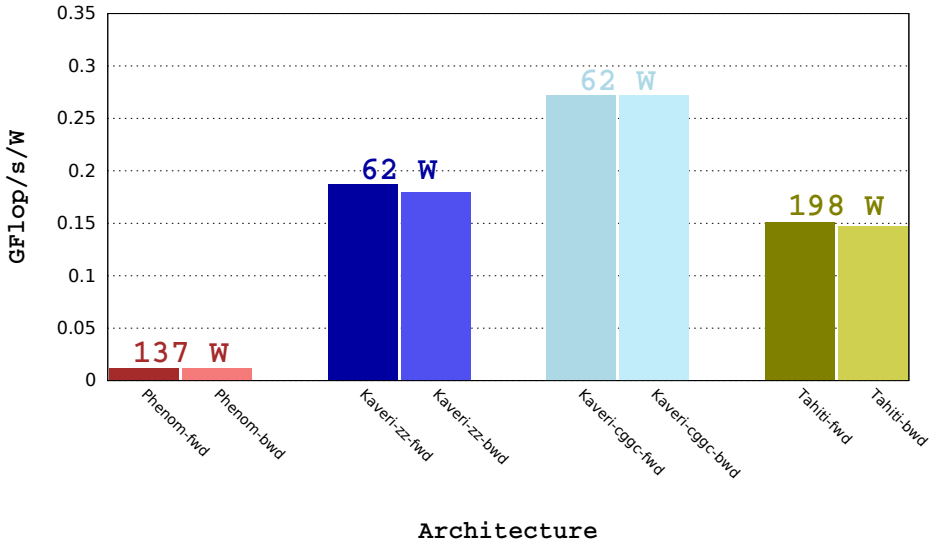



FIGURE 7.5: **Comparison** between the **performance per Watt** numbers of the **seismic migration** application obtained on the **CPU**, **APU** and **GPU**.

### 7.2.3 Performance and power efficiency

In this section, we measure the power consumption of the seismic migration on the Phenom CPU, on the Kaveri APU and on the Tahiti GPU. We recall that we measure the power consumption of each system as a whole, thus in the case of the discrete GPU based system, the measured power includes the power drawn by the CPU that manages the GPU. We correlate the power consumption measurements with the best OpenCL performance numbers, based on the overall timings presented in tables 7.5 and 7.6, in order to elaborate a performance per Watt presented in figure 7.5. The figure shows, here again, that the power efficiency of the seismic migration on the Phenom CPU is very low (roughly 0.013 $GFlop/s/W$ for both fwd and bwd). Besides, one can notice that the APU (while using the `cggc` DPS) is almost 2× more power efficient than the GPU which are both more than an order of magnitude more efficient than the Phenom CPU. As a matter of fact, even when using the `zz` DPS, the APU is 23% more energy efficient than the discrete GPU. This implies that the APU is an attractive solution for seismic migration when both performance and power are concerned.

### 7.2.4 OpenACC evaluation and comparison with OpenCL

In this section we show the OpenACC performance numbers of the seismic migration application (the performance of the subroutine $update\_wavefield()$ only). Similarly to the seismic modeling, we consider three OpenACC implementations. We summarize the performance numbers of the three variants in table 7.7. We also give the number of LOC that are added to the initial code and that for each OpenACC based implementation. We can see that the best performance is obtained when using OpenACC coupled with HMPPcg (almost a 28% of performance enhancement compared to the "OpenACC only" version, at the expense of adding 4 LOC). As a matter of fact the performance of the forward sweep as well as that of the backward sweep are "naturally" almost equal to the OpenACC performance of the seismic modeling. Finally, in table

| | Kaveri ($GFlop/s$) | | Tahiti ($GFlop/s$) | | #LOC |
|---|---|---|---|---|---|
| OpenACC only | fwd | 15.18 | fwd | 60.71 | 30 |
| | bwd | 15.01 | bwd | 59.91 | |
| OpenACC + HMPPcg | fwd | 17.83 | fwd | 77.98 | 34 |
| | bwd | 17.70 | bwd | 77.13 | |
| OpenACC + HMPPcg + code modification | fwd | 13.43 | fwd | 53.95 | 79 |
| | bwd | 13.68 | bwd | 52.81 | |

TABLE 7.7: Comparison of the different **OpenACC** implementations of the **seismic migration** application on **Kaveri** and **Tahiti**, in terms of performance and of numbers of lines of code (LOC) added to the initial CPU implementation.

7.8 we compare the OpenACC performance against the OpenCL performance of the seismic migration. When considering the computation times only, the best OpenACC implementation, which adds 34 LOC to the initial code, offers half the performance of the best OpenCL implementation of the seismic migration for both the Kaveri APU and the Tahiti GPU. However, the best OpenCL implementation of the seismic migration application required adding 779 LOC to the initial Fortran code.

## 7.3 Conclusion

To sum up, in this chapter we evaluated the seismic modeling and the seismic migration (RTM) applications at the node level. The evaluation included an OpenCL performance study as well as power efficiency measurements, both conducted on an *AMD Phenom TM II x6 1055t Processor*, on a Kaveri APU and on a Tahiti discrete GPU. It also comprised a comparative study between OpenCL and OpenACC based implementations of the both applications in terms of performance and programming efforts. The study showed that the hardware accelerators outperform the CPU and that the discrete GPU naturally delivers better performance than the APU, especially when considering the computation times only. However, it was shown that the PCI Express bus and the extensive need for I/O operations had seriously impacted the performance of the RTM on the discrete GPU, while the APU alleviated this impact. As a matter of fact, when using the zero-copy memory objects, the overhead due to the memory traffic between the CPU and the GPU is removed, but the performance of the APU is cut to half as compared to when using the `cggc` DPS. Besides, the APU was found an attractive solution for the seismic migration as far as both the performance and the power consumption are concerned. Moreover, this study showed that the directive based approach offered only half the OpenCL performance, since we were unable to reproduce the same optimizations

| | Kaveri ($GFlop/s$) | | Tahiti ($GFlop/s$) | | #LOC |
|---|---|---|---|---|---|
| OpenACC | fwd | 17.83 | fwd | 77.98 | 34 |
| | bwd | 17.70 | bwd | 77.13 | |
| OpenCL | fwd | 32.45 | fwd | 144.52 | 779 |
| | bwd | 31.39 | bwd | 138.86 | |

TABLE 7.8: **Comparison** between the best **OpenACC** implementation and the best scalar **OpenCL** implementation of the **seismic migration** application on **Kaveri** and **Tahiti**, in terms of of performance and of numbers of lines of code (LOC) added to the initial CPU implementation.

we used in OpenCL by means of the OpenACC directives, and even by modifying the initial code. However the directive based approch required much less programming efforts than implementing the applications in OpenCL (26× less LOC). Finally, this chapter helped to collect the recommendations to follow, at the node level, in terms of optimization techniques and auto-tuning parameters that we will use in the multi-node implementations of the seismic applications in chapter 8, where we will focus more on large scale related optimizations (MPI communications, domain decomposition etc.), and will investigate the interest of APUs in terms of achieving efficient scaling as compared to discrete GPUs.

# Chapter 8

# Large scale seismic applications on CPU/APU/GPU clusters

**Contents**



Subsurface high-quality seismic images can be challenging to obtain, especially when the source and receivers aperture is very large or when rocks or salt domes bury the oil or gas reservoirs. In order to improve the speed, quality and accuracy of subsurface images and interpretations we rely on large scale high performance compute facilities. In chapter 7 we discussed implementation details and shared performance numbers of the seismic applications on the node level. In this chapter we discuss a selection of well known problems in the HPC community related to large scale deployments of scientific applications in section 8.1, then try to take the seismic applications beyond the single node optimizations, in sections 8.2 and 8.3, which is considered as one of the main contributions of this work.

In details, we first sketch the most relevant guidelines in order to implement efficient large scale depth imaging applications on multi-node hybrid architectures. Those include the mainstream CPU clusters, GPU clusters and APU clusters. We discuss the architectural challenges and the algorithmic problems related to these implementations such as domain decomposition, load balancing, ghost region exchange, and communication overhead.

Then, we propose a large scale implementation of the seismic modeling application where MPI communications are overlapped with useful computation. We demonstrate

the reliability of this approach through a set of scalability tests. In addition to CPU clusters, we adapt the implementation to discrete GPU clusters and APU clusters. We show performance numbers of scalability on the different architectures followed by a performance comparison between the best variant on each platform.
Finally, given that the seismic modeling is a building block of the seismic migration, we extend the seismic modeling to propose a large scale implementation of the seismic migration. Besides the details that we point out for the seismic modeling we focus on several issues that are specific to the migration application such as the impact of IO operations, incurred by the enormous temporary data (Terabytes of data) generated during computation, on the application performance and scalability.
This chapter, jointly with the previous one, was subject to the following publication *"I. Said, P. Fortin, J.-L. Lamotte and H. Calandra. Leveraging the Accelerated Processing Units for seismic imaging: a performance and power efficiency comparison against CPUs and GPUs (submitted on October 2015 to an international journal)."*

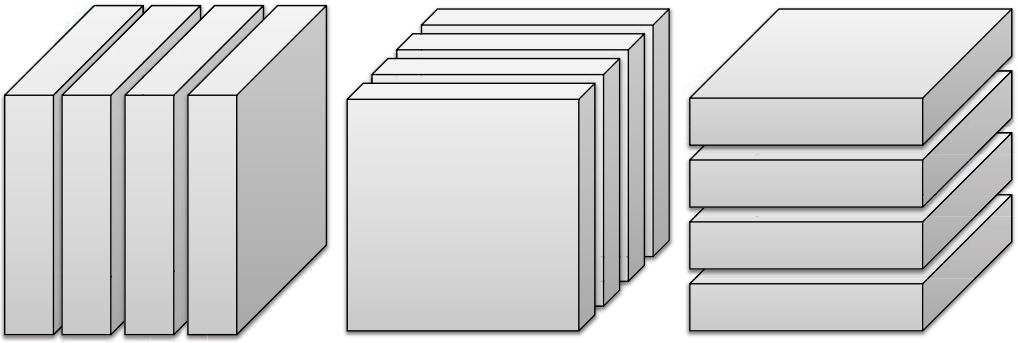
## 8.1 Large scale considerations

This section elaborates on some common practices often applied on large scale scientific applications with numerical solvers. Those are general considerations but may take special forms with depth imaging applications.

### 8.1.1 Domain decomposition

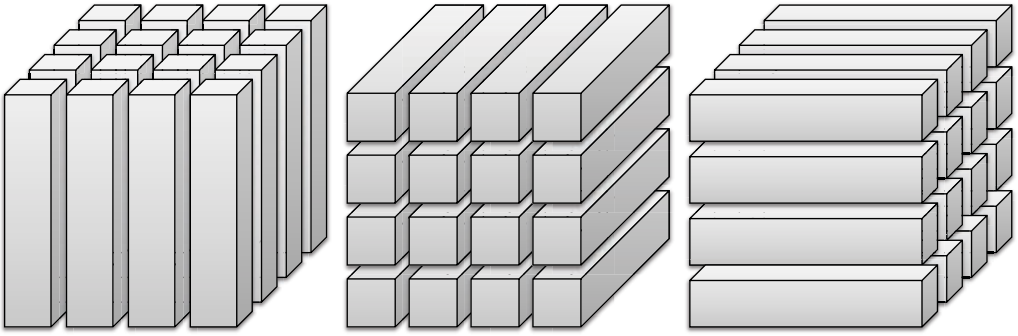
In the context of geophysics exploration, it is crucial to implement numerical methods to model seismic wave propagation in large structures at scales of, at least, tens of kilometers. The velocity model *3D SEG/EAGE salt model*, presented in figure 5.2, of the problem under study is approximately 8 km width by 27 km length by 4 km depth (note that the size of the compute grids related to this model depend on the discretization steps, see table 5.7). Furthermore, the numerical solution of the underlying PDE of the wave phenomena, cannot be processed on a single computer because of the huge memory and compute requirements. For instance, the compute grid $\Omega_{4\times4\times4}$, presented in the table 5.7 of the section 5.3.2.2, may be of size $2245 \times 6885 \times 1068$ which represents more than 16.5 billion grid points. Given that we consider a fine grid spacing ($4m$ in each direction) and that multiple instances of $\Omega_{4\times4\times4}$ are needed by the wave equation solver, almost a terabyte of data is required in order to perform RTM on the *3D SEG/EAGE salt model*.

For this purpose, the wave equation solver has to be deployed on the large scale (up to 64 nodes in our case) and involve processing large quantities of input data that can not be stored in one computer. Rather it has to be distributed on many compute resources in a *data parallel* fashion, i.e., by multiple processors working on different parts of the data. As a matter of fact, this is the dominant coarse-grained parallelization concept in scientific computing on distributed memory architectures. The SPMD (Single Program Multiple Data) programming model is used, as usually the same code is executed on all processors. This general strategy is referred to as the *domain decomposition method* [139].



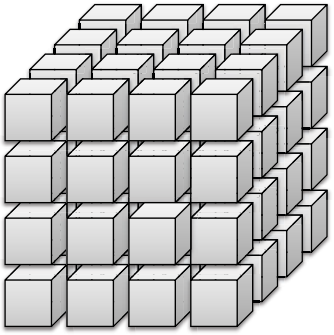
(a1) X-slab (a2) Y-slab (a3) Z-slab

(a) 1D or "Slabs" decomposition.



(b1) XY-pole (b2) ZX-pole (b3) ZY-pole

(b) 2D or "Poles" decomposition.



(c) 3D or "Cubes" decomposition.

FIGURE 8.1: Domain decomposition strategies of 3D cubic domain of size $L \times L \times L$, into $N$ subdomains. In this example, the number of subdomains ($N$) is not the same in each strategy.

By domain decomposition, a large discretized computational domain, e.g. a volume that defines discrete positions or amplitudes of physical quantities on a structured or an unstructured grid, is split into a set of sub-grids usually named *subdomains* that are distributed across the compute nodes. Usually the subdomains interact during the computation since most of the numerical methods, such as Finite Element Method (FEM), Finite Difference Method (FDM) and Finite Volume Method (FVM), involve neighboring computations and require communication between the compute nodes.

How exactly the subdomains should be formed out of the original grid may be a challenging problem to solve since the data volume to be communicated is often proportional to the inter-domain surface area. Furthermore, the decomposition may depend

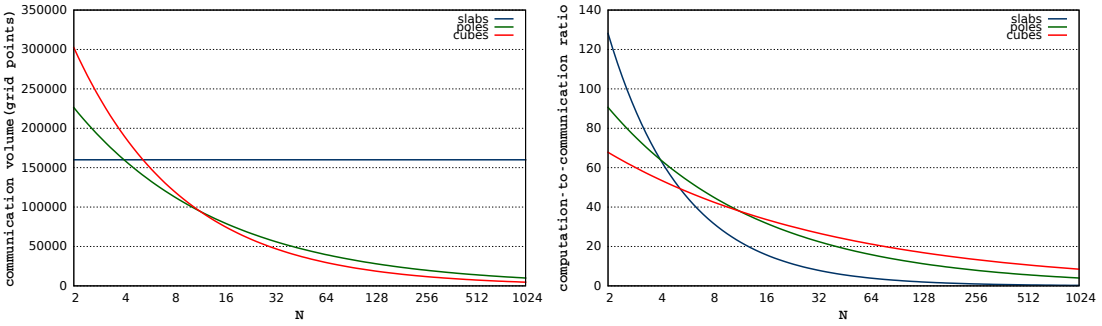
| | *Communication volume:* $g(L,N)$ | *Ratio* |
|---|---|---|
| Slabs | $g(L,N) = L.L.2.d = 2dL^2$ | $\frac{fL}{2dN}$ |
| Poles | $g(L,N) = L.\frac{L}{\sqrt{N}}.d.(2+2) = 4dL^2N^{-1/2}$ | $\frac{fL}{4dN^{1/2}}$ |
| Cubes | $g(L,N) = \frac{L}{\sqrt[3]{N}}.\frac{L}{\sqrt[3]{N}}.d.(2+2+2) = 6dL^2N^{-2/3}$ | $\frac{fL}{6dN^{1/3}}$ |

TABLE 8.1: Communication volume and computation-to-communication ratio, per subdomain, after domain decomposition of a cubic compute grid of size $L \times L \times L$ into $N$ subdomains. $d$ is the number of grid points needed to fulfill the computation dependency on each direction of each grid dimension.$f$ is the number of floating point operations needed to perform one elementary computation.

on the underlying hardware characteristics, such as the amount of memory available for computation and the hardware and software prefetching in the case of CPUs [84].

A thorough analysis is generally needed to determine the optimal decomposition for a given numerical problem and a given hardware architecture. Figure 8.1 summarizes the most common domain decomposition strategies depending on whether the domain cuts are performed in one, two or three dimensions. In the subfigure 8.1a, the different possibilities of decomposing a computational grid into *slabs* or layers are presented. The domain can also be split along two physical dimensions as shown in the subfigure 8.1b to form a set of *poles*. The original domain can also be divided into *cubes* as depicted in the subfigure 8.1c. Note that when we refer to cubic subdomain or cube domain decomposition, we do not mean that subdomains have to be perfectly cube-shaped but rather have rectangular faces. Suppose that the original domain is a cubic domain of size $L \times L \times L$ and decomposed into $N$ subdomains (we consider here that the size of the compute grid remains unchanged). The maximum number of grid points to be communicated by one subdomain with its neighbors, is referred to as $g(L,N)$ (inspired from [102]) since it depends essentially on $L$ and on the decomposition factor $N$. Note that we assume that each subdomain collaborates with the maximum number of neighbors possible which is two in the case of slabs, four in the case of poles and six in the case of cubes. In the table 8.1, we give the expression of $g(L,N)$ in each domain decomposition strategy. $d$ represents the degree of data dependency, on each direction of each grid dimension, required in order to perform an elementary computation. Note that both the data required to be sent and the data required to be received are considered in the evaluation of $d$, and that for a given degree $d$, $d$ buffers are used to exchange the data. For example in the case of a $2^{nd}$ order stencil or a Jacobi solver $d = 2$ (which means that one buffer is used to send data, and that one buffer to receive data), in the case of an $8^{th}$ order centered finite difference stencil scheme, which is the building block of the wave equation solver we use, $d = 8$.

The idea behind searching the optimal domain decomposition strategy is to minimize the inter-domain surface area and the overall communication volume on the one hand, and to increase the ratio of computation over the communication volume on the other hand. We plot in figure 8.2a the evolution of the volume of data to be communicated (in grid points) per subdomain as a function of the decomposition factor $N$. The computation to communication ratio per subdomain is shown in figure 8.2b. For the sake of example we pick $L = 100$, $d = 8$ and $f = 41$ (the number of operations used in an $8^{th}$ order stencil computation). We notice that the 1D decomposition incurs an $N$-independent communication overhead. This may harm the scalability, especially when the compute grid size remains the same, since the cost of communication is constant whereas the computation cost per process decreases (see figure 8.2b) which saturates




(a) (b)

FIGURE 8.2: The impact of domain decomposition strategies on the communication volume(a) and computation to communication ratio(b) per subdomain, with $L = 100$, $d = 8$ and $f = 41$.

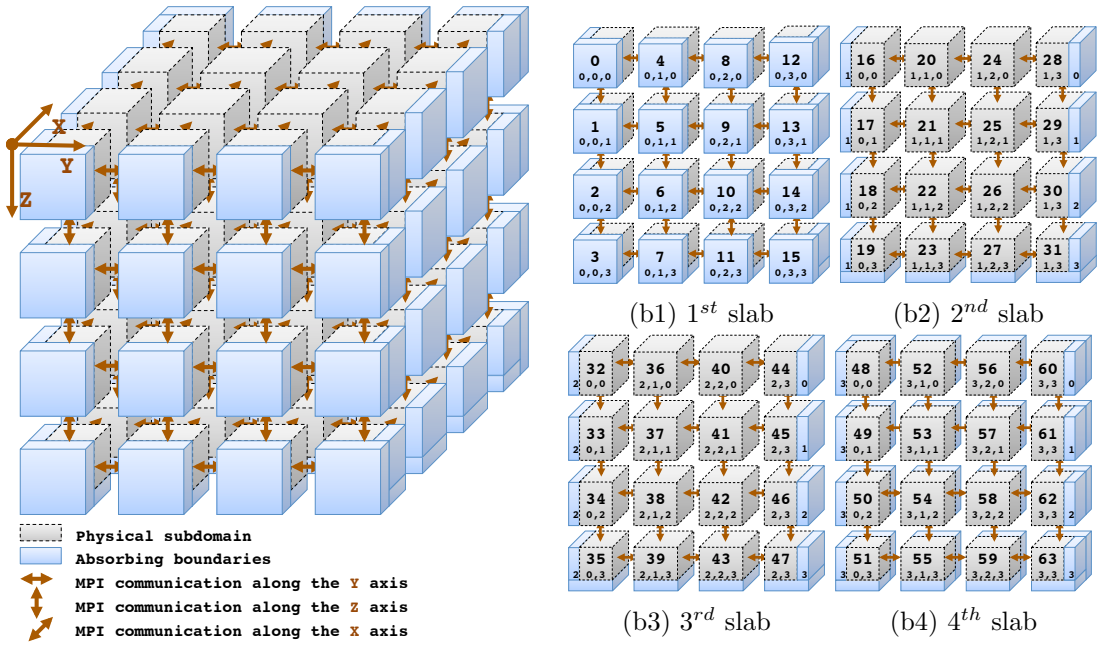
the performance as $N$ gets bigger. In the contrary, the negative power of $N$ appearing in the expression of $g(L, N)$ in both the "slabs" decomposition and the "cubes" decomposition dampens the communication overhead as $N$ increases as it is pointed out in the figure. More importantly, the cubic domain decomposition plot (red line) presents the steepest decline in figure 8.2b compared to that of the 2D decomposition, and has the smallest communication volume, when $N$ is greater than 10 or 11. It also shows the best computation to communication ratio when $N$ increases (see figure 8.2b).

Theoretically, we thus conclude that the 3D domain decomposition ensures the best scalability for large scale implementations. In [102], the authors present a detailed comparison of the different strategies through a 3D Jacobi solver. They also demonstrate that the best performance is often obtained with cubic subdomains or pole decomposition, given that some specific requirements can be fulfilled, e.g. the latency cost of the underlying network is dominated by the cost of its bandwidth.

In practice, hardware characteristics may sometimes influence the performance of such strategies. For example, emerging high-end processors feature hardware prefetching capabilities, that is a mechanism to pre-load data and reduce cache misses latencies. In this case, "poles" decomposition is more suitable if the prefetch dimension, which is usually the fastest in terms of memory accesses, is not partitioned. Furthermore, if accelerators such as GPUs or Intel's MIC are used, the communication model $g(L, N)$ have to be altered since a PCI Express communication is needed to transfer data from the host (CPU) to the accelerator. In [23], Abdelkhalek et al. use a 1D domain decomposition in order to mitigate the impact of PCI Express, which incurs additional communications to copy data from the CPU to the GPU prior computation, on the performance of their multi-GPU implementation of the seismic modeling application. Those copies would involve regions that are not contiguous in memory, if other domain decomposition strategy is used.

In the context of this work, in order to ensure a better scalability when big datasets are deployed on a large number of compute resources, we make use of the "cubes" domain decomposition to implement the wave equation solver, that is used in both seismic modeling and seismic migration, on the different architectures.

We propose in figure 8.3 an example of a "cubes" domain decomposition of one of the compute grids of the problem under study: $\Omega_{16\times16\times16}$ which is presented in figure




(a) $4 \times 4 \times \times 4$ domain decomposition: subdomain layout.

(b) Mapping between the MPI ranks and the cartesian coordinates of the decomposition.

FIGURE 8.3: 4x4x4 "cubes" domain decomposition of $\Omega_{16\times16\times16}$ using a total of 64 MPI processes. Each subdomain is assigned to one MPI process.

7.1, into 64 subdomains arranged in a cartesian coordinate system of $4 \times 4 \times 4$ in such that each subdomain is assigned to an MPI process allocated on an independent compute resource. Each process is responsible for updating, locally, the wavefields within its own sub-grid of the overall simulation volume. The global layout of the 64 subdomains and the different communication pipes are illustrated in figure 8.3a. The mapping between the MPI processes ranks and the cartesian coordinates is emphasized in figure 8.3b and is governed by the following formula: $rank(x, y, z) = (y + (x * ddy)) * ddz + z$, where $rank(x, y, z)$ is the MPI rank, $x$ is the coordinate of the MPI process with respect to the $X$ axis, $y$ is the one along the $Y$ axis and $z$ is that along the $Z$ axis. $dd_x$, $dd_y$ and $dd_z$ are defined in the notation summary 8.2 where all the symbols used in this section are described. The domain decomposition partitions the simulation volume into smaller subdomains where the total number of subdomains $(4 * 4 * 4)$ matches the number of MPI processes (64) used in the simulation.

By means of domain decomposition, we aim to conduct a series of scalability tests. We distinguish, as is customary in the high performance computing community, between *strong* and *weak* scalings. Strong scaling is running a fixed-sized problem on a varying number of processors, these include commodity CPUs, hardware accelerators (HWAs) such as GPUs and APUs etc. or co-processors such as FPGAs, and then study how

| | |
|---|---|
| $dd_x$ | the number of subdomains along the $X$ axis |
| $dd_y$ | the number of subdomains along the $Y$ axis |
| $dd_z$ | the number of subdomains along the $Z$ axis |

TABLE 8.2: Summary of the notations used, in section 8.1.1, to define the compute grid.

| $\#nodes$ | $dd_x$ | $dd_y$ | $dd_z$ | *compute grid* | |
|---|---|---|---|---|---|
| | | | | *seismic modeling* | *seismic migration* |
| 1 | 1 | 1 | 1 | $\Omega_{8\times8\times8}$ | $\Omega_{9\times9\times9}$ |
| 2 | 1 | 2 | 1 | | |
| 4 | 2 | 2 | 1 | | |
| 8 | 2 | 2 | 2 | | |
| 16 | 2 | 4 | 2 | | |
| 32 | 4 | 4 | 2 | | |
| 64 | 4 | 4 | 4 | | |

TABLE 8.3: Numerical parameters of the **domain decomposition** with respect to the **strong** scaling scenario.

| $\#nodes$ | $dd_x$ | $dd_y$ | $dd_z$ | *compute grid* |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | $\Omega_{16\times16\times16}$ |
| 2 | 1 | 2 | 1 | $\Omega_{16\times16\times8}$ |
| 4 | 2 | 2 | 1 | $\Omega_{16\times8\times8}$ |
| 8 | 2 | 2 | 2 | $\Omega_{8\times8\times8}$ |
| 16 | 2 | 4 | 2 | $\Omega_{8\times8\times4}$ |
| 32 | 4 | 4 | 2 | $\Omega_{8\times4\times4}$ |
| 64 | 4 | 4 | 4 | $\Omega_{4\times4\times4}$ |

TABLE 8.4: Numerical parameters of the **domain decomposition** with respect to the **weak** scaling scenario, for both **seismic modeling** and **seismic migration**.

the execution time varies with respect to the number of processors. Whereas, in a weak scaling configuration, the amount of work per processor is fixed since compute resources increase proportionally with the problem size. In the ideal case the execution time should remain constant with respect to the number of processors. From a geophysics exploration perspective, strong scaling is used to accelerate modeling or migration applications with a fixed amount of seismic data, e.g. seismic traces and velocity models, used as application inputs. Weak scaling is either used to cover larger input data or used to refine compute grids for a better accuracy and for a higher resolution subsurface imaging. We point out that in the scope of this work we use weak scaling rather for compute grid refinement.

Table 8.3 summarizes the domain decomposition details we use to conduct the strong scaling tests of seismic modeling and migration. It also presents the compute grid used for each configuration. We recall that the numerical parameters of the compute grids were given in section 5.3.2.2. We consider running one shot experiment on 1 to 64 nodes (this is valid for the CPU cluster only, we only use up to 16 APU nodes since that is the maximum capacity of the APU cluster; as for the GPU cluster we also use up to 64 nodes). Note that we do not allocate a bigger number of nodes per shot because, at a higher level a coarser grained and natural parallelism is applied since multiple shots are processed in the same time by independent groups of nodes. Note that the spatial discretization steps used in seismic migration is greater than that used in seismic modeling. That is because in the migration application more instances of the compute grid are needed, and using a discretization step equal to 8 would make it impossible to fit into the memory of one compute node. Table 8.4 presents the configuration of the weak scaling related domain decomposition, as well as the different compute grids to be used in this case.

| | *px* | *py* | *pz* | *compute grid* |
|---|---|---|---|---|
| seismic modeling | 33 | 33 | 33 | $\Omega_{8\times8\times8}$ |
| seismic migration | 29 | 29 | 29 | $\Omega_{9\times9\times9}$ |

TABLE 8.5: The width of the **PML layers** present in the compute grids with respect to the **strong** scaling configuration. The width depends on the spatial discretization steps, $dx$, $dy$ and $dz$.

| *#nodes* | *px* | *py* | *pz* | *compute grid* |
|---|---|---|---|---|
| 1 | 16 | 16 | 16 | $\Omega_{16\times16\times16}$ |
| 2 | 16 | 16 | 33 | $\Omega_{16\times16\times8}$ |
| 4 | 33 | 16 | 33 | $\Omega_{16\times8\times8}$ |
| 8 | 33 | 33 | 33 | $\Omega_{8\times8\times8}$ |
| 16 | 33 | 33 | 67 | $\Omega_{8\times8\times4}$ |
| 32 | 67 | 33 | 67 | $\Omega_{8\times4\times4}$ |
| 64 | 67 | 67 | 67 | $\Omega_{4\times4\times4}$ |

TABLE 8.6: The width of the **PML layers** present in the compute grids with respect to the **weak** scaling configuration. Applicable to both **seismic modeling** and **seismic migration**.

### 8.1.2 Boundary conditions

In each compute grid, the wave equation solution space is augmented by absorbing layers on the sides and on the bottom. We recall that those are the PML layers that incur extra computations in order to make sure that the wave information are not reflected on the domain edges and do not generate spurious noises that may affect the convergence of the wave equation solver. After partitioning the compute grid of the problem under study, not all the subdomains are subject to PML extra computation. The reader can see on the domain decomposition example in figure 8.3a that only the subdomains with blue partitions are involved in the PML boundary condition (*boundary* subdomains). Those that are fully grayed are referred to as *interior* subdomains and do not participate in solving the PML.

We present in table 8.5 the width of the PML layers applied to the compute grids in a strong scaling configuration. $px$, $py$ and $pz$ refer to the width of the absorbing layer on each direction of each grid dimension ($X$, $Y$ and $Z$ respectively). Note that in the case of the $Z$ dimension, only the bottom of a given compute grid is damped. Besides, we point out that the width of the PML layers in the compute grid used for the seismic modeling is different from that of the PML layers in the compute grid used for the seismic migration, because the discretization steps are different in either case. Table 8.6 shows the width of the damping layers on a weak scaling scenario. The number of PML grid points varies with respect to the number of compute nodes because it depends on the spatial discretization steps, each of which varies as the number of compute nodes increases in a weak scaling scenario (see table 8.4). On the top, the compute grids are subject to a different boundary condition since we apply a free surface boundary condition which means that no extra computation is needed on this area.

## 8.2 Seismic modeling

y In this section, we focus on the seismic modeling application. We present in-depth details about the large scale implementations on the *PANGEA* CPU cluster and on HWAs (GPUs and APUs) based clusters. We show performance numbers on the different architectures and focus on the impact of overlapping the MPI communications with useful computations on the performance.

### 8.2.1 Deployment on CPU clusters: performance issues and proposed solutions

Having in mind that most of the large scale optimizations and techniques that apply to CPU clusters may also apply to hardware accelerators based clusters, we put substantial efforts to address the hotspots and implementation issues related to the large scale CPU implementation of the seismic modeling application. First, we give implementation details about the seismic modeling application. Second, we focus on some MPI pitfalls such as synchronization problems and show how to avoid them. Then, we illustrate how the PML computations incur load imbalance in the application. Finally, we propose an explicit approach to overlap communication with computation and present scalability results, where we distinguish between strong and weak scaling scenarios, with a special emphasis on the impact of the communication-computation overlap on the application performance.

#### 8.2.1.1 Implementation details

In chapter 7, we gave a thorough description and optimization details of the computation driven by the seismic modeling application. Now, we focus on optimizing the MPI communications and underline their impact on the application performance. In this section, we briefly go through the large scale implementation of the seismic modeling application, and we describe the communication pattern, driven by domain decomposition, of the application.

The nature of the wave equation solver, described in section 6.2.2, the pattern of memory accesses of the 3D stencil, presented in section 6.2.2, and the domain decomposition strategy applied to the 3D data volume, discussed in section 8.1.1, determine the communication pattern of our seismic applications. Splitting the computational domain give rise to some *interface* regions, as depicted in figure 8.4, where wave propagation information must be shared among neighbors which interactions between subdomains. The interactions between subdomains consist of data exchanges, i.e. communications, that are performed through the Message Passing Interface (MPI) [12].

| | |
|---|---|
| $nx$ | the grid dimension along the $X$ axis |
| $ny$ | the grid dimension along the $Y$ axis |
| $nz$ | the grid dimension along the $Z$ axis |
| $dd$ | domain decomposition configuration |

TABLE 8.7: Summary of the notations used, in section 8.2, to define the compute grid.

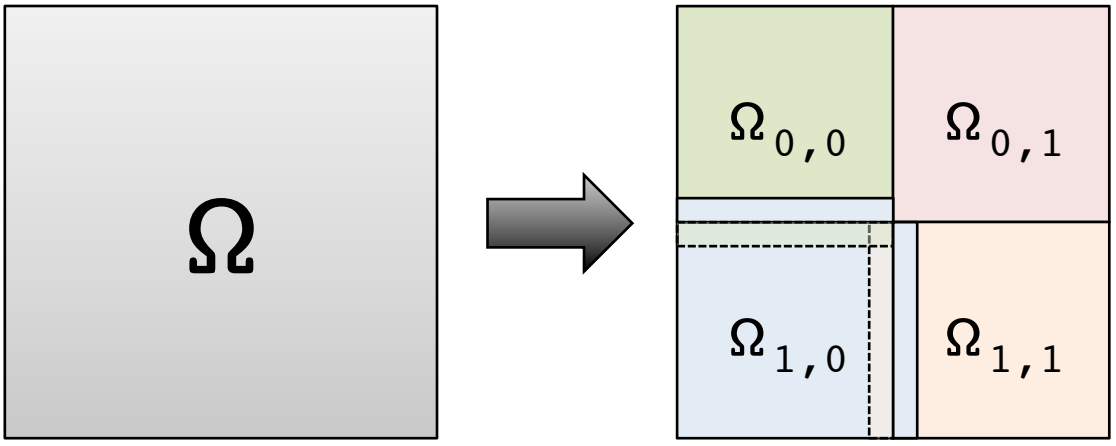



FIGURE 8.4: A 2D illustration of the interface regions (the small rectangles) after domain decomposition. As an example, the solid blue rectangles are wavefield values to be shared with the subdomain $\Omega_{1,0}$.

**Algorithm 8.1** High level description of the multi-node seismic modeling workflow.

```
1:  for t ∈ [0..T] do                          ▷ [0..T] is the simulation time-steps interval
2:      if mod(t, 2) == 0 then                 ▷ t is the time-step index
3:          add_seismic_source(u0, t)          ▷ u0 is the wavefield array at time-step t
4:          exchange_halos(u0)                 ▷ performs MPI communications with neighbors
5:          update_wavefield(u1, u0, t)        ▷ u1 is the wavefield array at time-step t − 1
6:          save_seismic_trace(u1, t)          ▷ described in section 7.1.1 (but not used here)
7:      else
8:          add_seismic_source(u1, t)
9:          exchange_halos(u1)
10:         update_wavefield(u0, u1, t)        ▷ see algorithm 7.1
11:         save_seismic_trace(u0, t)          ▷ not used here
12:     end if
13: end for
14:
```

Before going any further, we introduce in algorithm 8.1 the workflow that characterizes the large scale implementation of the seismic modeling application. It is accompanied by a notation summary in table 8.7 that helps reading the algorithm. The algorithm is an extension to the one node implementation that we described in algorithm 7.1. The workflow is an MPI program performed by one MPI process assigned to one subdomain. It consists of a loop over the simulation time-steps, during each of which the subdomain is updated with respect to the neighboring wave information and to the seismic source function. Note that this workflow corresponds only to one shot experiment.
First, the seismic source function introduces a new perturbation into the compute grid every time-step and let it propagate through the whole domain. This is implemented in *add_seismic_source*() and it concerns only the subdomain that contains the source which depends on the seismic shot configuration.
Then, the boundary sites of each subdomain require to be communicated to one or more adjacent subdomains before any computation. This takes place in *send_ghost_layers*() in the procedure *exchange_halos*(). In addition, all boundary values needed for the simulation sweep (time-step) must be retrieved from the relevant neighboring domains. This is what *receive_ghost_layers*() is for. In order to store this data, each subdomain

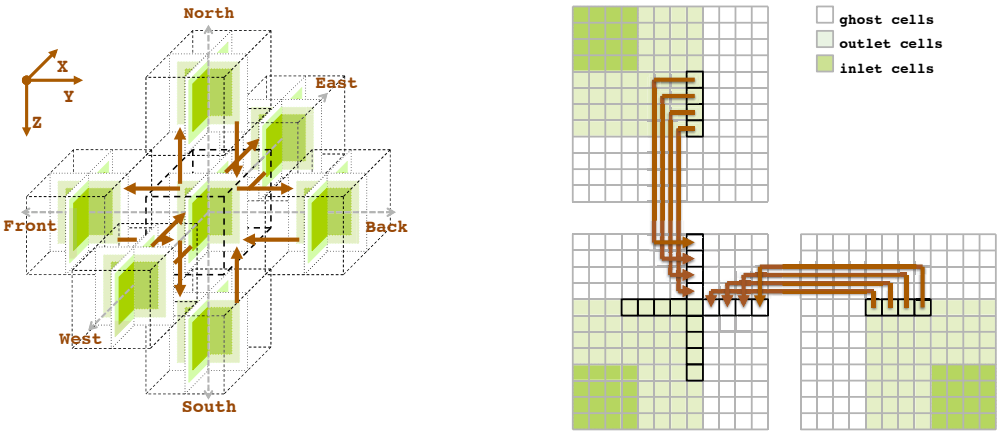
must be equipped with some extra grid points, the so-called *halos* or *ghost layers* (see figure 8.5). In the rest of the document we often refer to the ghost layers as *ghost faces* or *ghost regions*. Each subdomain exchanges ghost faces with its corresponding neighbors and saves them in the ghost area. In figure 8.5a, we give an example of a subdomain, that can be one of the fully grayed subdomains from figure 8.3, where the orange arrows illustrate its MPI interactions with the surrounding subdomains, the dark green squares represent the wavefields that are local to the subdomain and do not have any data dependency with the halos coming from neighbors (referred to as *inlet cells*) and the light green squares (referred to as *outlet cells*) define the wavefields that depend on the wave information collected from the neighbors and stored in the halos (the white squares). At the subdomains faces, the eight-order central finite difference scheme requires a four-cell padding (ghost layers) in each direction of each dimension to correctly propagate the waves. This determines the amount of data to be exchanged between the different subdomains ($g(N, L) = 6dL^2N^{-2/3}$, with $d = 8$, as modeled in section 8.1.1).
Then, each MPI process performs a finite difference stencil computation to update the wavefields that are local to its corresponding subdomain. This is defined in the procedure $fd\_stencil\_compute()$.
Finally, the subroutine $save\_seismic\_trace()$ is meant to periodically save the source wavefield in order to put together the seismic traces resulting from the computations (see section 7.1.1). However, and similarly to the chapter 7, we do not consider the data snapshotting for the seismic modeling as we focus our study mainly about computations and MPI communications (the data snapshotting will be considered in the seismic migration application).

The decomposition proposed in figure 8.3 is a representative example of all the "cubes" domain decomposition schemes used in this study. This scheme incurs MPI communications along the three physical dimensions $X$, $Y$ and $Z$. The MPI communications consist of exchanging three to six four-cell width ghost faces with immediate neighboring subdomains. For example subdomains 0, 3, 12, 15, 48, 51, 60 and 63 (the original domain corners) do exchange three faces with their neighbors. Those on the edges but not corners such as subdomains 1, 2, 13 and 14 issues send and receive four faces from neighbors. The subdomains situated on the external faces but not in the domain edges have to communicate with five neighbors. Finally, the interior subdomains such as 37, 38, 41 and 42 exchange ghost layers with six neighbors as depicted in figure 8.5a. Each subdomain, or more specifically each MPI process, has to issue at most a total of twelve MPI point-to-point communication calls in order to satisfy the data dependencies required by the computation of one physical entity in the solver e.g. velocity field or pressure field.

#### 8.2.1.2 Communications and related issues

In this section we identify the issues related to MPI communications in the seismic modeling workflow. We emphasize on MPI synchronization problems and present workarounds to mitigate their impact on the overall performance of the application.

Before going any further, we need to briefly recall the *synchronous/asynchronous* semantics of MPI communications. As a matter of fact, on the network layer the standard *MPI_Send* function uses two major protocols to send messages based on the message size: *eager* used for relatively short messages and *rendezvous* used for very long messages. In the case of short messages, usually defined as such by MPI implementations when they




(a) Exchanging ghost faces between subdomains

(b) 2D illustration

FIGURE 8.5: Exchanging ghost faces with immediate neighboring subdomains: the white cells represent the extra padding added to the compute grid and the light green (resp. the dark green) cells define the grid points whose computation depends on the surrounding subdomains (resp. the cells that can be updated locally). The axises orientation is an arbitrary choice used for the sole purpose to distinguish the MPI neighbors.

do not exceed 16 to 64KB (see the *MPI eager limit* for more specific information related to a given MPI implementation), the implementation can copy the message directly into an internal buffer and thus send the message out onto the network. This corresponds to the MPI asynchronous communications.
In the case of larger messages, i.e. greater than the eager limit, the internal MPI buffers are not large enough to accommodate the message and the MPI implementation cannot directly copy the message into the buffers. Instead, it switches to the *rendezvous* protocol, which requires the sending process to synchronize with the receiving process before the message is sent out. We talk about synchronous communications. In the context of our seismic modeling application, we summarize the sizes of the messages exchanged between the subdomains, in the *exchange_halos*() procedure, in tables 8.9 and 8.8. The sizes depend on the compute grid dimensions and on the domain decomposition configuration. The notations used in the tables are defined in table 8.7. Note that for the weak scaling scenario, given that the initial compute grid $\Omega_{16\times16\times16}$ has the largest dimension over the y-axis and the smallest over the z-axis, we double the dimensions of the compute grid along the z-axis first, then along the x-axis and finally along the y-axis, as the nodes count increases. Besides, when applying the domain decomposition we split the compute grid first with respect to the y-axis, then to the x-axis and finally to the z-axis.
We can conclude that all the send operations implemented in our seismic applications are synchronous MPI communications where the sender waits for an acknowledgment from the receive side before sending out the data.

The large scale implementation of the seismic modeling application, as described in algorithm 8.1, on the *PANGEA* cluster suffers from numerous MPI related technical issues and is subject to a couple of performance bottlenecks.

First of all, the MPI communications that are implemented in the *exchange_halos*() procedure are further detailed in algorithm 8.2. As an initial approach, we rely on blocking point-to-point MPI communications. Note that since the boundary sites to be exchanged are not necessary contiguous in memory, as one can see in figure 8.5b,

intermediate buffers are used to adequately arrange the data regions in memory prior to the MPI transfers. In the subroutine $pack\_halos\_in\_linear\_buffers()$, the data is packed into intermediate buffers ($bs_n$) prior the send operation. In the subroutine $unpack\_buffers\_into\_halo\_regions()$, the data is copied from the intermediate buffers ($br_n$) into the halo regions.

**Algorithm 8.2** Detailed description of the MPI communications of the seismic modeling application (procedure exchange_halos()).

```
1: procedure exchange_halos(u)                          ▷ u is a wavefield array
2:     for n ∈ {north, south, east, west, front, back} do   ▷ see figure 8.5a
3:         pack_halos_in_linear_buffers(u, bs_n, n)     ▷ n is the MPI neighbors index
4:         MPI_Send(bs_n, n)                            ▷ bs_n is an intermediate buffer
5:         MPI_Recv(br_n, n)                            ▷ br_n is an intermediate buffer
6:         unpack_buffers_into_halo_regions(br_n, u, n)
7:     end for
8: end procedure
```

Given that each subdomain has to issue, at most, six send requests to its immediate neighbors, this may cause deadlocks as all the processes may block on the sending request and no MPI process is available to launch the receiving phase. We recall that the messages are transmitted according to the rendezvous protocol (see tables 8.8 and 8.9) thus all the MPI ranks can be blocked on the send operation (line 4 in algorithm 8.2).

Second of all, MPI synchronous communications imply an implicit "pairwise" synchronization, that is sender and receiver must synchronize in a way that ensures full end-to-end delivery of the data. At the end of the day, this incurs a serialization of the

| *#nodes* | *nx* | *ny* | *nz* | *dd* | size over $X$ | size over $Y$ | size over $Z$ |
|---|---|---|---|---|---|---|---|
| 1 | 1122 | 3443 | 534 | 1x1x1 | – | – | – |
| 2 | | | | 1x2x1 | – | 9.14 | – |
| 4 | | | | 2x2x1 | 14.02 | 4.57 | – |
| 8 | | | | 2x2x2 | 7.01 | 2.28 | 14.73 |
| 16 | | | | 2x4x2 | 3.50 | 2.28 | 7.36 |
| 32 | | | | 4x4x2 | 3.50 | 1.14 | 3.67 |
| 64 | | | | 4x4x4 | 1.74 | 0.57 | 3.67 |

TABLE 8.8: Sizes of MPI **messages (in MB)** with respect to the **strong** scaling test configuration.

| *#nodes* | *nx* | *ny* | *nz* | *dd* | size over $X$ | size over $Y$ | size over $Z$ |
|---|---|---|---|---|---|---|---|
| 1 | 561 | 1721 | 267 | 1x1x1 | – | – | – |
| 2 | 561 | 1721 | 534 | 1x2x1 | – | 4.57 | – |
| 4 | 1122 | 1721 | 534 | 2x2x1 | 7.01 | 4.57 | – |
| 8 | 1122 | 3443 | 534 | 2x2x2 | 7.01 | 2.28 | 14.73 |
| 16 | 1122 | 3443 | 1068 | 2x4x2 | 7.01 | 4.57 | 7.36 |
| 32 | 2245 | 3443 | 1068 | 4x4x2 | 7.01 | 4.57 | 7.36 |
| 64 | 2245 | 6885 | 1068 | 4x4x4 | 7.01 | 2.28 | 14.73 |

TABLE 8.9: Sizes of MPI **messages (in MB)** with respect to the **weak** scaling test configuration.

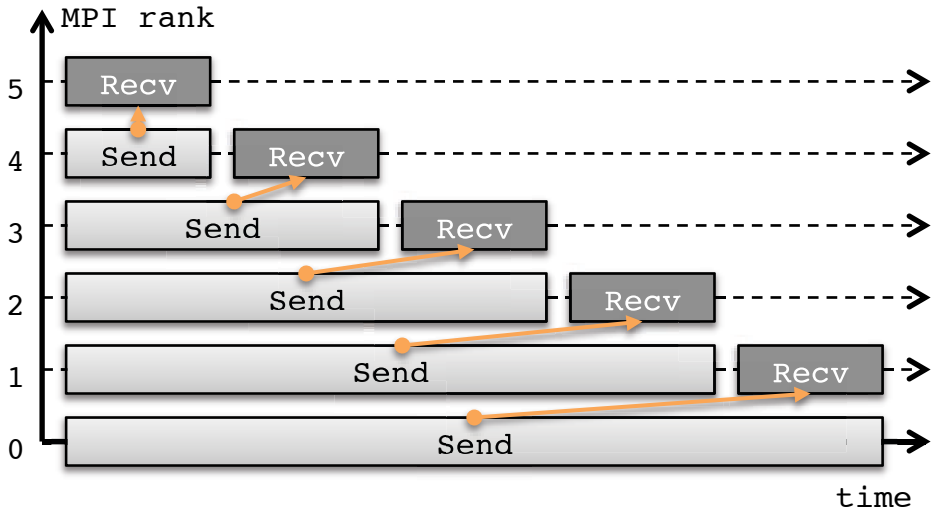



FIGURE 8.6: Timeline view of the linear shift problem where the send operations are synchronous. Extracted from [102].

overall MPI operations of the seismic modeling application, since their order is fixed beforehand as such that first the communications occur along the $Z$ axis, then along the $Y$ axis, and then along the $X$ axis. In figure 8.6, we illustrate how this problem can arise along one dimension when the MPI send operation are synchronous, i.e. using the rendezvous protocol. We refer to this synchronization problem as the *linear shift* problem. The five ranks, in the figure, will transmit messages in serial, one after the other, because no process can finish its send operation until the next process down the chain has finished its receive. The further up the chain a process is located, the longer it blocks on the send operation.

Following, we discuss three solutions that may be used to avoid deadlocks and overcome the linear shift problem described above. The MPI standard [12] offers a wide variety of how to handle point-to-point communications.

In the first solution, we chose to implement the ghost regions exchanges using blocking communications routines that are guaranteed not to deadlock such as *MPI_Sendrecv* which is equivalent to a combination of *MPI_Send* and *MPI_Recv* without temporal dependency. In some literature [102] it is mentioned that internally this call is often implemented as a combination of non-blocking calls and a wait call (the reader can also think about the MPI_Sendrecv_replace [1] that uses less buffers than those needed by MPI_Sendrecv, this routine was not used in the scope of this work). The algorithm 8.3 details the workflow of the new implementation of the *exchange_halos*() routine.

A second solution is to implement the procedure *exchange_halos*() using non-blocking MPI communications. In this case, the six messages are received as soon as possible from different neighbors, and thus deadlocks are avoided. The MPI standard [12] provides nonblocking point-to-point communications routines such as *MPI_Isend* and *MPI_Irecv* that return immediately but, according to the MPI semantics, it is not safe to use the buffers involved in the ongoing requests before calling a wait routine such as *MPI_Wait* or *MPI_Waitall*. This ensures the data consistency on the one hand and that all pending

[1] http://www.mpich.org/static/docs/v3.1/www3/MPI_Sendrecv_replace.html

**Algorithm 8.3** exchange_halos() using deadlock free MPI blocking communications.

```
1: procedure exchange_halos(u)                          ▷ u is a wavefield array
2:     for n ∈ {north, south, east, west, front, back} do   ▷ see figure 8.5a
3:         pack_halos_in_linear_buffers(u, bs_n, n)     ▷ n is the MPI neighbor index
4:         MPI_Sendrecv(bs_n, br_n, n)                  ▷ bs_n and br_n are intermediate buffers
5:         unpack_buffers_into_halo_regions(br_n, u, n)
6:     end for
7: end procedure
```

MPI transfers are terminated on the other hand. The algorithm 8.4 depicts the workflow used in this case.

**Algorithm 8.4** exchange_halos() MPI non-blocking communications.

```
1: procedure exchange_halos(u)                          ▷ u is a wavefield array
2:     for n ∈ {north, south, east, west, front, back} do   ▷ see figure 8.5a
3:         MPI_Irecv(br_n, n)                           ▷ n is the MPI neighbor index
4:     end for
5:     for n ∈ {north, south, east, west, front, back} do
6:         pack_halos_in_linear_buffers(u, bs_n, n)
7:         MPI_Isend(bs_n, n)                           ▷ bs_n and br_n are intermediate buffers
8:     end for
9:     MPI_Waitall(pending_requests)
10:    for n ∈ {north, south, east, west, front, back} do
11:        unpack_buffers_into_halo_regions(br_n, u, n)
12:    end for
13: end procedure
```

Note that using non-blocking communication may suppress the linear shift problem as well, since all the MPI processes are ready to receive messages on the same time. As it will be discussed in section 8.2.1.4.1, using non-blocking communications may suffer from an MPI implementation related issue (the *progress thread* issue) and thus it is not used in the seismic modeling application. As a matter of fact, given that functions such as *MPI_Sendrecv* are built on top of non-blocking MPI communication routines, we do not consider using the first solution either.

As a third solution, in order to avoid deadlocks and the linear shift problem, we may introduce an *alternated blocking communication* scheme. Using non-blocking communicating or deadlock free blocking MPI routines (*MPI_Sendrecv*) are not the only way to avoid these issues when exchanging halos between neighboring subdomains. Another alternative is to use regular blocking communication routines and make sure to interchange the *MPI_Send* and *MPI_Recv* calls on, i.e. all processes that have an even coordinate sum start with a send operation whereas all processes that have an odd coordinate sum with a receive operation, while making sure that there is a matching receive for every send executed (see algorithm 8.5). We consider the axises orientation presented in figure 8.5a, where the neighboring subdomains from the "North" and "South" are lying on the z-axis, the neighbors from the "East" and "West" on the x-axis and those from the "Front" and "Back" are on the y-axis.
Processes that have an even cartesian coordinate sum start one communication phase by exchanging halos with neighbors on the "North", "East" and "Front". Then they

issue a second communication phase where they exchange halos with the neighbors situated in the "South", "West" and "Back". In the contrary, the processes with an odd coordinate sum start the first phase by exchanging the halos with the MPI neighbors on the "South", "West" and "Back". During the second phase they exchange halos with the processes situated on the "North", "West" and "Front".
This scheme may alleviate the network congestion and avoid the progress thread problem related to the non-blocking MPI communications, thus we decided to use this communication pattern in the following experimentations described in the rest of the document.

**Algorithm 8.5** exchange_halos() using MPI alternated blocking communications.

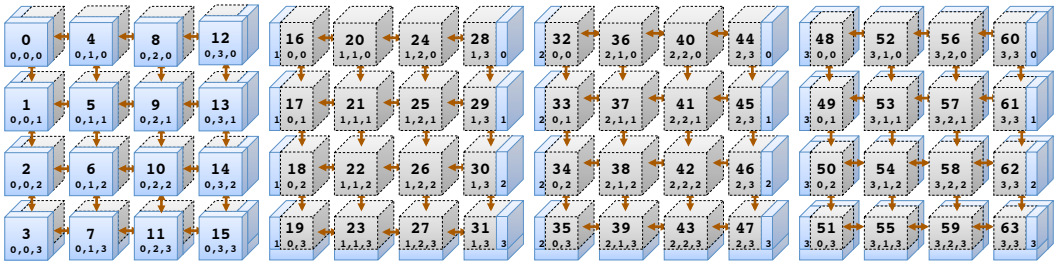
1: **procedure** exchange_halos($u$) ▷ $u$ is a wavefield array
2: **for** $n \in \{north, south, east, west, front, back\}$ **do** ▷ see figure 8.5a
3: pack_halos_in_linear_buffers($u$, $bs_n$, $n$) ▷ $n$ is the MPI neighbors index
4: **end for** ▷ $bs_n$ is an intermediate buffer
5: **if** $mod(coord, 2) == 0$ **then**
6: call MPI_send($bs_{north}$, north)
7: call MPI_recv($br_{north}$, north)
8: call MPI_send($bs_{east}$, east)
9: call MPI_recv($br_{east}$, east)
10: call MPI_send($bs_{front}$, front)
11: call MPI_recv($br_{front}$, front)
12: **else**
13: call MPI_recv($br_{south}$, south)
14: call MPI_send($bs_{south}$, south)
15: call MPI_recv($br_{west}$, west)
16: call MPI_send($bs_{west}$, west)
17: call MPI_recv($br_{back}$, back)
18: call MPI_send($bs_{back}$, back)
19: **end if**
20: **if** $mod(coord, 2) == 0$ **then**
21: call MPI_send($bs_{south}$, south)
22: call MPI_recv($br_{south}$, south)
23: call MPI_send($bs_{west}$, west)
24: call MPI_recv($br_{west}$, west)
25: call MPI_send($bs_{back}$, back)
26: call MPI_recv($br_{back}$, back)
27: **else**
28: call MPI_recv($br_{north}$, north)
29: call MPI_send($bs_{north}$, north)
30: call MPI_recv($br_{east}$, east)
31: call MPI_send($bs_{east}$, east)
32: call MPI_recv($br_{front}$, front)
33: call MPI_send($bs_{front}$, front)
34: **end if**
35: **for** $n \in \{north, south, east, west, front, back\}$ **do**
36: unpack_buffers_into_halo_regions($br_n$, $u$, $n$) ▷ $br_n$ is an intermediate buffer
37: **end for**
38: **end procedure**

#### 8.2.1.3 Load balancing

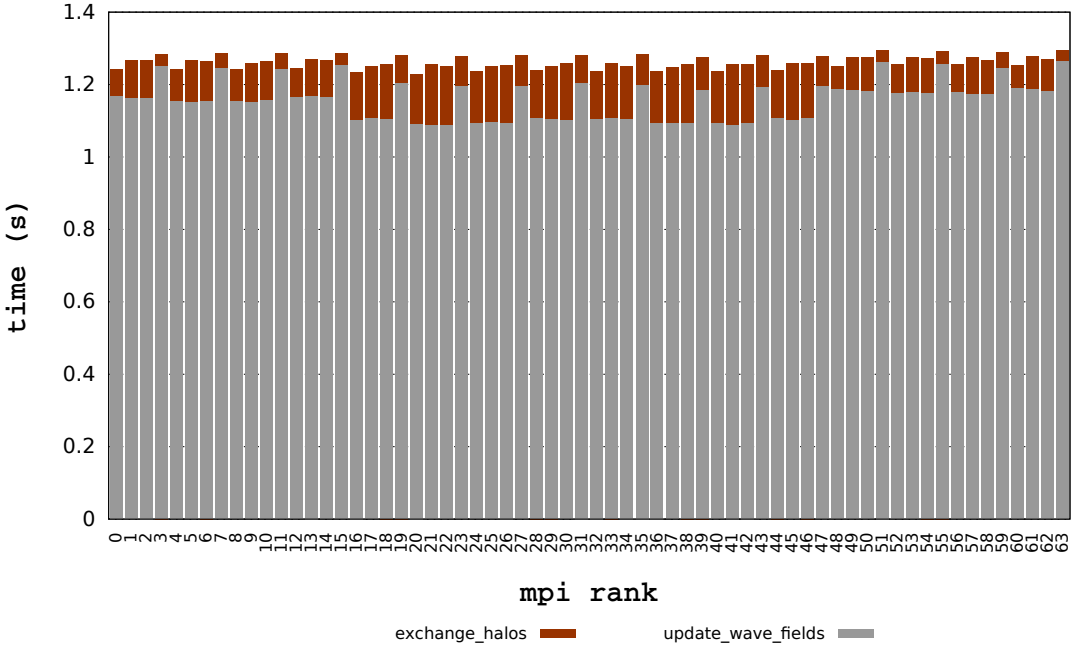
Another potential reason to increase the overhead of implicit MPI synchronizations, is load imbalance. Indeed, when solving the wave equation in a large scale, each MPI process follows the workflow described in the algorithm 8.1 for its subdomain. The computational load differs from one process to another since those that are involved in solving PML regions take longer computation times to update the wavefields. As a matter of fact, updating PML regions incurs more floating point operations, more memory accesses and higher pressure on caches. This creates a load imbalance between the boundary and interior areas. To study the impact of this load imbalance, we measured the execution time of a seismic migration, where we used the velocity grid $\mathcal{V}$ and the numerical parameters detailed in figure 5.2, ran by 64 MPI processes laid out into a 4x4x4 cube domain decomposition as sketched in the figure 8.7a. We detail the times spent by each processor on each step of the workflow in figure 8.7b. The dark orange bars refers to the communication time spent in the *exchange_halos()* routine. The gray bars is the time spent in *update_wavefield()* in the wave equation solver. We notice that the interior areas such as the MPI ranks 20-22, 24-26, 36-38 and 40-42 have the smallest wavefield update time. Whereas the boundary areas can be classified in three categories:

- **subdomains with three PML faces**: those are the domain lower corners that are subdomains 3, 15, 51 and 63. The PML faces (the blue layer in figure 8.7a) are laid along the x, y and z-axis. Sweeping over the y and z-axis when updating these points can induce a high rate of cache misses caused by the irregularity of memory accesses. For this reason those regions have the highest update time.
- **subdomains with two PML faces**: those are subdomains such as 0-2, 12-14, 48-50 and 60-62. We noticed that subdomains that have one face over the two that is laid along the z-axis (7, 11, 55 and 59) are slower and have nearly the same update time than the domain lower corners. We conclude that having PML along the x-axis is almost costless.
- **subdomains with one PML face**: we can group those subdomains into three different bins. Some of these subdomains such as 16-18, 28-30, 32-34 and 44-46 have one PML face to update. Updating these PML faces is slightly slower than updating the interior areas but we can assume they have almost the same update time as the interior areas. This is because, for each subdomain, the PML face is laid along the fastest memory axis, i.e. the x-axis, and thus the memory accesses are contiguous and regular. Subdomains such as 4-6, 8-10, 52-54 and 56-58 have the PML face laid along the y-axis and needs longer time to update the PML cells. Finally the subdomains such as 23, 27, 39 and 43 are the slowest when updating the wavefields since the PML face is mostly laid along the slowest axis which is the z-axis.

In order to further emphasize that the imbalance is caused by the PML extra computations (mainly along the y-axis and z-axis) we remove the PML computation and show the new execution times of the 64 MPI processes in figure 8.8. The update time is regularly the same among almost all the processes. We still can notice some difference specially on the communication times. Indeed, the communication operations are not equal among the processes: for example the MPI rank number 22 issues 6 exchange requests to fulfill its data dependency needed for its computation (this also applies to



(a) MPI ranks layout




(b) Execution time of each MPI process

FIGURE 8.7: Demystification of the load imbalance due to the PML boundary condition, example on a 4x4x4 domain decomposition.

all the interior areas) whereas others such as the edges (1 and 2) issues only 3 requests. So far this may explain the difference in communication times in figure 8.8 although the MPI pairwise synchronization make the overall times look the same. Note that the execution time of the wave equation solver (gray bars) is shorter in figure 8.8 than in that of an interior subdomain (without PML layers) in figure 8.7b. This is because in 8.7b we test whether a wavefield is in a PML region or not, whereas, in 8.8 this conditional test is removed.

To mitigate the load balancing problem, many solutions can be used. *Static partitioning* is one of them: when partitioning the compute grid, subdomains in the boundary areas can be smaller than those in the interior areas and are defined with respect to a performance model based on the computational complexity of each area. A similar approach is *dynamic partitioning*, where PML partitions and interior partitions are dynamically adjusted throughout the simulation time-steps with respect to the ratio between the timed execution of a PML grid point and that of an interior grid point. MPI barriers can be used periodically through-out the simulation time-steps in order to force the MPI processes to synchronize and thus reduce the accumulation of overheads. This solution will be tested in section 8.2.1.4. In addition, other solutions focused on redesigning the

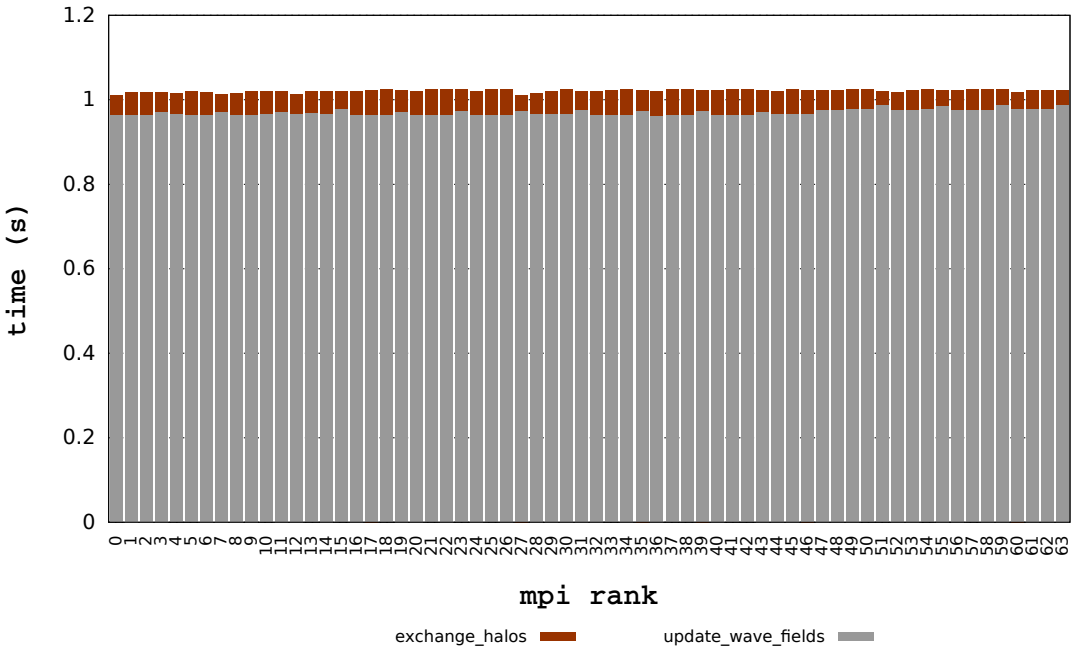



FIGURE 8.8: Execution time of each MPI process (no PML).

MPI barriers and MPI memory fences, usually used by one-sided communications, by reducing their overhead and minimizing their waiting queues. They are then used to periodically synchronize MPI processes and thus alleviate the load imbalance effect.

Another research [76] investigates the behavior of asynchronous PDE solvers where no synchronization is enforced, i.e. the proposed algorithm keeps computing even if necessary information is not yet available. Message arrivals are then modeled as random variables, which brings stochasticity into the analysis. This reduces the MPI communications and thus the MPI synchronization at the expense of a lower accuracy. Furthermore, in [103], the authors categorize the reasons of load imbalance of parallel hybrid applications and enumerate a list of software tools that help users identify them.

#### 8.2.1.4 Communication-computation overlap

We try throughout this section to study the impact of "hiding" the MPI communication costs, as implemented in our large scale wave equation solvers. These communications were found in previous works [22] to be a serious bottleneck to GPU large scale implementations of seismic modeling applications. First, we demonstrate the limits of non-blocking MPI communications. Then, we propose an explicit communication-computation overlap technique. We apply it to the seismic modeling application on the CPU cluster. Finally, we show performance results with considering strong and weak scaling scenarios.

**8.2.1.4.1 Problems of non-blocking MPI communications** According to the MPI standard [12], point-to-point MPI communications can be *blocking* or *non-blocking*. Blocking communications induce a serialization of communication operations and of the

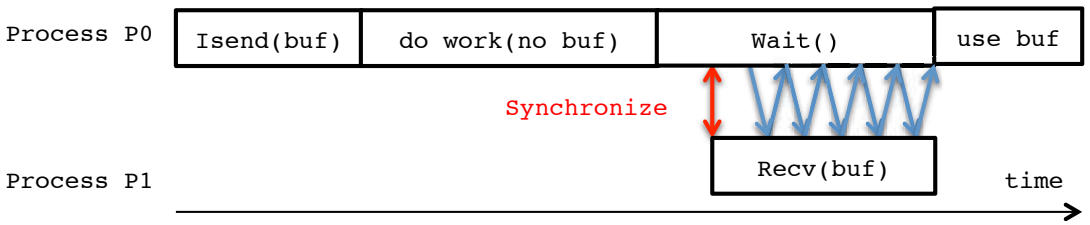



FIGURE 8.9: Possible timeline view of an implementation of a non-blocking synchronous MPI send operation: the sender (P0) and the receiver (P1) might synchronize once P0 have issued an MPI_Wait call. The actual message transmission takes place during the MPI_Wait call.

computation workflow since the blocking MPI calls do not return until the communication is finished. From buffers perspective, this implies that buffers passed to functions such as *MPI_Send()* can be modified, once the functions return, without altering any messages in flight, and one can be sure that the message has been completely received after an *MPI_Recv()*. In the contrary, non-blocking routines, namely *MPI_Isend* and *MPI_Irecv* don't hang and return immediately and are *supposed* to process concurrently with useful computation. Based on the MPI buffer semantics, non-blocking functions return as soon as possible and the programmer has to make sure that all pending MPI operations are fully finished before safely accessing the used memory buffers. This can be done with the help of one of the following MPI routines: *MPI_Wait*, *MPI_Waitall*, *MPI_Waitany*or *MPI_Test*.

One can assume that the non-blocking semantics used by the MPI implementations, by operating in background to progress messages transmission, allows computation and communication to be overlapped straitforwardly. The reality is quite different in most MPI implementations. To try to understand the behavior of an MPI non-blocking communication, it is mandatory to investigate how exactly the message transfers are implemented. We try to give more insights about a generic use case that may include a non-blocking MPI send operation in figure 8.9. The workflow presents a sequence of events where the sender process initiates a non-blocking and synchronous operation. Then it synchronizes with the receiver process before starting the data transfer. At the end, the sender makes sure that the ongoing MPI operations are terminated by means of a wait routine. It is then safe to use the memory buffers that were involved in the non-blocking workflow. It is to be noted that some MPI implementations may delegate the actual data transfer to an auxiliary thread, usually referred to as the *progress thread*, that is internal to the MPI runtime.

Unfortunately, most of the current MPI implementations seem not to use progress threads [104], and can not insure that non-blocking transfers are held in background immediately when the MPI call is launched by the application, but rather perform MPI progress when $MPI\_Wait()$, $MPI\_Test()$, or even any other MPI function is called, exactly as showed in the figure 8.9. In other words, MPI non-blocking communications can not really run in background and be interleaved with useful computations unless functions such as $MPI\_Wait()$ and $MPI\_Test()$ are called right after the $MPI\_Isend$ and $MPI\_Irecv$ to force message transfers to progress inside the MPI library.

We chose to use non-blocking MPI routines in order to test whether the Intel implementation features a progress thread to operate in background. It can be seen in

algorithm 8.1 that, throughout the simulation time-steps, communication and computation are interleaved and thus "serialized". In other words, there is no concurrency between stencil updates and the exchange of ghost layers. This reduces the chance for increasing the parallel efficiency of the seismic modeling application by overlapping MPI communication and computation. The initial algorithm had to be modified in such a way that stencil updates of the interior areas (they are local to the subdomains and have no external data dependency) are performed while copying the boundary areas to intermediate buffers and transmitting them to the halo layers of neighboring subdomains. Then, boundary sites can be updated. The new algorithm is presented in 8.6.

The *exchange_halos*() procedure is split into two routines: *start_exchange_halos*(),

**Algorithm 8.6** The seismic modeling workflow using non-blocking semantics.

```
1:  for t ∈ [0..T] do                          ▷ [0..T] is the simulation time-step interval
2:      if mod(t, 2) == 0 then                 ▷ t is the time-step index
3:          add_seismic_source(u0, t)
4:          start_exchange_halos(u0)           ▷ see algorithm 8.7
5:          update_interior_wavefield(u1, u0, t)
6:          finish_exchange_halos(u0)          ▷ see algorithm 8.7
7:          update_boundary_wavefield(u1, u0, t)
8:          save_seismic_trace(u1, t)          ▷ not used
9:      else
10:         add_seismic_source(u1, t)
11:         start_exchange_halos(u1)
12:         update_interior_wavefield(u0, u1, t)
13:         finish_exchange_halos(u1)
14:         update_boundary_wavefield(u0, u1, t)
15:         save_seismic_trace(u0, t)          ▷ not used
16:     end if
17: end for
```

where each subdomain issues non-blocking and synchronous MPI calls to swap the ghost layers (detailed in figure 8.5a) at its edges with its nearest neighboring subdomains. Depending on its coordinates, each subdomain can have 3, 4 or 6 neighbors. In the routine $finish\_exchange\_halos()$, we insure that all transiting MPI messages are well received and that it is safe to access the ghost regions. As a matter of fact, this routine contains the MPI wait calls. $update\_interior\_wavefield()$ is used to locally update the wavefields whose computation does not depend on the ghost layers. This computation is a good candidate for overlapping. $update\_boundary\_wavefield()$ is called to update the sites on the faces whose computations require the ghost layers already received from the nearest neighbors. We have implemented this algorithm, using the Intel MPI implementation[2] (version 4.1.3), and we show the execution times of the different components of the workflow in figure 8.10a. For the sake of comparison we also tried blocking MPI routines in the same workflow and show the execution times in figure 8.10b. Note that the workflow used to generate figure 8.10b is described in algorithm 8.8. By comparing figures 8.10a and 8.10b, we can notice that most of the communication time (the dark orange bar) is spent in $finish\_exchange\_halos()$ which uses the MPI_Waitall routine. At the end of the day, we may conclude that the use of non-blocking communications, as they are implemented in the Intel MPI library, does not allow the overlap of the local wavefields update with the halos exchange. One possible explanation would be that the

[2] https://software.intel.com/en-us/intel-mpi-library

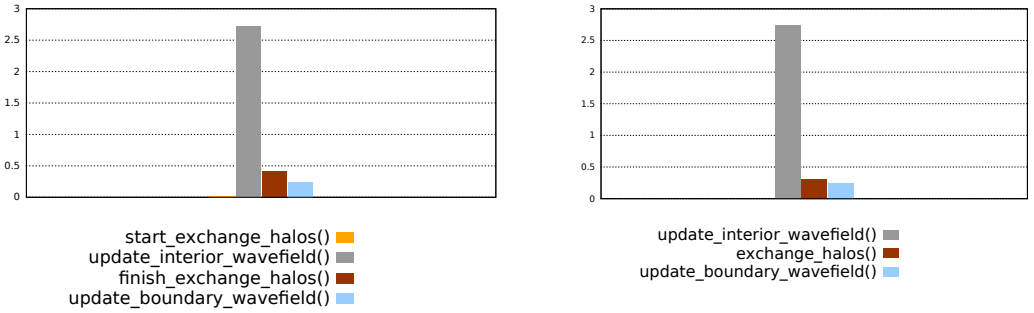
**Algorithm 8.7** Detailed description of the subroutines used in algorithm 8.6.

```
1: procedure start_exchange_halos(u)                          ▷ u is a wavefield array
2:     for n ∈ [neighbors] do                                 ▷ n is the MPI neighbors index
3:         MPI_Irecv(br_n, n)                                 ▷ br_n is an intermediate buffer
4:     end for
5:     for n ∈ [neighbors] do
6:         pack_halos_in_linear_buffers(u, bs_n, n)           ▷ bs_n is an intermediate buffer
7:         MPI_Isend(bs_n, n)
8:     end for
9: end procedure
10:
11: procedure finish_exchange_halos(u)
12:     MPI_Waitall(pending_requests)
13:     for n ∈ [neighbors] do
14:         unpack_buffers_into_halo_regions(br_n, u, n)
15:     end for
16: end procedure
17:
18: procedure update_interior_wavefield(u, v, t)
19:     for z ∈ [4..nz − 5] do                                 ▷ z is the array coordinate in Z
20:         for y ∈ [4..ny − 5] do                             ▷ y is the array coordinate in Y
21:             for x ∈ [4..nx − 5] do                         ▷ x is the array coordinate in X
22:                 laplacian = fd_stencil_compute(v, x, y, z, t)   ▷ see algorithm 6.1
23:                 if (x, y, z) ∈ PML region then ▷ propagate the wave in PML regions
24:                     compute_pml(u, x, y, z, laplacian, t)
25:                 else                                       ▷ propagate the wave in core regions
26:                     compute_core(u, x, y, z, laplacian, t)
27:                 end if
28:             end for
29:         end for
30:     end for
31: end procedure
32:
33: procedure update_boundary_wavefield(u, v, t)
34:     for {z, y, x}! ∈ [4..nz − 5] × [4..ny − 5] × [4..nx − 5] do
35:         laplacian = fd_stencil_compute(v, x, y, z, t)
36:         if (x, y, z) ∈ PML region then
37:             compute_pml(u, x, y, z, laplacian, t)
38:         else
39:             compute_core(u, x, y, z, laplacian, t)
40:         end if
41:     end for
42: end procedure
```

Intel MPI implementation does not provide a progress thread. Using blocking MPI communication would have the same behavior where computation phases are inter-leaved with communication periods. Moreover, even if a progress thread is provided by an MPI implementation, it has the disadvantage to use extra resources without any controlled interactions with the application.

**8.2.1.4.2 Proposed solutions** To address this problem, explicit solutions for overlapping computation and communication are proposed. In [104], a survey of the capability of current MPI implementations to perform "truly" non-blocking point-to-point communications is presented. An OpenMP based solution, where a separate application thread is dedicated for communication, is discussed and applied to a parallel sparse CRS matrix-vector multiplication case study. Similarly, in [187] the authors use an *explicit overlap technique* in a sparse matrix-vector multiplication algorithm. In [223] APSM (Asynchronous Progress Support for MPI) [3] is presented as an extension library, based on the POSIX threading model, that overloads some MPI functions to ensure the progress of non-blocking MPI routines (by means of a progress thread) in background with minimal impact on the performance of code execution in application programs. MT-MPI [198] is another modified multi-threaded MPI library implementation that transparently coordinates with the threading runtime system to share idle threads with the application. The authors demonstrate the benefit of such internal parallelism for various aspects of MPI processing including communication-computation overlap and I/O operations.
At a lower level, the authors of [206] propose a new MPI rendezvous protocol using the RDMA Read. This new implementation is specially designed for InfiniBand [152] featured networks and relies on top of the *Verbs Level API*[153]. It handles MPI message arrivals using software interrupts after which message transfers are launched using the RDMA Read protocol allowing each sender and its receiver counterpart to inherently overlap communication with computation. Another interesting OpenMP approach is proposed in [125], for MPI +OpenMP hybrid applications where some constraints were introduced and have to be verified beforehand in order to know when it is beneficial for a distributed algorithm to overlap communication and computation. Assume that we allocate $P$ OpenMP threads for a given iterative workload and normalize its execution time to 1. Assume that the communication takes time $C$. If no overlap is applied

[3] http://git.rrze.uni-erlangen.de/gitweb/?p=apsm.git;a=summary




(a) With non-blocking routines.

(b) With blocking routines.

FIGURE 8.10: The new seismic modeling workflow: the test was ran on 16 compute nodes with one MPI process per node. Each time is the mean value of 1000 iterations of the wave equation solver measured on the master MPI process.

the work associated with computation is $P * (1 - C)$. If the overlap is considered, one OpenMP thread is explicitly dedicated to communications and the computation is spread across $P - 1$ threads, and the execution time (given that communications are totally overlapped) is:

$$T = \frac{P * (1 - C)}{P - 1}$$

There is speedup only if $T < 1$, that is only if $C > 1/P$. In another hand $C$ has to be less than $P/(2P - 1)$ otherwise the communication time would be dominant. The maximum speedup is then evaluated as $S = \frac{2P-1}{P}$. The authors apply these constraints on a stencil computation use case and show a performance gain of up to 31%.
Other works focus more on efficient MPI/OpenMP hybrid programming models. In [146], the authors give some hints and thumb rules to ensure a successful interaction between the OpenMP programming model and the MPI standard in hybrid applications. They use a set of the NAS benchmarks [4] to emphasize their thesis. The authors also enumerate a list of software tools that can be used to inspect the detailed behavior of OpenMP/MPI hybrid programs. Finally, a useful performance test suite is proposed in [208] to probe the ability of an MPI implementation to overlap communication with computation. It also benchmarks the overhead of multi-threading within an MPI implementation as well as the amount of concurrency in different threads making MPI calls. Note that the last three references concern applications where OpenMP threads are initially used for computation, which is beyond the scope of this work.

We present, in the rest of the section, a description of our approach to overlap MPI communications with computation in the seismic modeling application, as long as a summary of implementation details about thereof. The overlap technique that we propose consists of an explicit OpenMP based solution similar to [104]. Within each MPI process, two OpenMP threads are created using an OpenMP parallel construct. One thread is fully dedicated to exchange halos with up to six immediate neighboring processes. The other thread is responsible for solving the wave equation locally to the subdomain. The two threads are then synchronized implicitly by completing the OpenMP parallel region.

**Algorithm 8.8** The seismic modeling workflow using blocking semantics.

```
1:  for t ∈ [0..T] do                          ▷ [0..T] is the simulation time-step interval
2:      if mod(t, 2) == 0 then                 ▷ t is the time-step index
3:          add_seismic_source(u0, t)
4:          exchange_halos(u0)
5:          update_interior_wavefield(u1, u0, t)        ▷ see algorithm 8.6
6:          update_boundary_wavefield(u1, u0, t)        ▷ see algorithm 8.6
7:          save_seismic_trace(u1, t)                   ▷ not used
8:      else
9:          add_seismic_source(u1, t)
10:         exchange_halos(u1)
11:         update_interior_wavefield(u0, u1, t)
12:         update_boundary_wavefield(u0, u1, t)
13:         save_seismic_trace(u0, t)                   ▷ not used
14:     end if
15: end for
```

[4] http://www.nas.nasa.gov/publications/npb.html

In order to make hybrid (MPI-thread) programming possible the MPI standard provides four levels of thread safety. These are in the form of what commitments the application makes to the MPI implementation.

- $MPI\_THREAD\_SINGLE$: the application is single threaded. All the MPI routines are issued by the application master thread.
- $MPI\_THREAD\_FUNNELED$: the application can have multiple threads but only one thread can issue MPI calls. That thread should be the master thread.
- $MPI\_THREAD\_SERIALIZED$: the application can have multiple threads that may make MPI calls but only one at a time.
- $MPI\_THREAD\_MULTIPLE$: the application can have multiple threads that may make MPI calls at any time.

All MPI implementations support by default $MPI\_THREAD\_SINGLE$. The other levels are activated at runtime by linking against thread-safe MPI libraries and calling the $MPI\_Init\_thread$ function call. Since there was no need that multiple threads have to issue MPI calls concurrently, we chose to use the $MPI\_THREAD\_FUNNELED$ support in order to allow MPI calls inside OpenMP parallel regions by the master thread (thread number 0).

**Algorithm 8.9** The seismic modeling workflow using the explicit communication-computation overlap solution.

```
1: for t ∈ [0..T] do                          ▷ [0..T] is the simulation time-step interval
2:     if mod(t, 2) == 0 then                 ▷ t is the time-step index
3:         add_seismic_source(u0, t)
4:         !$OMP PARALLEL NUM_THREADS(2)
5:         if tid == 0 then                   ▷ tid is the OpenMP thread index
6:             exchange_halos(u0)             ▷ described in algorithm 8.5
7:         else
8:             update_interior_wavefield(u1, u0, t)   ▷ see algorithm 8.6
9:         end if
10:        !$OMP END PARALLEL
11:        update_boundary_wavefield(u1, u0, t)       ▷ see algorithm 8.6
12:        save_seismic_trace(u1, t)                  ▷ not used
13:    else
14:        add_seismic_source(u1, t)
15:        !$OMP PARALLEL NUM_THREADS(2)
16:        if tid == 0 then
17:            exchange_halos(u1)
18:        else
19:            update_interior_wavefield(u0, u1, t)
20:        end if
21:        !$OMP END PARALLEL
22:        update_boundary_wavefield(u0, u1, t)
23:        save_seismic_trace(u0, t)                  ▷ not used
24:    end if
25: end for
```

The algorithm 8.9 depicts the final implementation of the application. The subroutine *exchange_halos*() is that of the alternated blocking communication scheme defined in algorithm 8.5. The message transfers are explicitly progressing by the master thread while computation is taking place concurrently by another thread in the subroutine *update_interior_wavefield*(). In this situation, this means that if the application issues a blocking MPI call the entire process does not block but only the calling thread. As a consequence, it is safe to use blocking MPI functions, namely *MPI_Send* and *MPI_Recv* in order to overlap communication with computation by means of separate OpenMP threads. In other words, no matter which MPI communication routines we use, we can perform MPI communications concurrently with useful computation. We recall that ultimately, we decided to implement the seismic modeling application using the alternated blocking communication scheme, described in section 8.2.1.2, managed by a dedicated OpenMP thread. At the end of the day, we try to overlap the communication with the update of the interior areas. The boundary sites are then updated outside the OpenMP parallel construct in *update_boundary_wavefield*(). Figure 8.11 shows a timeline view of the overlapping algorithm.

**8.2.1.4.3 Performance results** We apply the *communication-computation overlap* solution to the seismic modeling application and we deploy it on the *PANGEA* CPU cluster. We consider using the *3D SEG/EAGE salt model* ($\mathcal{V}$) as the initial dataset for the application. We run a series of scaling tests on 1 to 64 compute nodes, each has sixteen cores and features a 2-way SMT [132]. We distinguish between weak and strong scaling scenarios. We also consider three different MPI processes configurations. The first one is a *one MPI process per node* configuration, where one thread is created to handle communications, and one thread to solve the wave equation. Although this configuration is useless when it comes to sustained performance on the CPU cluster, it may give us some insights about the behavior of the MPI communications on hardware accelerators (GPU and APU) where the MPI communication scheme is very similar and where one thread is dedicated to communication and another is used to manage the accelerator.
The second configuration is *sixteen MPI processes per node*, where we fully utilize the physical cores for computation and where we aim to rely on SMT to manage MPI communications concurrently to computation. This is supposed to be the most natural and efficient configuration when it comes to the application performance. This configuration incurs the usage of a total of 32 threads per compute node when the overlap is activated. Note that the domain decomposition configurations differ from what we have presented in tables 8.3 and 8.4, as more MPI processes are involved. We rather give the domain decomposition parameters, corresponding to this case, in tables 8.10 and 8.11.
The third is an *eight MPI processes per node* configuration. The sole motivation behind using this configuration, is that we are investigating whether SMT would impact

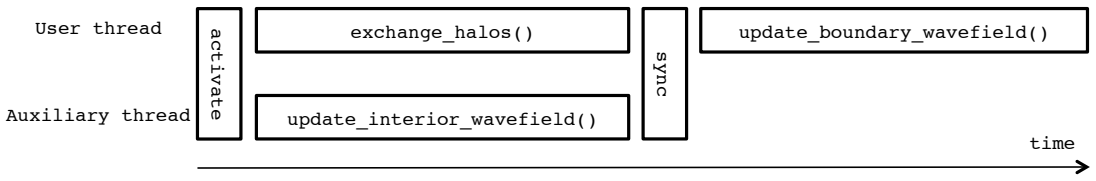



FIGURE 8.11: Timeline view of the explicit computation-communication overlap algorithm used in the seismic modeling application.

| *#nodes* | *#MPI/node* | | | | | | *compute grid* | |
|---|---|---|---|---|---|---|---|---|
| | *8 MPI/node* | | | *16 MPI/node* | | | | |
| | $dd_x$ | $dd_y$ | $dd_z$ | $dd_x$ | $dd_y$ | $dd_z$ | *seismic modeling* | *seismic migration* |
| 1 | 2 | 2 | 2 | 2 | 4 | 2 | $\Omega_{8\times8\times8}$ | $\Omega_{9\times9\times9}$ |
| 2 | 2 | 4 | 2 | 4 | 4 | 2 | | |
| 4 | 4 | 4 | 2 | 4 | 4 | 4 | | |
| 8 | 4 | 4 | 4 | 4 | 8 | 4 | | |
| 16 | 4 | 8 | 4 | 8 | 8 | 4 | | |
| 32 | 8 | 8 | 4 | 8 | 8 | 8 | | |
| 64 | 8 | 8 | 8 | 8 | 16 | 8 | | |

TABLE 8.10: Numerical parameters of the **domain decomposition**, with respect to the **strong** scaling scenario, and with considering **8** or **16 MPI processes** per compute node.

| *#nodes* | *#MPI/node* | | | | | | *compute grid* |
|---|---|---|---|---|---|---|---|
| | *8 MPI/node* | | | *16 MPI/node* | | | |
| | $dd_x$ | $dd_y$ | $dd_z$ | $dd_x$ | $dd_y$ | $dd_z$ | |
| 1 | 2 | 2 | 2 | 2 | 4 | 2 | $\Omega_{16\times16\times16}$ |
| 2 | 2 | 4 | 2 | 4 | 4 | 2 | $\Omega_{16\times16\times8}$ |
| 4 | 4 | 4 | 2 | 4 | 4 | 4 | $\Omega_{16\times8\times8}$ |
| 8 | 4 | 4 | 4 | 4 | 8 | 4 | $\Omega_{8\times8\times8}$ |
| 16 | 4 | 8 | 4 | 8 | 8 | 4 | $\Omega_{8\times8\times4}$ |
| 32 | 8 | 8 | 4 | 8 | 8 | 8 | $\Omega_{8\times4\times4}$ |
| 64 | 8 | 8 | 8 | 8 | 16 | 8 | $\Omega_{4\times4\times4}$ |

TABLE 8.11: Numerical parameters of the **domain decomposition**, with respect to the **weak** scaling scenario, and with considering **8** or **16 MPI processes** per compute node. Applicable to both seismic modeling and seismic migration.

the performance of the application rather than allow overlap possibilities when we fully utilize each node (see previous configuration). Alternatively, we wonder if using half of the physical cores for computation and dedicating the other half to communication would be more beneficial. The tables 8.10 and 8.11 also list the domain decomposition parameters and the compute grids configuration for this test bed.
In order to help read the following figures that summarize the performance results, we depict in the table 8.12 the signification of the notations we used in the figures. We sketch small inverted triangles (see `perfect scaling`) on top the histograms in order to show what an ideal scaling (strong or weak) would look like, which would help the reader to vision the scaling efficiency of our implementations. Note that, in each figure the first reference to the perfect scaling corresponds to the best execution time achieved with respect to the first test case (often the test case "1 node"), in that figure.

| | |
|---|---|
| `in` | the time spent to update the interior region of the subdomain |
| `out` | the time spent to update the boundary regions of the subdomain |
| `comm` | the time spent in MPI communications |
| `max[in,comm]` | the maximum between *comm* and *in* |
| `perfect scaling` | an artificial reference that mimics a perfect strong/weak scaling |
| `noverlap` | results with no communication-computation overlap is applied |
| `woverlap` | results with overlapping communications with computations |

TABLE 8.12: Description of the notations used in the figures illustrating the performance results of the **seismic modeling**.

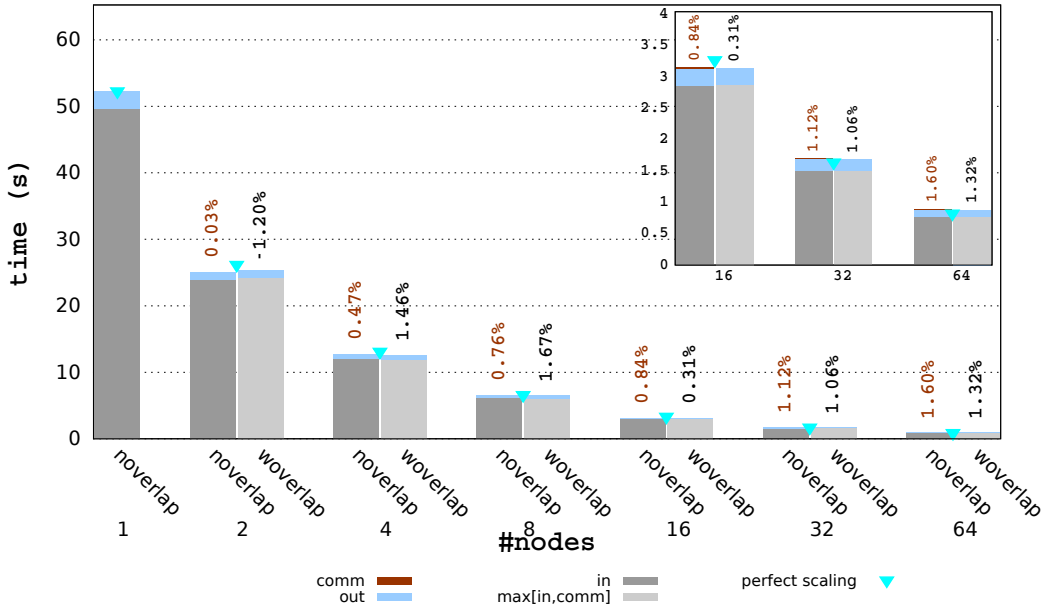



FIGURE 8.12: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **CPU cluster**. The **strong** scaling is considered with placing **1 MPI** process on each compute node. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

**8.2.1.4.3.1 Strong scaling tests** To begin with, we place only one MPI process per compute node, in other words only one subdomain is allocated on a compute node. After an early experiment we noticed a couple of interesting behaviors that we describe as follows. First of all, after running the test on the *3D SEG/EAGE salt model* (the timings are depicted in figure 8.12), we noticed that the communications are partially or totally overlapped in most of all the test instances. However, we observed that this configuration suffers from an unfavorable communication to computation ratio since the communication periods (*exchange_halos*()) are found to be very short compared to the computation time. As a matter of fact, a compute node on the *PANGEA* cluster is a double socket each of which has eight physical cores and features a 2-way SMT. Allocating one MPI process on each compute node implies a long execution time in order to update the wavefields (update_interior_wavefield() and update_boundary_wavefield()) of each corresponding subdomain. Given that the aim of this experiment is to demonstrate the impact of the explicit communication-computation overlap approach on the performance of the seismic application with this process placement configuration, we studied here another dataset with greater communication to computation ratios. We decided to run the seismic modeling application on a smaller data set *Small 3D SEG/EAGE salt model* ($\mathcal{W}$) whose velocity grid information is given in table 8.13. We recall that *3D SEG/EAGE salt model* (the $\mathcal{V}$ dataset) is presented in figure 5.2, which also summarizes its different SEP parameters.

Using the velocity grid $\mathcal{W}$ altered the communication computation ratio as the workload per MPI process is lighter. We can read in table 8.14 a comparison between the percentage of MPI communications, over the overall execution time of some iterations of the solver, when the *3D SEG/EAGE salt model* is used with that when the *Small 3D SEG/EAGE salt model* is used. We can see that the more MPI processes are used the bigger is the communication to computation ratio. It increases up to 30% when the

| *SEP parameter* | *value* |
|---|---|
| n1 | 169 |
| n2 | 169 |
| n3 | 50 |
| d1 | 12.5 m |
| d2 | 40.0 m |
| d3 | 20.0 m |

TABLE 8.13: **Numerical configuration** of $\mathcal{W}$: the **velocity grid** of the *Small 3D SEG/EAGE salt model*. The velocity grid is a cubic section extracted from $\mathcal{V}$ dataset.

small velocity model is used. This will enable us to show the efficiency of our overlapping technique.

Second of all, when we first implemented the overlap technique, we made sure to time all the actions of the two threads, that are active in each MPI domain, separately. We noticed very quickly that the communication time increases throughout the simulation iterations. It is yet another MPI synchronization problem. We mean by communication time a time aggregation that includes the time spent to arrange the data onto linear buffers to make it ready for MPI send operations, the actual message transfer time over the network, the time to unpack the data into the corresponding halo layers and also the time spent in the implicit pairwise MPI synchronization. We also noticed that after few iterations the communication time, measured by the OpenMP master thread, is nearly as important as the interior wavefields update time, measured by the auxiliary OpenMP thread, and sometimes slightly higher. After investigation we found out that this is due to the load balancing problem that affects the seismic modeling application as discussed in section 8.2.1.3. Indeed, this incurs an incremental communication overhead since the MPI processes are less and less synchronized throughout the simulation iterations.

| *dataset* | *#nodes* | *overall* | *communication* | *% communication* |
|---|---|---|---|---|
| $\mathcal{V}$ | 1 | 52.19 | – | – |
| | 2 | 26.70 | 0.008 | 0.03% |
| | 4 | 12.77 | 0.410 | 3.21% |
| | 8 | 6.56 | 0.509 | 7.75% |
| | 16 | 3.14 | 0.160 | 5.09% |
| | 32 | 1.68 | 0.083 | 4.94% |
| | 64 | 0.88 | 0.090 | 10.18% |
| $\mathcal{W}$ | 1 | 0.3792 | – | – |
| | 2 | 0.19 | 0.0016 | 0.85% |
| | 4 | 0.10 | 0.0059 | 5.74% |
| | 8 | 0.055 | 0.0089 | 16.17% |
| | 16 | 0.0193 | 0.0041 | 21.23% |
| | 32 | 0.0114 | 0.0021 | 18.36% |
| | 64 | 0.00625 | 0.0020 | 31.95% |

TABLE 8.14: Percentages of the **MPI communications** of the **seismic modeling** application, with respect to the overall execution times, when using the datasets: $\mathcal{V}$ and $\mathcal{W}$. The **strong** scaling scenario is considered, and each time is a mean value of 1000 iterations of the simulation measured on the **master MPI process**.

To address this problem, the solutions relative to the load imbalance problem presented in section 8.2.1.3 can be applied. We have considered here the use of MPI barriers. They can be considered to alleviate the communication delay effect. We tried to apply a periodic (every 18 to 25 iterations) MPI barrier in order to synchronize all the computation threads without affecting the communication threads. Note that we need the $MPI\_THREAD\_MULTIPLE$ support to do that, because within each MPI process, both the communication thread and the computation thread make calls the MPI library. Although the barrier helped preventing the communication overhead from increasing, the global synchronization introduced by them added another overhead to the table. Ultimately, it was not beneficial to the application and did not solve the problem.

Besides, an important effort was put on investigating how threads, i.e. only one thread which is in fact one process in the case where no overlap is performed or two OpenMP threads when the overlap is applied to the application, are placed on the CPU cores in order to avoid NUMA effects [169] and SMT overheads [184]. Following the Intel MPI implementation recommendation, we tended to pin each communication thread along with its corresponding computation thread in the same CPU socket. This can be done using environment variables provided by the Intel MPI implementation, namely I_MPI_PIN_DOMAIN=socket [131] which ensures that the two OpenMP threads will remain in the same CPU socket. Furthermore, we found out that even in the noverlap case, process pinning was also important. Indeed, the MPI process is being per default placed on the first CPU core (CPU0 or core number 0) of each compute node and the early results showed that the execution time of the local wavefields update was longer in the noverlap case compared to that in the woverlap case. This seemed strange since it consisted of the very same operation and thus we should have obtained the same execution time in both cases. After a thorough examination, it turned out that pinning the single MPI process on a CPU core other than CPU0 gave the execution time expected. A possible explanation of this behavior is that other daemons (MPI daemon, etc.) used by Intel MPI are sporadically using the CPU0. The application process is then concurrently using the CPU core with those latter. As a consequence we set the environment variable I_MPI_PIN_DOMAIN=[000000FE] (using masks) to pin the thread in one socket but outside CPU0. This manipulation solved the problem.

Figure 8.13 shows the execution time of the seismic application in two different scenarios: *noverlap* where the serial approach (see algorithm 8.8) is applied. And *woverlap* where the communication-computation OpenMP based overlap technique is applied (see algorithm 8.9). For woverlap, the light grey stacks represent the maximum time between the $update\_interior\_wavefield()$ time and the communication ($exchange\_halos()$) time. The dark grey legend is the execution time spent to update the local inlet wavefields, the light blue is the time to update the boundary wavefields. The orange legend corresponds to the MPI communication time, i.e. the time spent to exchange halos with neighboring subdomains. The orange percentages on the top of the histogram bars represent the ratio of the MPI communication time to the average iteration time. The black percentages on the top of the histogram bars represent the performance gain (or loss) obtained when communication is overlapped with communication. We recall that these results are obtained after using the *Small 3D SEG/EAGE salt model* with one MPI process per compute node. For each test the slowest MPI process, i.e. the process that has the biggest average iteration time, is taken as reference. Here, the execution times correspond to the strong scaling scenario. It has to be noted that the same legend is used in the rest of the performance charts present in this chapter. As it can be seen in figure 8.13, the communication time ratio becomes more and more important as we increase the

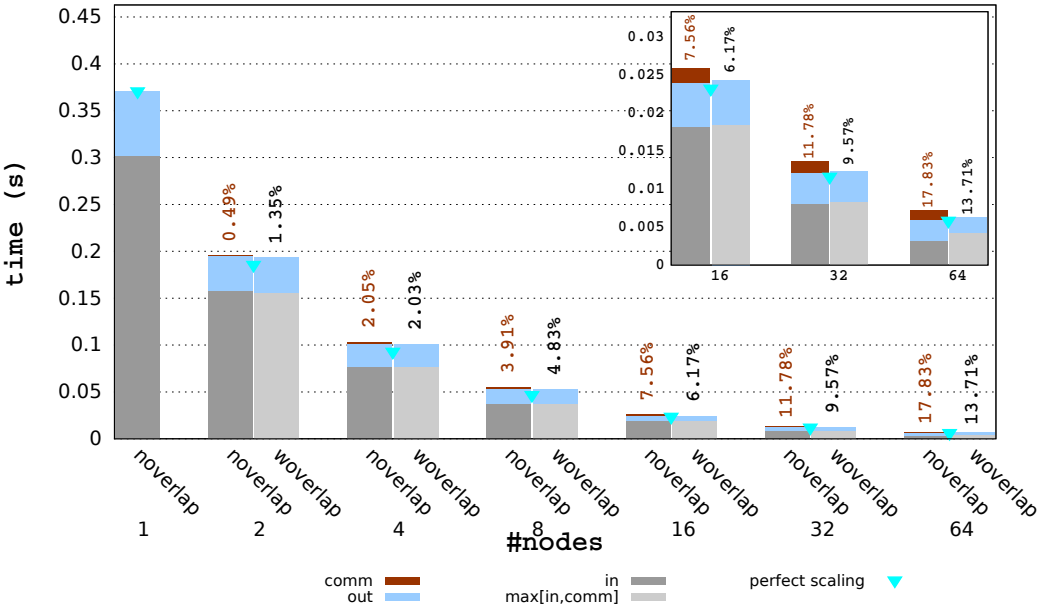



FIGURE 8.13: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **CPU cluster**. The **strong** scaling is considered with placing **1 MPI** process on each compute node. The $\mathcal{W}$ **dataset** is used as input data. Each execution time is the average of 1000 simulation iterations.

number of compute nodes (16 to 64 compute nodes, see figure 8.2). Further, the overlap technique is beneficial to the seismic modeling application performance, specially when a high number of processes is used (16 to 64), as we notice that we can gain up to 15% of the average execution time after each iteration. This gain is achieved when running on 64 compute nodes where we observed that the communication time represents 20% of the overall execution time.
We thus demonstrated that a performance gain is possible on the CPU cluster when communications are overlapped with the computation. We can expect that on GPU and APU clusters, the overlap technique will have positive impact on the performance as well.

Then, in figure 8.14 we increase the number of MPI processes per compute node and show the execution time with and consider two strategies. First, the *eight processes per node strategy*, where we make sure to equally spread the processes across the two sockets of each compute node. This can be done using the I_MPI_PIN_DOMAIN environment variable in the Intel MPI runtime. When the communication is overlapped with computation, 8 other threads are allocated and placed in such a manner that each communication thread runs in the same socket its corresponding computation thread is in. Second, the *sixteen processes per node strategy* where we fully utilize each compute node (each compute node has 16 cores). We recall that in this case we rely on the SMT to use 32 threads per compute node.

We show the detailed execution times, with and without communication-computation overlap, for the strong scaling scenario on 1 to 64 compute nodes. We compare the two strategies, each histogram representing a test instance, i.e. the deployment of the seismic application while combining the overlap technique with the MPI process distribution strategy which implies 4 cases per test instance. In the figure 8.14 each case is referenced as a number, that indicates the number of MPI processes per compute node, juxtaposed

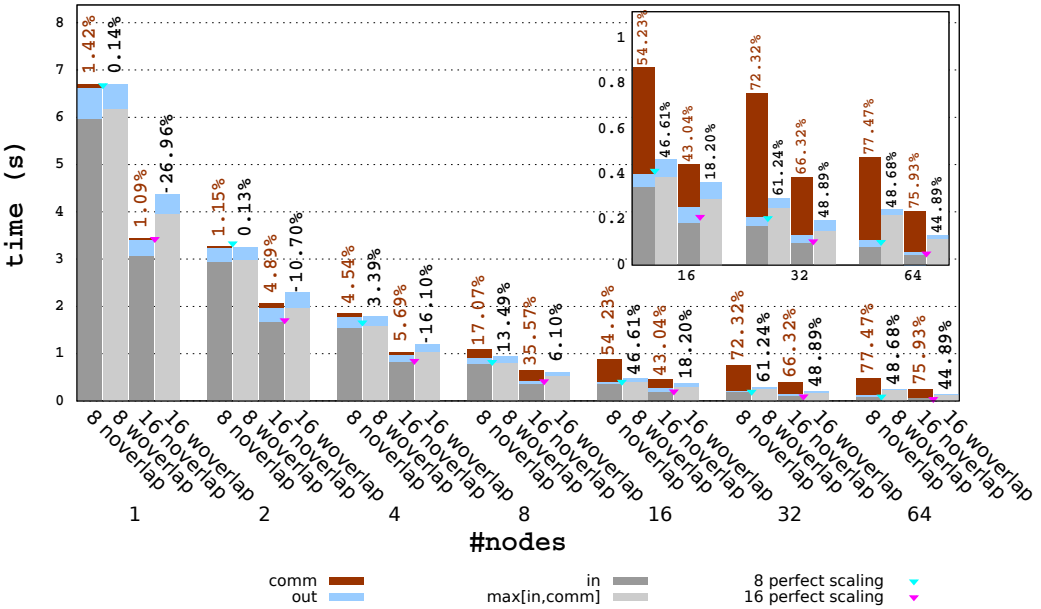



FIGURE 8.14: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **CPU cluster**. The **strong** scaling is considered with placing **8 or 16 MPI** processes on each compute node, making a total number of 8 to 1024 processes. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 simulation iterations.

with a statement that tells whether the overlap technique is applied (woverlap) or not (noverlap).

As expected, we can see that when we do not overlap communication with computation the 16 per node strategy gives the best performance compared to the that with 8 per node. It is also the same for the woverlap case. In some test cases overlapping the communication with computation did not help, specially when 16 MPI processes are used per compute node. For instance we can mind the negative percentages of the cases 2 nodes and 4 nodes. This is because the ratio of communication to computation is very small on the one hand, and because the communication thread (SMT) influences the computation thread on the other. In the contrary, when the amount of communication becomes more important as we increase the number of the used compute nodes (8, 16, 32 and 64) the woverlap version is outperforming the noverlap. We observe up to 44% or 48% of performance gain when applying the explicit overlap technique. The case 32 nodes seems pathological since the performance gain exceeds 50%: we believe that the communication time in the noverlap case is exceptionally high. We can conclude that despite the SMT overhead, using sixteen MPI process per compute node is profitable when the number of nodes is relatively high (more then 32 or 64).

**8.2.1.4.3.2 Weak scaling tests** Following, we explore the impact of the overlap technique on the performance while taking into consideration the weak scaling scenario instead, i.e the problem size is subsequently doubled along with the compute resources. The table 5.7 summarizes the numerical configuration for this test scenario. Similarly to the strong scaling scenario, we start our testings with the one MPI process per compute node configuration. We run the tests on 1 to 64 compute nodes of the *PANGEA* CPU

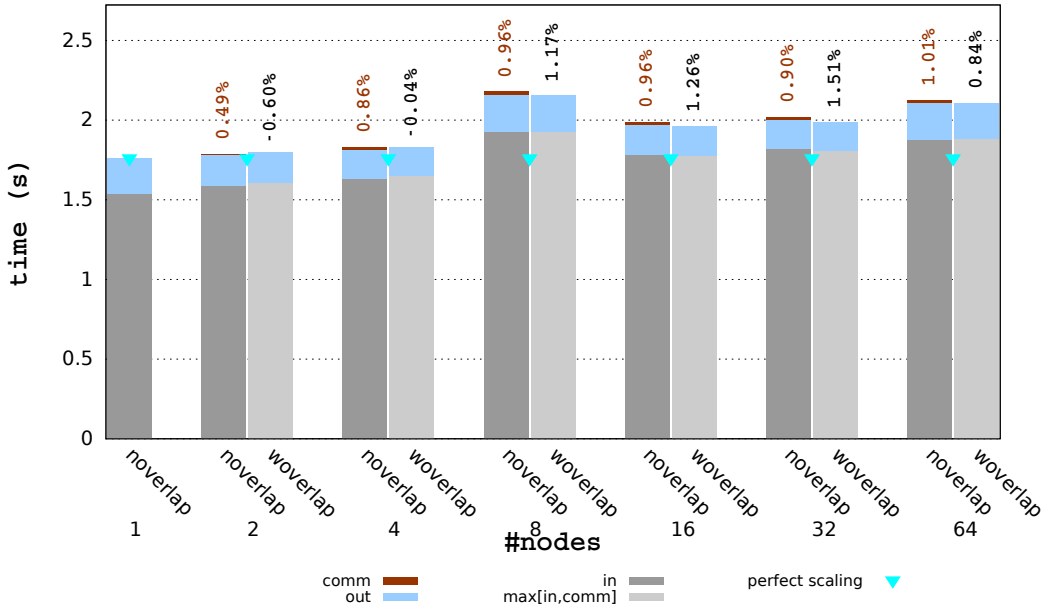



FIGURE 8.15: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **CPU cluster**. The **weak** scaling is considered with placing **1 MPI** process on each compute node. The $\mathcal{W}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

cluster. For this test configuration we use the *Small 3D SEG/EAGE salt model* as initial dataset.

We show the execution times of the weak scaling test on figure 8.15. We compare the woverlap version, where the communication is overlapped with the computation, against the noverlap version. We follow the same process and thread pinning recommendations as those used for the strong scaling scenario, i.e. the communication thread and the computation thread are always running on the same socket, and we avoid using the CPU0 for the noverlap version.

Unlike the previous results, the amount of communication is insignificant compared to the overall execution time (see the orange percentages) and this is applicable even when we increase the number of nodes. This is due to the weak scaling characteristics where the compute workload per node is constant no matter how many nodes are involved in the test. Albeit very small, the communication was partially or totally overlapped with computation for most the test instances (8, 16, 32 and 64). For example we observed a performance gain of 0.84% for the 64 nodes case (where the computation time represents 1.01% of the overall time). It is interesting to say that in this test case we did not notice any MPI synchronization problem as stated above (the dark grey bars are equal to the light grey ones). We can see that the test case with 8 nodes has bigger execution times compared to the others. This behavior remained unexplained since all the execution times of all the test instances are supposed to be similar in a weak scaling context.

Now we consider the two other process placement strategies: eight MPI processes per compute node and sixteen MPI processes per compute node. We run the seismic modeling application on 1 to 64 nodes of the *PANGEA* cluster in a weak scaling fashion. We also compare the woverlap version against the noverlap version to measure the impact

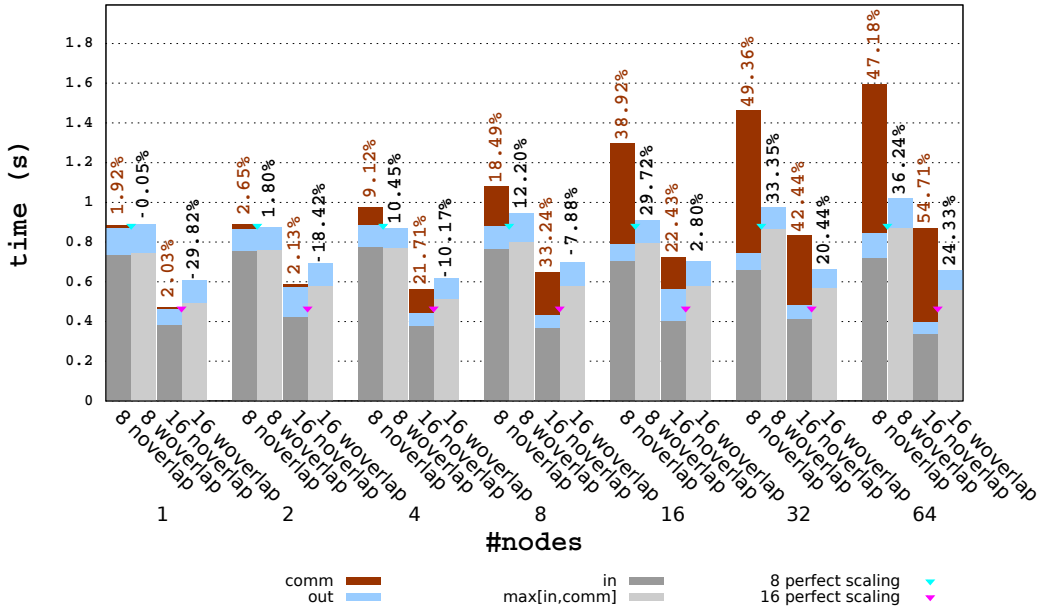



FIGURE 8.16: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **CPU cluster**. The **weak** scaling is considered with placing **8 or 16 MPI** processes on each compute node, making a total number of 8 to 1024 processes. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

of the communication-computation overlap on the application performance. The figure 8.16 summarizes all the timing details. We can conclude that the sixteen per node strategy outperforms the eight per node strategy whether the overlap is activated or not. Besides, the explicit overlap technique enhanced the performance of all the test instances of the eight per node strategy as we can observe a performance gain up to 36%.

However, it was beneficial for the sixteen per node strategy only when a high number of nodes is used (16, 32 and 64 nodes) where we noticed up to 24% of performance gain. When a small number of nodes is used (1 to 8 nodes) the SMT effect is hampering the performance improvement since the overhead caused by the communication thread is more important than the amount of communication itself. That explains the negative gain percentages in figure 8.16.

To conclude on this section, we showed that using an auxiliary thread to handle MPI communications allowed overlapping those communications with the wave equation solver computation. We demonstrated that the communication-computation overlap technique helps enhance the seismic modeling performance in most of the cases. We emphasized that thread pinning played a primary role to help use the compute resources efficiently and orchestrate the communication threads along with the computation threads. We believe that the "one per node" configuration is a quick overview of what we can expect when we deploy the application on hardware accelerators where one thread can be dedicated to communication and another to manage the accelerator. This is studied in the following section.

### 8.2.2 Deployment on hardware accelerators

After having optimized a large scale MPI implementation of the seismic application on the CPU cluster, and having addressed the most important hotspots of its multi-node workflow, we try in this section to adapt the seismic modeling application to the hardware accelerator based clusters, namely to the GPU cluster (see table 5.2) and to the APU (see table 5.3) cluster. To this purpose, we propose an MPI+OpenCL implementation. First, we describe the large scale workflow of the seismic modeling on the GPU and APU clusters. We give an overview about the implementation details, while underlying the changes that we had to introduce in the algorithm and that were driven by the different architectures characteristics and hardware features. Second, we present the OpenCL performance results of the seismic modeling on the discrete GPU cluster, and on the APU cluster with (`zz` data placement strategy (DPS)) and without using the zero-copy memory objects (`cggc` DPS).

#### 8.2.2.1 Implementation details

In this section we describe the large scale workflow of the seismic modeling on the GPU cluster and on the APU cluster. Given that the core of the algorithm, which consists in the wave propagation in an isotropic medium, was thoroughly described in chapter 7, we particularly focus on the communication and on how data is migrated from GPU cores to the CPU and then to the other nodes of the cluster. We assume that the reader had gone through the multi-CPU implementation of the seismic modeling, where we detailed the most relevant aspects and problems related to the large scale implementation of the application (see section 8.2.1), and which remain applicable to the HWAs based multi-node implementations.

When it comes to implementing the seismic modeling on HWAs, a special care has to be put on how the data is managed in terms of traffic between the main CPU and the accelerator (a discrete GPU or an integrated GPU of an APU in our case). Indeed, we have mentioned in section 8.2.1.2, that the ghost faces are copied (or packed) into contiguous intermediate buffers prior to performing the MPI communications with the neighbors of each subdomain (see $pack\_halos\_in\_linear\_buffers()$ in algorithm 8.5). After receiving the data from the neighbors, it is copied back (or unpacked) to the wavefield arrays (see $unpack\_buffers\_into\_halo\_regions()$ in algorithm 8.5). In the rest of the section, we will refer to these operations respectively as *packing* and *unpacking* data. However, given that the computation is held on the GPU cores, the regions to be copied to the neighbors have first to be retrieved from the GPU memory to the main memory in order to send the correct updated wavefield values to the immediate neighbors. In addition, at the end of the computation related to the wave propagation, the updated wavefield array is periodically stored (snapshotting) in order to further reconstruct a sequence of seismic traces at the end of the simulation. Here again, the wavefield array should be first copied from the GPU memory before performing such an operation. Similarly to the multi-CPU implementation and also to the one-node implementation (see algorithm 7.1) of the seismic modeling, we do not consider data snapshotting in the seismic modeling workflow for accelerators, since it is a collective operation which may interfere with the MPI communications. It will be rather discussed in the seismic migration implementations, where the data snapshotting is a local operation to each subdomain.

**Algorithm 8.10** Description of the algorithm of the seismic modeling implementation on the GPU and APU clusters.

```
1: for t ∈ [0..T] do                                   ▷ [0..T] is the simulation time-step interval
2:     if mod(t, 2) == 0 then                          ▷ t is the time-step index
3:         add_seismic_source(u0, t)
4:         exchange_halos(u0)
5:         update_wavefield(u1, u0, t)
6:         dtoh_wavefield(u1)                          ▷ retrieves the data from the GPU (not used)
7:         save_seismic_trace(u1, t)                   ▷ not used
8:     else
9:         add_seismic_source(u1, t)
10:        exchange_halos(u1)
11:        update_wavefield(u0, u1, t)
12:        dtoh_wavefield(u0)
13:        save_seismic_trace(u0, t)
14:    end if
15: end for
16:
17: procedure exchange_halos(u)                        ▷ u is a wavefield array
18:    for n ∈ {north, south, east, west, front, back} do
19:        pack_halos_in_linear_buffers(u, bs_n, n)    ▷ by means of OpenCL kernels
20:        dtoh_halos(u_cpu, bs_n)                     ▷ retrieves halos from the GPU memory
21:    end for                                         ▷ bs_n is an intermediate buffer
22:    ....          ▷ the alternated blocking communication scheme, see algorithm 8.5
23:    for n ∈ {north, south, east, west, front, back} do
24:        htod_halos(br_n, u_cpu)                     ▷ copies halos to the GPU memory
25:        unpack_buffers_into_halo_regions(br_n, u, n)  ▷ using OpenCL kernels
26:    end for
27: end procedure
28:
29: procedure update_wavefield(u, v, t)                ▷ runs on the GPU using OpenCL
30:    each thread (x, y) ∈ nx × ny
31:    for z ∈ [0..nz] do                              ▷ z is the wavefield coordinate in Z
32:        laplacian = fd_stencil_compute(v, x, y, z, t)
33:        if (x, y, z) ∈ PML region then
34:            compute_pml(u, x, y, z, laplacian, t)
35:        else
36:            compute_core(u, x, y, z, laplacian, t)
37:        end if
38:    end for
39: end procedure
```

The changes related to memory manipulation that we had to introduce in the workflow, as to adapt it to HWAs, are highlighted (marked in **bold**) in the algorithm 8.10, where we give a general overview of the seismic modeling workflow on GPU and APU clusters. First, as it has been discussed in the one-node implementation of the seismic modeling in chapter 7, the seismic source term is injected into the computation using OpenCL tasks in the subroutine *add_seismic_source*(). Note that only the subdomain in which the source is placed is concerned by this computation.
Then, as it was seen in the algorithm related to the multi-CPU implementation, each subdomain exchanges the halos, using the alternated blocking communication scheme (see algorithm 8.5), with its immediate neighbors in the subroutine *exchange_halos*(). But, we had to introduce several changes to this subroutine since the wavefield arrays do not necessarily lie on the main memory anymore. For an obvious need of performance enhancement, the data packing, in the multi-GPU implementation as well as in the multi-APU (with the `cggc` DPS), is performed by the GPU by means of up to six OpenCL kernels depending on the domain decomposition configuration. The OpenCL kernels are responsible for copying the ghost layers, that may be broken up into many small regions in the GPU memory, from the most up-to-date wavefield array into contiguous GPU buffers. This is done in the new *pack_halos_in_linear_buffers*() subroutine in algorithm 8.10, which helps prepare the buffers to be copied to the main memory, either via the PCI Express bus or by explicit copies (APU with the `cggc` DPS). It is to be noted that the efficiency of packing and unpacking the data with the help of GPU kernels was already shown in [22]. The *dtoh* notation, used in the algorithm stands for "device to host" and we use it to indicate that a data transfer from the GPU memory to the main memory is taking place. In the contrary, *htod* stands for "host to device" and informs that we have to copy data from the main memory to the GPU memory. Thus, in the subroutine *dtoh_halos*() we copy the packed ghost faces from the GPU memory to the host memory (into the intermediate buffers $bs_n$) prior to the MPI communications. As a matter of fact, this manipulation depends on the used hardware. In the GPU cluster, the subroutine *dtoh_halos*() is synonymous to a memory transfer, via the PCI Express bus, from the discrete GPU of each compute node to the main memory of the same node. For the APU cluster, it rather consists in a simple memory copy from the `c` memory to the `g` memory, when the `cggc` DPS is applied, and in a memory map operation (from an OpenCL standpoint, we talk about the *clEnqueueMapBuffer* function) in order to allow the CPU of an APU to access the wavefield arrays that the integrated GPU was updating, when the `zz` DPS is used. Once packed and brought to the main memory, the ghost faces are exchanged with the neighbors by means of MPI communications. After receiving the data, the opposite workflow is performed. The ghost layers are stored in the main memory (intermediate buffers $br_n$), and then copied to the GPU memory in the subroutine *htod_halos*(). Here again, this operation depends whether we are targeting the GPU cluster or the APU cluster. In the first case, a memory transfer from the CPU memory to the GPU memory via the PCI Express bus is performed. If the APU cluster with the `cggc` DPS is targeted, then an explicit copy from `c` to `g` is issued. In the case where the zero-copy memory objects (the `zz` DPS) are used in the APU cluster, the subroutine merely consists in unmapping the array wavefield from the CPU memory (with the help of the OpenCL function *clEnqueueUnmapMemObject*), in order to allow the GPU cores to operate on the received data. The subroutine *unpack_buffers_into_halo_regions*() is then performed, by the GPU with the help of up to six OpenCL kernels (one for each of the subdomain faces), in order to unpack the contiguous halos back to the right locations into the wavefield arrays.

After that, the computation related to the wave propagation takes place in the subroutine *update_wavefield*(), where a 2D thread grid is deployed on the GPU and the stencil computations are performed on the core region and, if applicable, on the PML regions of each subdomain. Note that we rely on the "multiple kernels" approach to perform computations on the GPU, which had been found in chapter 7 more efficient than using a single OpenCL kernel.
Finally, the updated wavefield array is usually stored on disk (data snapshotting) in the subroutine *save_seismic_trace*() in order to put together a sequence of seismic traces that illustrates the reflecting events of the simulation. Similarly to the multi-CPU implementation, in the HWAs based implementation this operation should be preceded by a memory transfer from the GPU memory to the main memory, or by a memory map operation, which is held in the subroutine *htod_wavefield*(). However, we have mentioned that the data snapshotting is not considered in our seismic modeling workflow, therefore these two subroutines are not used and are rather presented to give to the reader a complete picture about the large scale seismic modeling workflow on HWAs based clusters.

Note that having the GPU perform the packing and the unpacking of the ghost regions, has the advantage to transfer only small buffers back an forth between the CPU and the GPU, otherwise the whole wavefield arrays had to be transferred which would had strongly harmed the performance of the seismic modeling on the HWAs based clusters, especially on the GPU cluster. Note also, that there are other alternatives to what we have presented, such as using the rectangular copies, provided by the OpenCL standard. This solution was tested and was found less efficient than the OpenCL kernels based approach.

Now that we have presented the general workflow of the application, we introduce in algorithm 8.11 the explicit mechanism that we use to overlap the MPI communications with the computations. Relying on this approach had shown performance benefits on the multi-CPU implementation of the seismic modeling (see section 8.2.1.4.3): we aim to investigate whether it is also advantageous to the muti-GPU and to the multi-APU implementations. This technique is based on the OpenMP threading model and was discussed in section 8.2.1.4. The algorithm shows that the main computation is split into two steps. First, the interior (or core) region of the wavefield array is updated (subroutine *update_interior_wavefield*()), and then the boundary region is computed (subroutine *update_boundary_wavefield*()). Thanks to this separation in the computation, the MPI communications are issued by a dedicated OpenMP thread concurrently with another thread driving the core computation that runs on the GPU. In reality this makes a total number of three threads as when creating an OpenCL command queue to run an OpenCL kernel on the GPU, another thread is spawned (as far as the AMD OpenCL implementation is concerned). But, the third thread did not really affect the CPU resources usage.

#### 8.2.2.2 Performance results

In this section we present the performance results of the large scale OpenCL implementations of the seismic modeling on the GPU cluster (see table 5.2) and on the APU cluster (see table 5.3). We consider overlapping the communications with computation and emphasize its impact on the application performance. We distinguish the strong

**Algorithm 8.11** Description of the seismic modeling algorithm on the GPU and APU clusters, with overlapping communications with computations.

```
1: for t ∈ [0..T] do                                  ▷ [0..T] is the simulation time-step interval
2:     if mod(t, 2) == 0 then                          ▷ t is the time-step index
3:         add_seismic_source(u0, t)
4:         !$OMP PARALLEL NUM_THREADS(2)
5:         if tid == 0 then                            ▷ tid is the OpenMP thread index
6:             exchange_halos(u0)                      ▷ described in algorithm 8.10
7:         else
8:             update_interior_wavefield(u1, u0, t)
9:         end if
10:        !$OMP END PARALLEL
11:        update_boundary_wavefield(u1, u0, t)
12:        dtoh_wavefield(u1)                          ▷ not used
13:        save_seismic_trace(u1, t)                   ▷ not used
14:    else
15:        add_seismic_source(u1, t)
16:        !$OMP PARALLEL NUM_THREADS(2)
17:        if tid == 0 then                            ▷ tid is the OpenMP thread index
18:            exchange_halos(u1)
19:        else
20:            update_interior_wavefield(u0, u1, t)
21:        end if
22:        !$OMP END PARALLEL
23:        update_boundary_wavefield(u0, u1, t)
24:        dtoh_wavefield(u0)
25:        save_seismic_trace(u0, t)
26:    end if
27: end for
28:
29: procedure update_interior_wavefield(u, v, t)      ▷ runs on the GPU
30:    each thread x, y ∈ [4..nx − 5] × [4..ny − 5]
31:    for z ∈ [4..nz − 5] do                          ▷ z is the wavefield coordinate in Z
32:        laplacian = fd_stencil_compute(v, x, y, z, t)
33:        if (x, y, z) ∈ PML region then
34:            compute_pml(u, x, y, z, laplacian, t)
35:        else
36:            compute_core(u, x, y, z, laplacian, t)
37:        end if
38:    end for
39: end procedure
40: procedure update_boundary_wavefield((u, v, t)     ▷ runs on the GPU
41:    each thread x, y ∈ [0..3] ∪ [nx − 4..nx − 1] × [0..3] ∪ [ny − 4..nx − 1]
42:    for z ∈ [0..3] ∪ [nz − 4..nz − 1] do
43:        laplacian = fd_stencil_compute(v, x, y, z, t)
44:        if (x, y, z) ∈ PML region then
45:            compute_pml(u, x, y, z, laplacian, t)
46:        else
47:            compute_core(u, x, y, z, laplacian, t)
48:        end if
49:    end for
50: end procedure
```

| | |
|---|---|
| `pack` | the time spent in packing the halos |
| `unpack` | the time spent in unpacking the halos |
| `d-h-comm` | the time spent in the subroutines *dtoh_halos*() and *htod_halos*() |

TABLE 8.15: Description of the additional notations used in the figures illustrating the performance results of the seismic modeling on HWAs based clusters.

scaling scenario from the weak scaling scenario. For each case we present the performance numbers obtained on he GPU cluster, on the APU cluster when considering the `cggc` DPS, and on the APU cluster with the zero-copy memory objects (the `zz` DPS).

We deploy the multi-node implementation on *DIP* and *RDHPC* clusters, running the simulation throughout 1000 time-steps, and show the execution time of the slowest MPI process in each test case. We recall that the maximum number of nodes of the APU cluster is 16, therefore our scaling tests involve a number of nodes that only ranges from 1 to 16. Note that in the case of discrete GPUs the memory objects are pinned in order to improve the bandwidth of the PCI Express bus (Gen 3) while transferring the data between the CPU and the GPU or vice versa. We recall that the data snapshotting is not considered, thus the presented times do not include any I/O operation. In order to help read the following figures that summarize the performance results, in addition to the table 8.12, we summarize in the table 8.15 the signification of the additional notations we used in the figures. We recall that `perfect scaling` is an indication, based on the best execution time of the first test case in each figure, that is only used to evaluate the scaling, on the HWAs based clusters, of our multi-node implementation of the seismic modeling (see table 8.12).

**8.2.2.2.1 Strong scaling tests** We ran the seismic modeling implementation on the compute grid $\Omega_{8\times8\times8}$ while using the *3D SEG/EAGE salt model* ($\mathcal{V}$) as an input data. We consider the case where we apply the algorithm 8.10 which does not involve overlapping MPI communications with computation (`noverlap`), and the case where we make use of the explicit communication-computation overlap technique explained in algorithm 8.11 (`woverlap`). Due to the memory limitations related to the memory capacity of the discrete GPUs (12 GB at most) and to the OpenCL specification (version 1.2), which states that the maximum amount of memory that can be allocated at once is 1/4 the total size of the global memory, the strong scaling case is reduced to two test cases (8 and 16 nodes) for the GPU cluster and the APU cluster (with the `cggc` strategy). However, the `zz` DPS allows to consider the strong scaling on the APU cluster from 1 to 16 nodes.

The figure 8.17 shows the performance numbers of the strong scaling test applied to the seismic modeling application on the GPU cluster. We emphasize the impact of the computation-communication overlap on the performance.
The reader can see that using an auxiliary thread on each compute node, concurrently with the thread that drives the discrete GPU, allows up to nearly 14% of performance enhancement. We recall that when the communication-computation overlap technique is used, two separate OpenCL kernels are deployed; one to update the interior grid points (the routine $update\_interior\_wavefield()$), the other to compute the values of the grid points on the domain boundaries (the routine $update\_boundary\_wavefield()$). However, only one OpenCL kernel is used if we do not rely on the communication-computation overlap. Therefore, we only distinguish between the `in` and the `out` times (see table

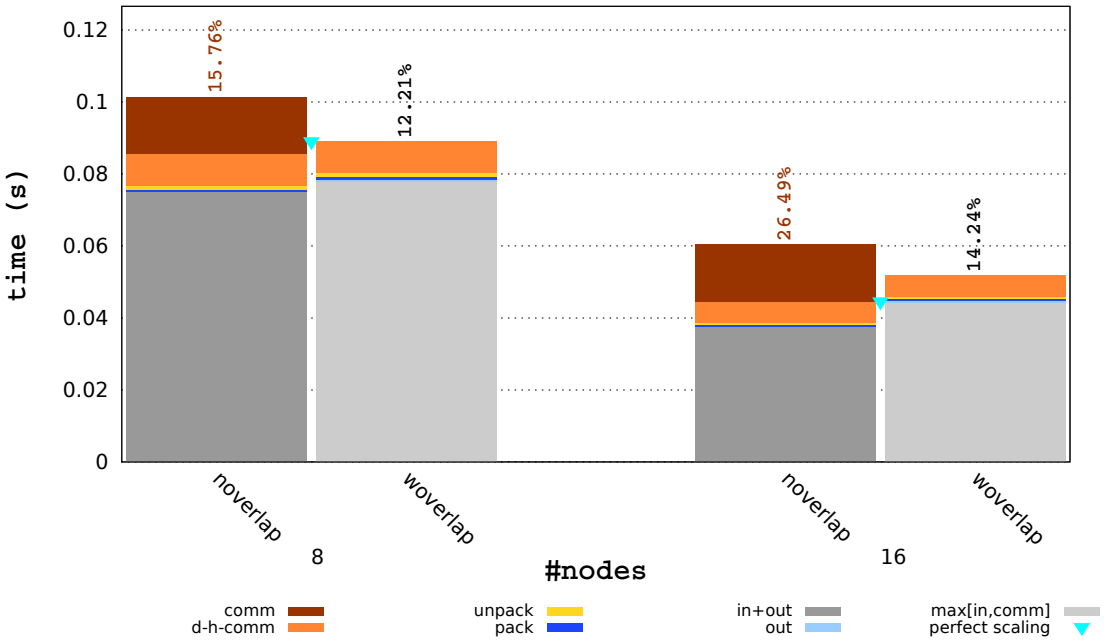



FIGURE 8.17: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **GPU cluster**. The **strong** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

8.12) only in the `woverlap` case. We recall that the communication time (`comm`) does not include the times of packing the boundary grid points onto the temporary buffers during the MPI communication as well as the times of unpacking the received data into the wavefield arrays (as it was the case for the multi-CPU implementation). The packing and the unpacking times are presented separately and we notice that they are not overlapped with computation in the `woverlap` case. As a matter of fact, overlapping these two operations means also overlapping the data transfers through the PCI Express bus. However, due to data dependency this is not possible since the memory location, which is the destination of the PCI transfer, is being manipulated at the same time by the OpenCL kernel in charge of updating the wavefield arrays. This is presented as an undefined behavior in the OpenCL standard (version 1.2).
In addition, the test case "16 nodes" shows that mainly the PCI Express bus overhead hinders performing a near to perfect strong scaling. Note that the difference between `in` and `max[in,comm]` in the test case "16 nodes", is due to that the communications time (which also includes the MPI synchronization overhead) surpasses the computations time when using the overlap technique, as this particular process has to synchronize with the other MPI processes subject to longer computations (to update larger PML regions for instance).

The figure 8.18 shows the performance numbers of the seismic modeling for the strong scaling scenario on the APU cluster with the `cggc` DPS. Similarly to the GPU cluster, the test is limited to 8 and 16 nodes. Besides, in this case the device to host and host to device memory transfers overhead is reduced (see `d-h-comm`) as the data is now transferred through the Garlic bus rather than through the PCI Express bus in the case of discrete GPUs. We notice that overlapping the MPI communication with computation helped to enhance the overall performance up to 19% here again, offering

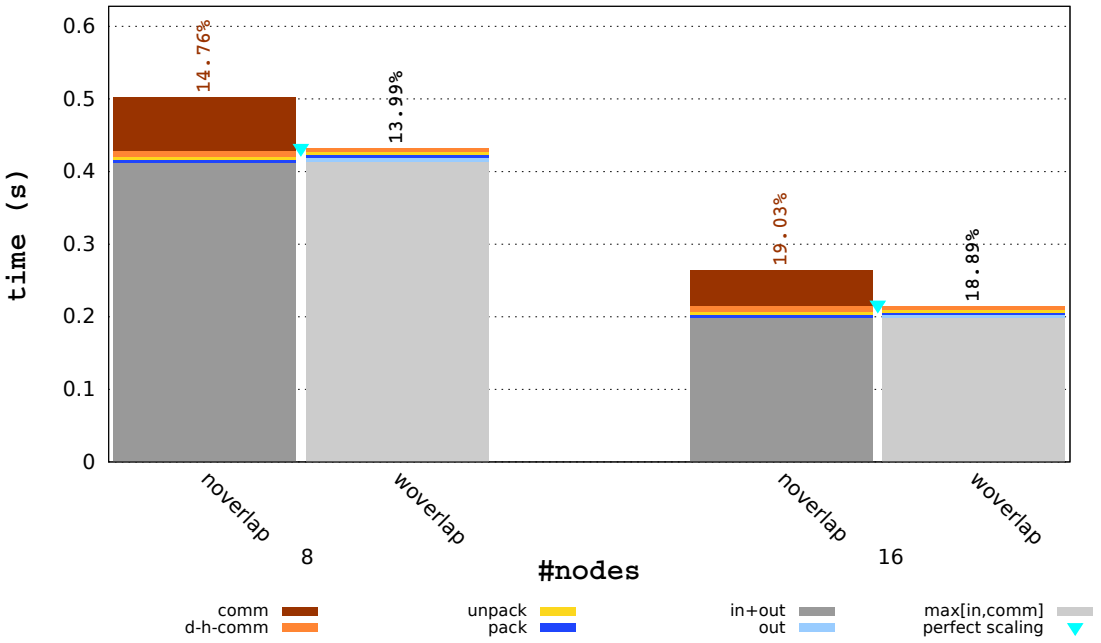



FIGURE 8.18: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **APU cluster** with the **cggc** DPS. The **strong** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

a near to perfect strong scaling thanks to the reduced overhead of data copies from the integrated GPU to the CPU and vice versa.

Following we present, in figure 8.19, the performance results of the seismic modeling on the APU cluster when using the zero-copy memory objects (the `zz` DPS). In this case, one can notice that the overhead `d-h-comm` is almost non existent thanks to the zero-copy memory objects. Besides, the MPI communications are almost fully overlapped with computations in the cases 2, 4, 8 and 16 nodes, resulting in a performance improvement of up to 14%, and to a near to perfect strong scaling. However, relying on the `zz` DPS delivers only half the performance achieved on the APU cluster when using the `cggc` DPS. It is to be noted that we found the amount of MPI communications in the test cases "2 nodes" and "4 nodes" abnormally low. Besides, we point out that using the zero-copy memory objects has reduced the performance approximately to half, as compared to using the `cggc` DPS.

In general for both the GPU and APU clusters, the percentages of the MPI communications time with respect to the overall time of the seismic modeling, are higher in comparison to the multi-CPU implementation, which is due to the fact that the computations are much more accelerated with HWAs rather than with CPUs. Therefore, it is mandatory to optimize the MPI communications on the HWAs based clusters, and to overlap them with the computations on GPUs, whenever it is possible, especially that the CPUs in GPU based systems are often idle and can be charged with such a task.

**8.2.2.2.2 Weak scaling tests** Now, we consider running the seismic modeling implementation on the compute grids detailed in table 5.7, with respect to the weak scaling scenario. Given that the smallest compute grid in this test bed, i.e. $\Omega_{16\times16\times16}$,

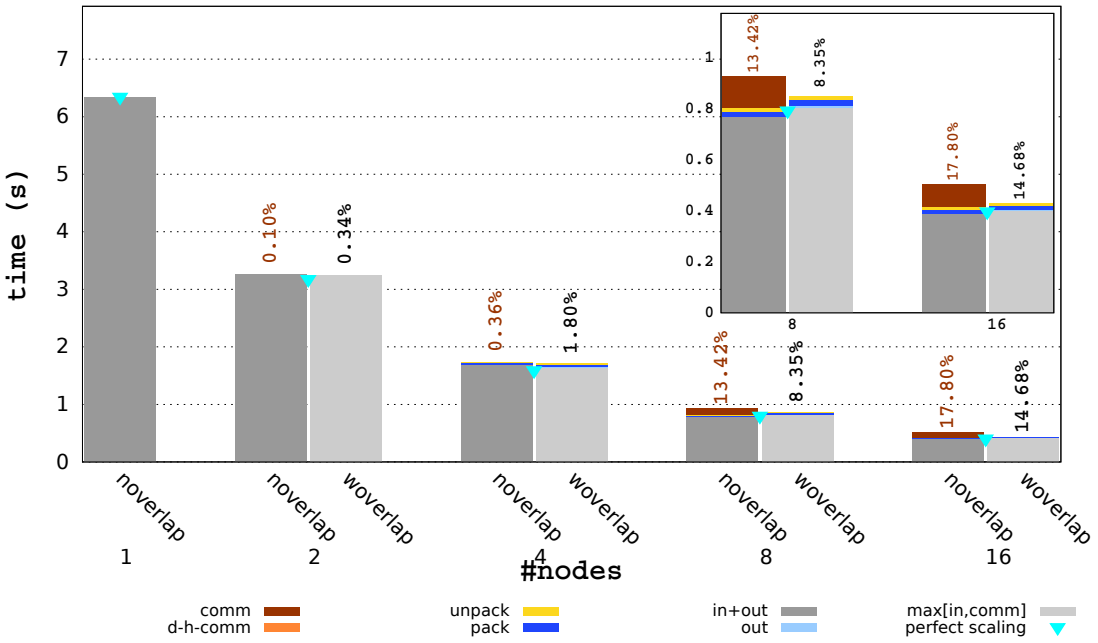



FIGURE 8.19: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **APU cluster** using the **zero-copy** memory objects. The **strong** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

fits in one GPU based compute node as well as in one APU based compute node, the weak scaling scenario involves a number of compute nodes that ranges from 1 to 16.

The figure 8.20 depicts the performance numbers obtained on the GPU cluster. The reader can notice that overlapping the MPI communications with OpenCL computations on the GPU is also beneficial in the weak scaling scenario, as we observe up to 12% of performance gain. It is also to be noted that the communications time gets bigger as we increase the number of nodes. Proportionally to the communications time, the PCI transfers (`d-h-comm`), the data packing (`pack`) and unpacking times (`unpack`) increase as the number of nodes is doubled. Moreover, the MPI communications are less efficiently overlapped in the test case "16 nodes", since this particular MPI process (we recall that the times depicted in the figures correspond to the slowest MPI rank in each test configuration) synchronizes with other processes that take longer time in computations (due to the PML layers) and spend less time in MPI communications, which creates an additional overhead. Besides, we can see that the overhead incurred by the memory transfers via the PCI Express bus has a negative impact on the performance, as it partially harms the parallel efficiency of the seismic modeling on the GPU cluster (see `perfect scaling`). Finally, one can see that the computation time in the test cases "8 nodes" and "16 nodes" are lower than that in the other cases. This is only due to the fact that we are measuring the times of the slowest MPI process in each configuration, and that depending on the number of PML faces that each subdomain contains, the `in` time may vary from an MPI process to another.

In the figure 8.21, we illustrate the OpenCL performance results of the seismic modeling on the APU cluster, using the `cggc` DPS and considering the weak scaling scenario. At first glance, one can see that the overhead due to the memory traffic between the CPU and the GPU is reduced (in proportion with respect to the overall

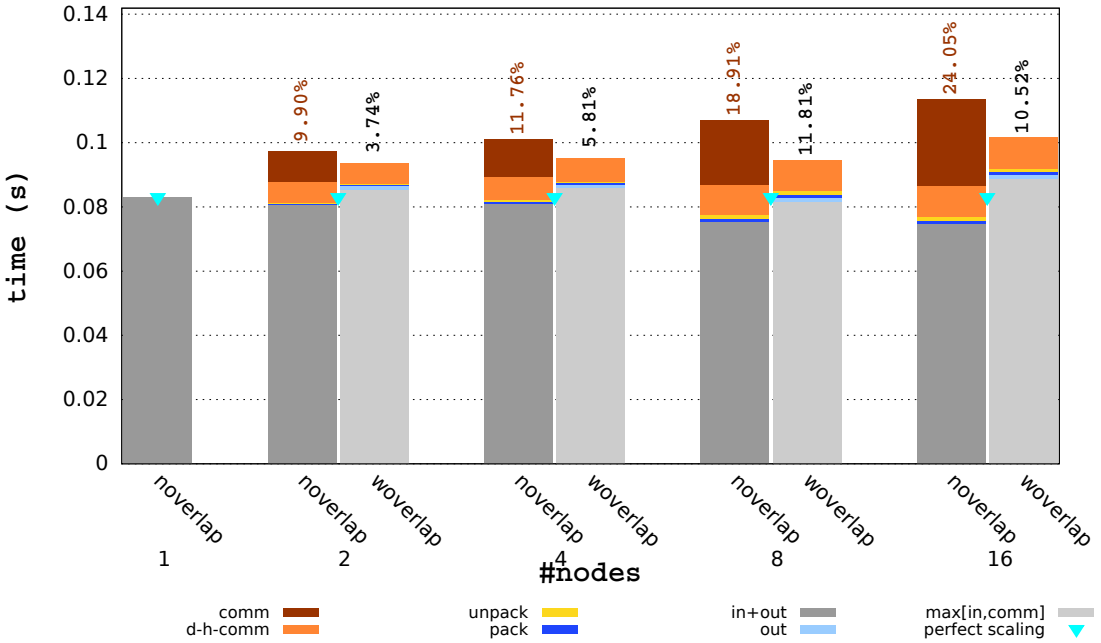



FIGURE 8.20: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **GPU cluster**. The **weak** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

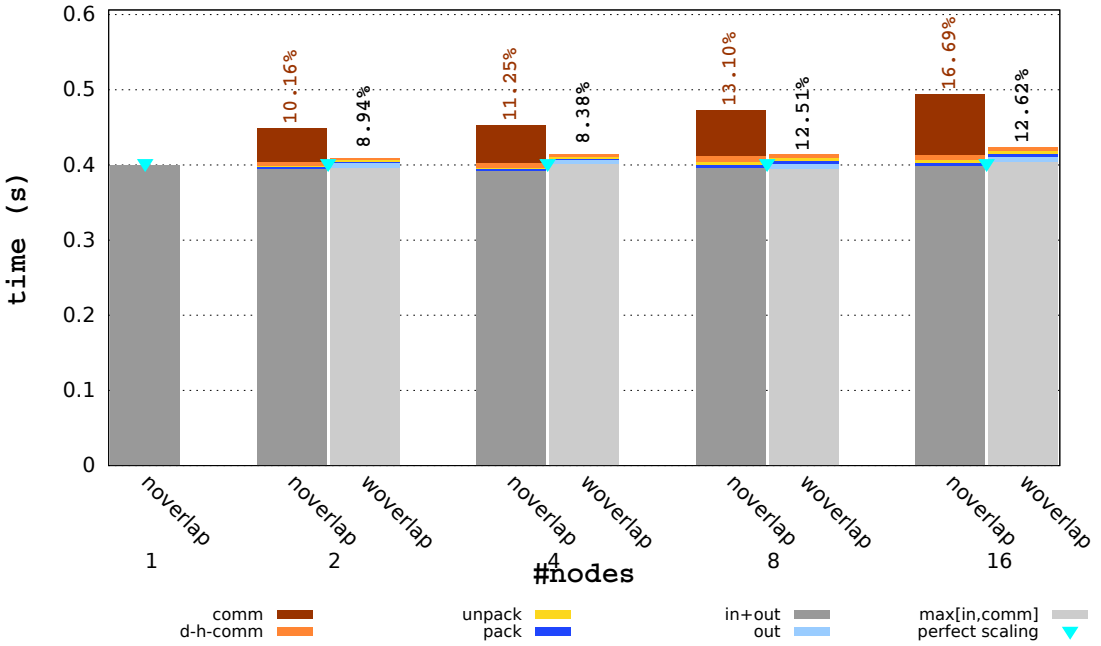



FIGURE 8.21: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **APU cluster**, with the **cggc** DPS. The **weak** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

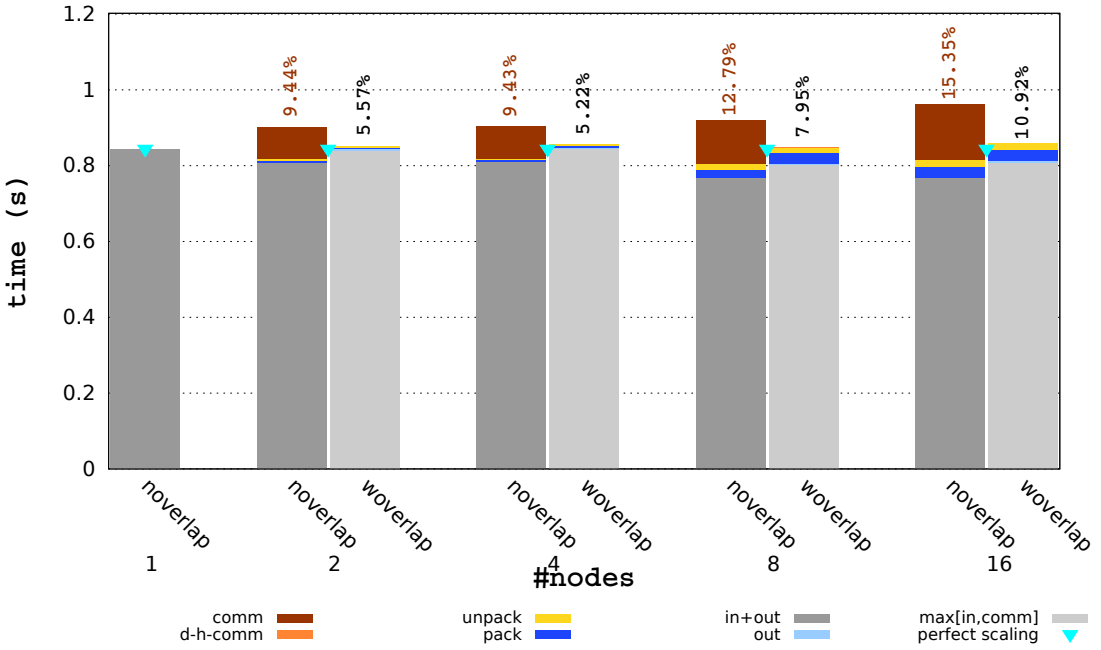



FIGURE 8.22: Performance impact of the communication-computation overlap on the **seismic modeling** application on the **APU cluster** using the **zero-copy** memory objects. The **weak** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

time). Besides, the communication-computation overlap technique offered a significant performance gain (up to 12%) as the MPI communications were almost fully overlapped, even more efficiently in comparison with the weak scaling on the GPU cluster. This had led to a good scaling (near to perfect). Finally, we present in figure 8.22 the performance results for the seismic modeling on the APU cluster with using the zero-copy memory objects. Similarly to the strong scaling scenario, we notice that the overhead due to the data copies between the GPU and the CPU is removed thanks to the zero-copy memory objects. Besides, overlapping the communications improved the performance, and that the gain reached up to 11%. However, we can see that the `pack` and `unpack` times are increasing as we double the number of nodes. This is explained by the fact that the more nodes we use the bigger is the compute grid and the more ghost layers are exchanged between neighbors. Besides, in the case of the APU cluster with the `zz` DPS, the data packing and the data unpacking are performed by the CPU, as opposed to the implementation on the GPU cluster where they are performed, with the help of OpenCL kernels, on the GPU. By proceeding as such, the packing and unpacking subroutines are found less efficient on the CPU (especially on the CPU of the APU which has a low compute power). Note that instead of packing and unpacking data, we could have used the MPI derived data-types, which would have prevented the intermediate copies prior to issuing the MPI communications. This approach is a possible future work.

To sum up, we have shown that the communication-computation overlap had improved the OpenCL performance of the seismic modeling on the GPU cluster and on the APU cluster regardless of the used DPS. Besides, on the GPU cluster the overhead caused by the memory traffic on the PCI Express bus was found to be a limiting factor that prevents the application from scaling ideally, as the number of the compute nodes increases. On the APU cluster, although the overall performance is lower than that

reported by the GPU cluster, the impact of data copies between the `c` memory and the `g` is less significant. Moreover, the overhead was totally suppressed when the zero-copy memory objects were used. Finally, the ratio of the performance numbers of the APU cluster when using the `cggc` DPS, to those when using the `zz` (for both strong and weak scalings) is between 0.5 and 0.6, which corresponds to the ratio of the sustained memory bandwidth of host-visible device memory to that of the integrated GPU global memory.

## 8.3 Seismic migration

In this section, we rely on the seismic modeling workflow and optimizations to put together a large scale implementation of the seismic migration (RTM). We describe the workflow of the application on the CPU cluster and on the HWAs based clusters. We show performance numbers on the different architectures and conduct a comparison study between the CPU cluster, the APU cluster and the GPU cluster. The study includes a comparison based on the execution times (performance), a comparison based on the theoretical maximum power consumption (power efficiency), and a comparison based on the throughput in terms of RTM shots (production efficiency).

### 8.3.1 Deployment on CPU clusters

In this section we evaluate an MPI+Fortran based implementation of the seismic migration on the *PANGEA* cluster. First, we give implementation details about the seismic migration workflow on the CPU cluster. Second, we focus on some performance pitfalls related to the RTM and we present scalability results where we distinguish between strong and weak scaling scenarios, with a special emphasis on the impact of using asynchronous I/O on the application performance.

#### 8.3.1.1 Implementation details

In section 8.2.1.1 we had gone through the description of the large scale workflow of the seismic modeling on the CPU cluster. In this section, we extend this workflow to give a high level picture of the large scale RTM algorithm on the CPU cluster, as well as details about its implementation using MPI+Fortran.

We recall that the RTM algorithm comprises three successive stages (see algorithm 7.2). The forward sweep, or "FWD", whose workflow is equivalent to the seismic modeling. The backward sweep, where the seismic wave is propagated backward in time (this workflow is also similar to the seismic modeling). The third stage is computing the imaging condition in order to produce the final image. In practice the second and third stages are merged into one single step referred to as the backward sweep or "BWD". We describe in algorithm 8.12 the large scale workflow of the seismic migration on the CPU cluster. Given that the communication-computation overlap mechanism had offered a significant performance improvement for the seismic modeling application on the CPU cluster (see section 8.2.1.4), we introduced it to the seismic migration workflow from the very beginning. Therefore, in the algorithm 8.12 we already distinguishes the OpenMP

**Algorithm 8.12** High level description of the multi-node RTM workflow.

```
1: for t ∈ [0..T] do                                   ▷ the forward sweep (or FWD)
2:     if mod(t, 2) == 0 then                          ▷ t is the time-step index
3:         add_seismic_source(u0, t)
4:         !$OMP PARALLEL NUM_THREADS(2)
5:         if tid == 0 then                            ▷ tid is the OpenMP thread index
6:             exchange_halos(u0)                      ▷ described in algorithm 8.5
7:         else
8:             update_interior_wavefield(u1, u0, t)    ▷ see algorithm 8.6
9:         end if
10:        !$OMP END PARALLEL
11:        update_boundary_wavefield(u1, u0, t)        ▷ see algorithm 8.6
12:        save_seismic_snapshot(u1, t)                ▷ this involves I/O operations
13:    else
14:        add_seismic_source(u1, t)
15:        !$OMP PARALLEL NUM_THREADS(2)
16:        if tid == 0 then
17:            exchange_halos(u1)
18:        else
19:            update_interior_wavefield(u0, u1, t)
20:        end if
21:        !$OMP END PARALLEL
22:        update_boundary_wavefield(u0, u1, t)
23:        save_seismic_snapshot(u0, t)
24:    end if
25: end for
26:
27: for t ∈ [T..0] do                                  ▷ the backward sweep (or BWD)
28:    read_seismic_snapshot(tmp, t)                   ▷ tmp is used to read snapshots
29:    if mod(t, 2) == 0 then
30:        add_seismic_receivers(u0, t)
31:        !$OMP PARALLEL NUM_THREADS(2)
32:        if tid == 0 then
33:            exchange_halos(u0)
34:        else
35:            update_interior_wavefield(u1, u0, t)
36:        end if
37:        !$OMP END PARALLEL
38:        update_boundary_wavefield(u1, u0, t)
39:        imaging_condition(u1, tmp, t)               ▷ see algorithm 7.2
40:    else
41:        add_seismic_receivers(u1, t)
42:        !$OMP PARALLEL NUM_THREADS(2)
43:        if tid == 0 then
44:            exchange_halos(u1)
45:        else
46:            update_interior_wavefield(u0, u1, t)
47:        end if
48:        !$OMP END PARALLEL
49:        update_boundary_wavefield(u0, u1, t)
50:        imaging_condition(u0, tmp, t)
51:    end if
52: end for
```

thread dedicated to MPI communications from that in charge of performing the computation.
The forward modeling is roughly similar to the seismic modeling workflow described in algorithm 8.9. However, we recall that we use the selective checkpointing method to rebuild the source wavefield during the backward sweep, which requires data snapshotting during the forward sweep. We recall that the data snapshotting frequency $k$ that we chose is equal to 10. In the algorithm, writing the snapshots to the hard disks (which requires extensive I/O operations) is performed in the subroutine *save_seismic_snapshot*(). During the BWD, the simulation is inverted backward in time. This stage of the algorithm starts with reading the wavefield snapshots (which incurs I/O operations) that were saved during the FWD, in the subroutine *read_seismic_snapshot*(). The workflow related to the MPI communications and to exchanging halos between the neighbors is the same as that in the seismic modeling and was thoroughly described in section 8.2.1.2. The workflow related to the imaging condition computation is described in algorithm 7.2. It is to be noted, that when reading or writing the wavefield snapshots we compared using the synchronous I/O operations and the asynchronous I/O operations. Asynchronous I/O is a form of input/output processing that delegates the I/O requests to an internal pool of threads, and returns to the application immediately. It also requires a synchronization mechanism in order to ensure data consistency. In Fortran, asynchronous I/O was introduced in the 2003 standard and is implemented, at the time of writing, only in the Intel compiler. Since we perform a wavefield snapshot every 10 time-steps after each of which we overwrite the wavefield array being written to storage, we inserted waiting routines before the subroutines *update_interior_wavefield*() in order to make sure that the data is consistent before overwriting it.

#### 8.3.1.2 Performance results

In this section we present the performance results of the large scale MPI+Fortran implementation of the seismic migration on the *PANGEA* cluster (see table 5.1). We consider using the asynchronous I/O routines, provided by the Fortran 2003 standard (implemented in the Intel Fortran compiler), and emphasize its impact on the application performance. We distinguish the strong scaling scenario from the weak scaling scenario.

**8.3.1.2.1 Strong scaling tests** We ran the seismic migration implementation on the compute grid $\Omega_{9\times9\times9}$ while using the *3D SEG/EAGE salt model* ($\mathcal{V}$) as an input data. We consider using one MPI process per compute node in order to predict the behavior of the seismic migration implementation on the HWAs based clusters. Besides, we consider another scenario where we place sixteen processes per node since it was shown the most efficient configuration on the CPU cluster (see section 8.2.1.4). Each test was ran over 1000 iterations and we show the performance numbers of the slowest MPI process in each configuration. We follow the same process and thread pinning recommendations as those used for the seismic modeling, i.e. the communication thread and the computation thread are always running on the same socket, and we avoid using the core number 0. The table 8.16, in addition to the table 8.12, helps understand the signification of the symbols present in the following performance results figures.

The figure 8.23 represents the performance results of the seismic migration on the CPU cluster with respect to the strong scaling scenario, and while placing one MPI process per compute node. The figure emphasizes the impact of using asynchronous I/O on

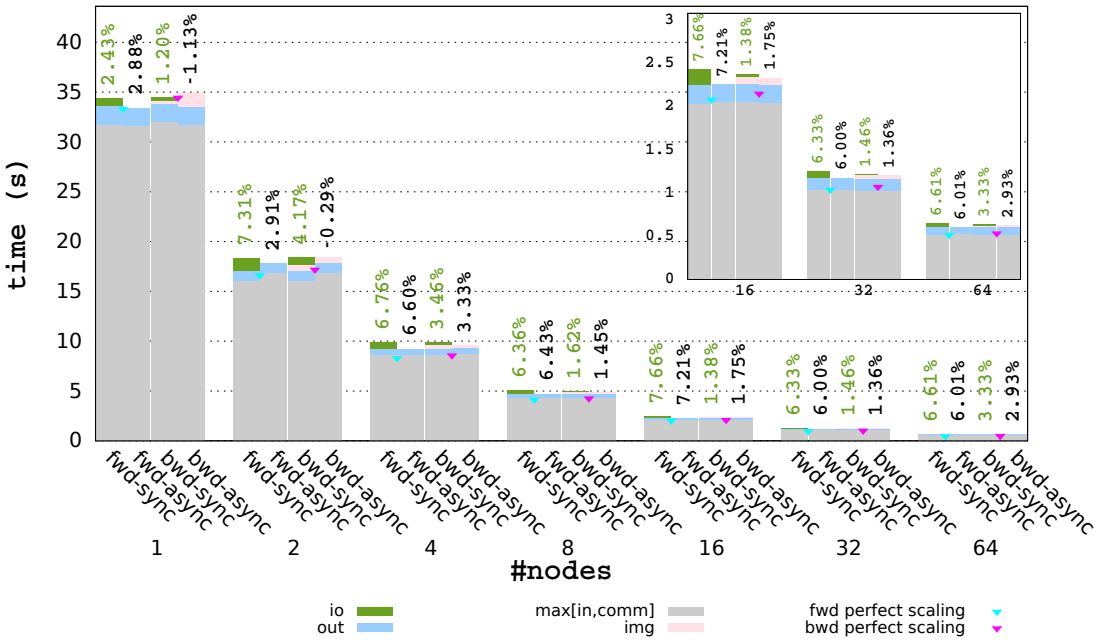



FIGURE 8.23: Performance impact of the asynchronous I/O on the **seismic migration** application on the **CPU cluster**. The **strong** scaling is considered and only **1 MPI** process is used on each compute node. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

the application performance. The dark green percentages on top of the histogram bars represent the ratio of the time spent issuing synchronous I/O operations with respect to the average iteration time. The black percentages represent the performance gain (or loss) offered when using asynchronous I/O operations. We recall that the MPI communications are overlapped with computations on the seismic migration implementation, thus we do not illustrate the communications time in the figures (`comm`) but we rather show the `max[in,comm]` time. We notice that the asynchronous I/O have improved the performance in almost all the test cases except for the BWD sweep in the cases "1 node" and "2 nodes". Besides, on can notice that the I/O time in the FWD is usually higher than that in the BWD. This difference can be explained by the fact that the "write" I/O operations (in the subroutine *save_seismic_snapshot*()) takes a longer time than that of the "read" I/O operations (in the subroutine *read_seismic_snapshot*()). In addition, the figure reports a close to perfect strong scaling as the number of nodes increases.

| | |
|---|---|
| `io` | the time spent in I/O for data snapshotting |
| `img` | the time spent in computing the imaging condition |
| `fwd perfect scaling` | an artificial reference that mimics a perfect scaling for FWD |
| `bwd perfect scaling` | an artificial reference that mimics a perfect scaling for BWD |
| `fwd-sync` | results of FWD using the synchronous I/O operations |
| `fwd-async` | results of FWD using the asynchronous I/O operations |
| `bwd-sync` | results of BWD using the synchronous I/O operations |
| `bwd-async` | results of BWD using the asynchronous I/O operations |

TABLE 8.16: Description of the notations used in the figures illustrating the performance results of the seismic migration.

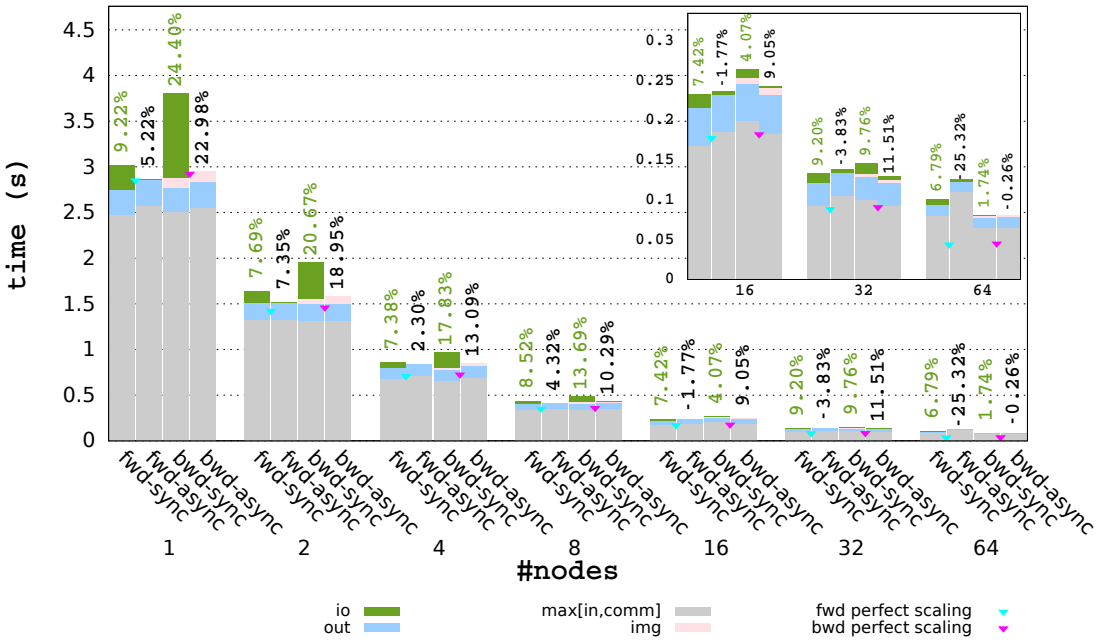



FIGURE 8.24: Performance impact of the asynchronous I/O on the **seismic migration** application on the **CPU cluster**. The **strong** scaling is considered and **16 MPI** processes are used on each compute node. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

The figure 8.24 represents the performance results of the seismic migration on the CPU cluster with respect to the strong scaling scenario, and while placing sixteen MPI processes per compute node. It also shows the impact of using asynchronous I/O on the application performance in this case. We point out that the I/O percentages, with respect to the overall execution times, are decreasing as the nodes count increases, because in each test case with a high number of nodes (16, 32 and 64) the communications time surpasses the computations time (see figure 8.14) which reduces the fraction of the overall time that the I/O operations represent. Besides, we notice that the asynchronous I/O is beneficial only for a small number of nodes (up to 8), and has a sporadic effect on the performance if a higher number of nodes is used. On the one hand, SMT is exploited to overlap the MPI communications (whose time increases as the number of nodes increases). On the other hand, the asynchronous I/O engine has its own thread pool that shares resources with the communication threads and with the computations threads. This results in an overuse of each compute core which leads to a lower strong scaling than with one MPI process per node (see `fwd perfect scaling` and `bwd perfect scaling`).

**8.3.1.2.2 Weak scaling tests** Following, we explore the impact of the asynchronous I/O on the performance of the seismic migration, while taking into consideration the weak scaling scenario instead. We recall that the table 5.7 summarizes the numerical configuration of the compute grids of this test scenario. Similarly to the strong scaling scenario, we start our testings with the one MPI process per compute node configuration. We run the tests on 1 to 64 compute nodes of the *PANGEA* CPU cluster. For this test configuration we use the *Small 3D SEG/EAGE salt model* as initial

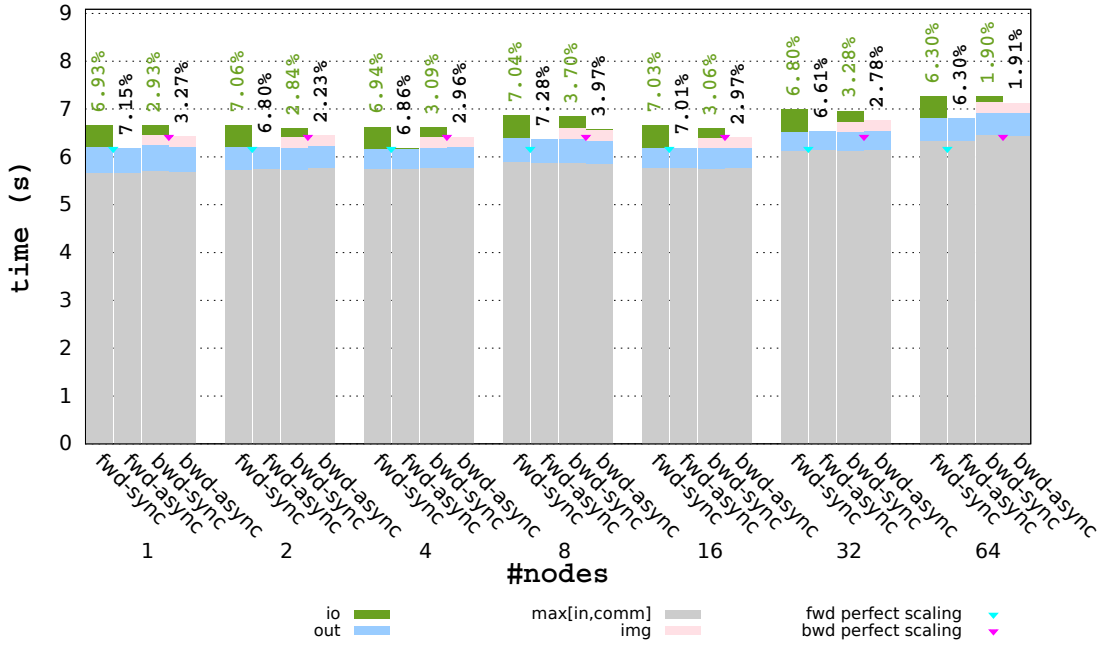



FIGURE 8.25: Performance impact of the asynchronous I/O on the **seismic migration** application on the **CPU cluster**. The **weak** scaling is considered and only **1 MPI** process is used on each compute node. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

dataset. Then we consider the weak scaling test with sixteen MPI processes on each compute node.

The figure 8.25 summarizes the execution times of the weak scaling test from 1 to 64 nodes using only one MPI process per compute node. First, one can see that the asynchronous I/O operations have perfectly helped reducing the overhead related to I/O in the seismic migration application. Second, we observe a linear scaling for a number of nodes ranging from 1 to 4. After that, the more nodes we use the lower becomes the scaling. This is caused by the raise of the neighbors count per subdomains which increases the communication occurrences as the nodes count increases.

Finally, we show the execution times of the weak scaling test, while considering the sixteen MPI processes per compute node case, in figure 8.26. Surprisingly, the figure reports that the I/O overhead during the BWD sweep is higher than the overhead during the FWD (this was also noted in the strong scaling test with sixteen MPI processes per compute node). This is a strange problem that would require further investigations. However, we can see that relying on the asynchronous I/O operations had helped to partially hide this overhead, especially in the BWD sweep. Besides, the figure shows that the scaling is generally lower than the one with one MPI process per node due to the overuse (by multiple threads) of the compute nodes.

### 8.3.2 Deployment on hardware accelerators

In this section we propose an MPI+OpenCL large scale implementation of the RTM on *DIP* and *RDHPC* clusters. First, we give an overview about the changes we had to introduce to the application workflow in order to adapt it to HWAs. Second, we

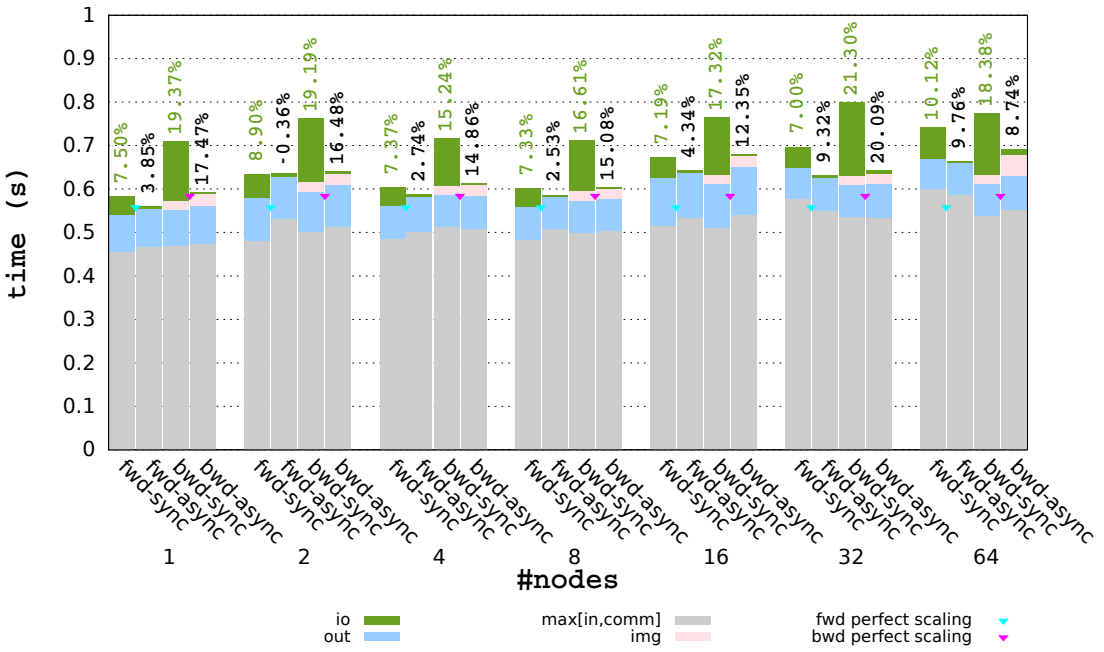



FIGURE 8.26: Performance impact of the asynchronous I/O on the **seismic migration** application on the **CPU cluster**. The **weak** scaling is considered and **16 MPI** processes are used on each compute node. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

present an OpenCL performance evaluation, based on strong and weak scaling tests, on the GPU cluster and on the APU cluster while considering the two DPSs: `cggc` and `zz`.

#### 8.3.2.1 Implementation details

In this section we describe the large scale workflow of the seismic migration on the GPU cluster, and on the APU cluster. The workflow is relying on the algorithm 8.11 which characterizes the seismic modeling on HWAs based clusters, and in which the communications are overlapped with computations. Similarly to the implementation of the seismic migration on CPU clusters, we particularly focus on studying the impact of the asynchronous I/O operations on the application performance.

The algorithm 8.13 shows a high level description of the large scale seismic migration workflow on the HWAs based clusters. The FWD step of the application is equivalent to the workflow of the seismic modeling application described in section 8.2.2, except that in the case of the seismic migration we consider the data snapshotting every 10 time-steps of computation. The data snapshotting is held in the subroutine *save_seismic_snapshot*() and preceded by a data transfer between the GPU memory and the main memory, in the case of the GPU cluster or the APU cluster with the `cggc` DPS, or by a memory mapping operation in the case of the APU cluster with the `zz` DPS (see the subroutine *dtoh_wavefield*()). In the BWD, every 10 time-steps (the same frequency as the data snapshotting frequency) of the algorithm, the corresponding source wavefield snapshot is read from the local storage of each compute node (the subroutine *read_seismic_snapshot*()), in order to be correlated with the updated receiver wavefield, and to progressively build the final image in the subroutine *imaging_condition*(). The

**Algorithm 8.13** Description of the seismic migration algorithm on the GPU and APU clusters, with overlapping communications with computations and considering the asynchronous I/O.

```
1:  for t ∈ [0..T] do                                          ▷ the FWD stage
2:      if mod(t, 2) == 0 then                                 ▷ t is the time-step index
3:          add_seismic_source(u0, t)
4:          !$OMP PARALLEL NUM_THREADS(2)
5:          if tid == 0 then                                   ▷ tid is the OpenMP thread index
6:              exchange_halos(u0)                             ▷ described in algorithm 8.10
7:          else
8:              update_interior_wavefield(u1, u0, t)           ▷ see algorithm 8.11
9:          end if
10:         !$OMP END PARALLEL
11:         update_boundary_wavefield(u1, u0, t)               ▷ see algorithm 8.11
12:         dtoh_wavefield(u1)
13:         save_seismic_snapshot(u1, t)                       ▷ may use asynchronous I/O
14:     else
15:         add_seismic_source(u1, t)
16:         !$OMP PARALLEL NUM_THREADS(2)
17:         if tid == 0 then
18:             exchange_halos(u1)
19:         else
20:             update_interior_wavefield(u0, u1, t)
21:         end if
22:         !$OMP END PARALLEL
23:         update_boundary_wavefield(u0, u1, t)
24:         dtoh_wavefield(u0)
25:         save_seismic_snapshot(u0, t)
26:     end if
27: end for
28:
29: for t ∈ [T..0] do                                          ▷ the BWD stage
30:     read_seismic_snapshot(tmp, t)                          ▷ may use asynchronous I/O
31:     if mod(t, 2) == 0 then
32:         add_seismic_receivers(u0, t)
33:         !$OMP PARALLEL NUM_THREADS(2)
34:         if tid == 0 then
35:             exchange_halos(u0)                             ▷ see algorithm 8.10
36:         else
37:             update_interior_wavefield(u1, u0, t)
38:         end if
39:         !$OMP END PARALLEL
40:         update_boundary_wavefield(u1, u0, t)
41:         dtoh_wavefield(u1)          ▷ retrieve the wavefield array from the GPU memory
42:         imaging_condition(u1, tmp, t)                      ▷ see algorithm 7.2
43:     else
44:         add_seismic_receivers(u1, t)
45:         !$OMP PARALLEL NUM_THREADS(2)
46:         if tid == 0 then
47:             exchange_halos(u1)
48:         else
49:             update_interior_wavefield(u0, u1, t)
50:         end if
51:         !$OMP END PARALLEL
52:         update_boundary_wavefield(u0, u1, t)
53:         dtoh_wavefield(u0)
54:         imaging_condition(u0, tmp, t)
55:     end if
56: end for
```

| | |
|---|---|
| `dtoh-io` | the time spent in the subroutine $dtoh\_wavefield()$ |

TABLE 8.17: Description of the additional notations used in the figures illustrating the performance results of the seismic migration on HWAs based clusters.

computations subroutines, used to update the receiver wavefield values, as well as the communications subroutine are the same as those used in the seismic modeling workflow and were already covered in the algorithms 8.10 and 8.11. We point out that the subroutine $imaging\_condition()$ is performed by the CPU (with the help of OpenMP) in our implementation of the seismic migration for both GPU and APU clusters, in order to save space on the HWAs memories and to be able to conduct the same scaling tests, namely strong scaling tests, as in the seismic modeling. Therefore, prior performing the imaging condition for the RTM, the most up-to-date wavefield array is retrieved (every 10 time-steps in our case) from the GPU memory (see the subroutine $dtoh\_wavefield()$). Note that writing the snapshots to the hard disks also requires memory transfers through the PCI Express bus in the case of the discrete GPU cluster or a copy from the GPU memory to CPU memory in the case of the APU cluster. Those data transfers are copies are not overlapped with the computations. Besides, as it had been shown in section 8.2.2, the MPI communication-computation overlap had allowed a performance enhancement of the seismic modeling application on the HWAs based clusters. Therefore we only consider the case where we overlap MPI communications with computations in our seismic migration large scale implementations on the GPU and APU clusters.

#### 8.3.2.2 Performance results

We present in this section the performance numbers of the seismic migration on the surveyed HWAs based clusters. We consider both the strong and weak scaling scenarios. For the APU cluster, we consider using two data placement strategies: *cggc* where the wavefield arrays are duplicated in the GPU memory and *zz* where the arrays are zero-copy memory objects.

For each test we present the performance of the FWD stage and the performance of the BWD stage separately. We recall that we use the selective checkpointing method to rebuild the source wavefield during the BWD. The data snapshotting frequency $K$ is equal to 10. In order to help read the performance results depicted in the following figures, we summarize in table 8.17 the signification of the figure legends that are specific to the seismic migration implementation on the HWAs based clusters. We recall that the rest of the legends were described in tables 8.12 and 8.15.

**8.3.2.2.1 Strong scaling tests** We ran the seismic migration implementation on the compute grid $\Omega_{9\times9\times9}$ using the *3D SEG/EAGE salt model* ($\mathcal{V}$) as an input data, over 1000 time-steps. We distinguish the case where we use the Fortran 2003 asynchronous I/O (`async`) from the case where we use the regular synchronous I/O (`sync`). We recall that we show the execution times of the slowest MPI process in each configuration. We also recall that, given the limited space of each discrete GPU off-chip memory, and of the `g` memory of each APU, the strong scaling scenario comprises two test cases: "8 nodes" and "16 nodes".

In figure 8.27, we show the performance numbers of the seismic migration on the discrete GPU cluster with respect to the strong scaling scenario. First, we noticed that the I/O percentages are more important than those reported in the case of the seismic migration on the CPU cluster. As a matter of fact, the computation times are accelerated by the GPU, as compared to the computations times of the CPU (in the case of the implementation on the CPU cluster), which results in the time dedicated to I/O representing a higher fraction of the overall time. It is thus mandatory to rely on asynchronous I/O operations in the implementations of the seismic migration on the HWAs based clusters in order to obtain a good scaling. According to the performance results, overlapping the I/O operations with the computations is more beneficial to the HWAs based clusters than to the CPU cluster (with placing 16 MPI processes per compute node). This is because the CPUs in the HWAs based clusters are often idle and are exploited as resources for asynchronous I/O. Second, we note that the use of the asynchronous I/O showed a performance enhancement of up to 40%, which improved the scaling efficiency of the seismic migration on the GPU cluster. Finally, the overhead due to memory transfers over the PCI Express bus (`d-h-comm+dtoh-io`) represents about 16% of the overall time of the `async` case, which may harm the scaling if the node counts increases (this was confirmed by conducting additional strong scaling tests on 32 and 64 GPU based nodes, which reported that the PCI overhead had reached 24% of the overall time).

The figure 8.28 shows the performance numbers of the seismic migration on the APU cluster with respect to the strong scaling scenario and with the `cggc` DPS. We note that the I/O times are more important in the forward sweep than in the backward sweep. This informs that the write operations are more expensive than the read operations. Besides, the asynchronous I/O is also beneficial to the APU (especially for the test case 16 nodes), as it offers a performance gain up to 19%. Finally, we notice that the overhead

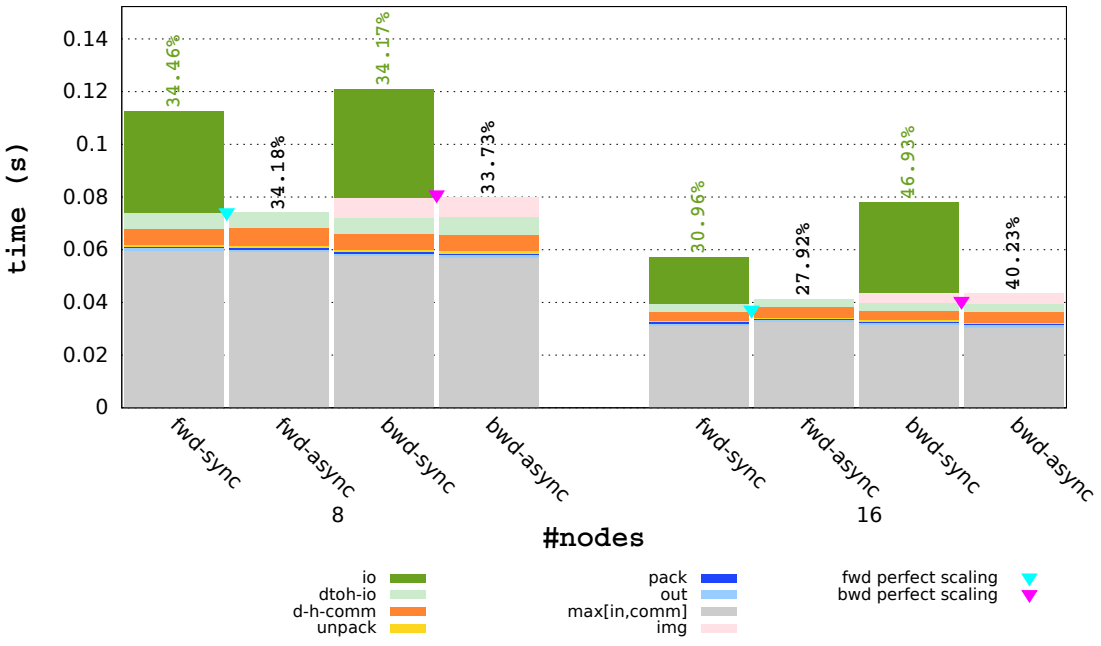



FIGURE 8.27: Performance impact of the asynchronous I/O on the **seismic migration** application on the **GPU cluster**. The **strong** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

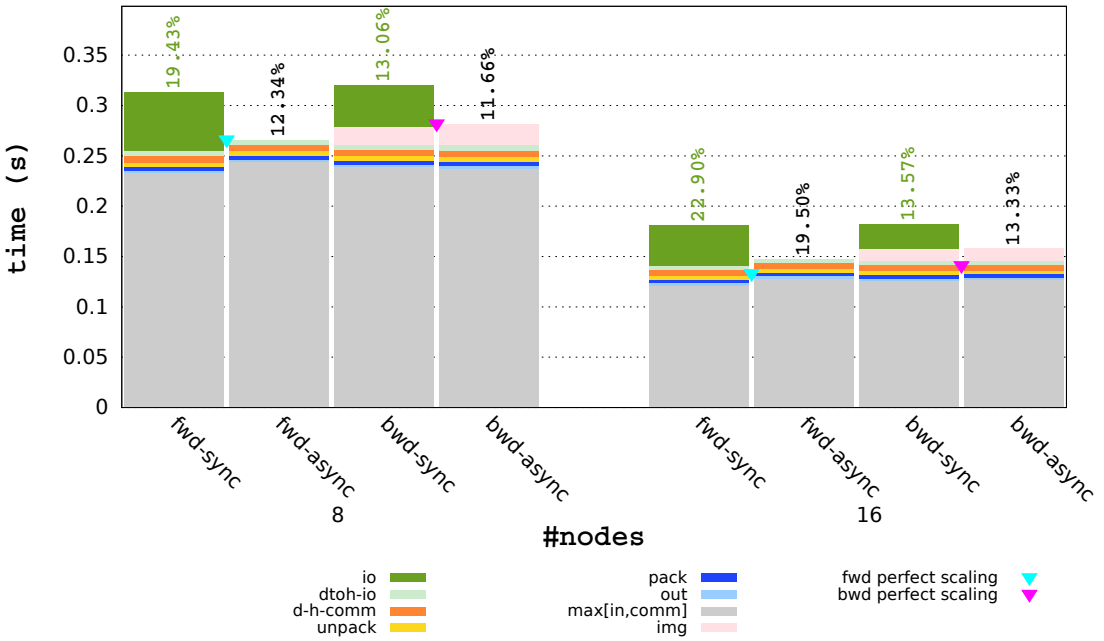



FIGURE 8.28: Performance impact of the asynchronous I/O on the **seismic migration** application on the **APU cluster** with the **cggc** DPS. The **strong** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

due to memory traffic between the `c` memory and the `g` memory is reduced as compared to that observed in the case of discrete GPUs, thanks to the Garlic bus. However, it is still a limiting factor to achieving a linear scaling.

Similarly to the seismic modeling, we consider using the zero-copy memory objects in the implementation of the seismic migration on the APU cluster. In the figure 8.29 we present the performance of the strong scaling scenario. One can notice that the asynchronous I/O had offered a performance gain in this case as well. The figure reports up to 11% of performance enhancement which helped achieving a very good scaling (except for the test case "2 nodes"). As a matter of fact, the overhead due to memory traffic between the CPU and GPU is completely suppressed. It is to be noted that the I/O time fractions are smaller in this case, as compared to the results with the `cggc` DPS, since the computations times with the `zz` DPS are almost twice as high as with the `cggc` DPS.

**8.3.2.2.2 Weak scaling tests** In this paragraph, we consider running the seismic migration implementation on the compute grids detailed in table 5.7, with respect to the weak scaling scenario. Given that the smallest compute grid in this test bed, i.e. $\Omega_{16\times16\times16}$, fits in one GPU based compute node as well as in one APU based compute node, the weak scaling scenario involves a number of compute nodes that ranges from 1 to 16.

The figure 8.30 represents the performance numbers of the weak scaling test of the seismic migration on the discrete GPU cluster. Given that the size of the compute grids is doubling as the nodes count increases (the biggest compute grid is $\Omega_{8\times8\times4}$), the percentage of the I/O time per subdomain is almost constant for all the test cases. Thanks

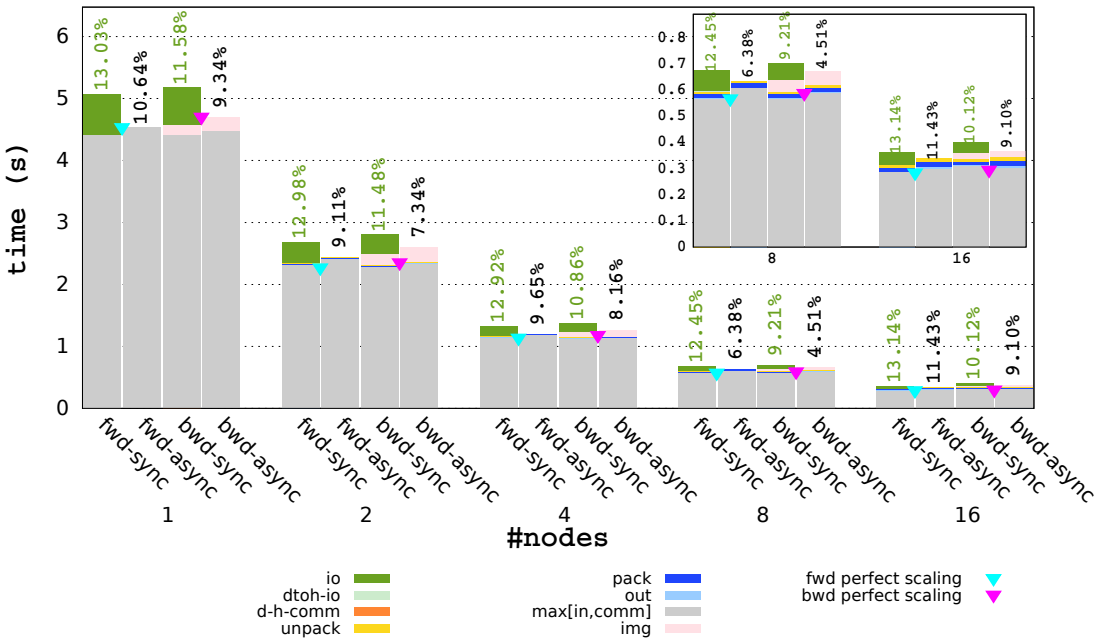



FIGURE 8.29: Performance impact of the asynchronous I/O on the **seismic migration** application on the **APU cluster** using the **zero-copy** memory objects. The **strong** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

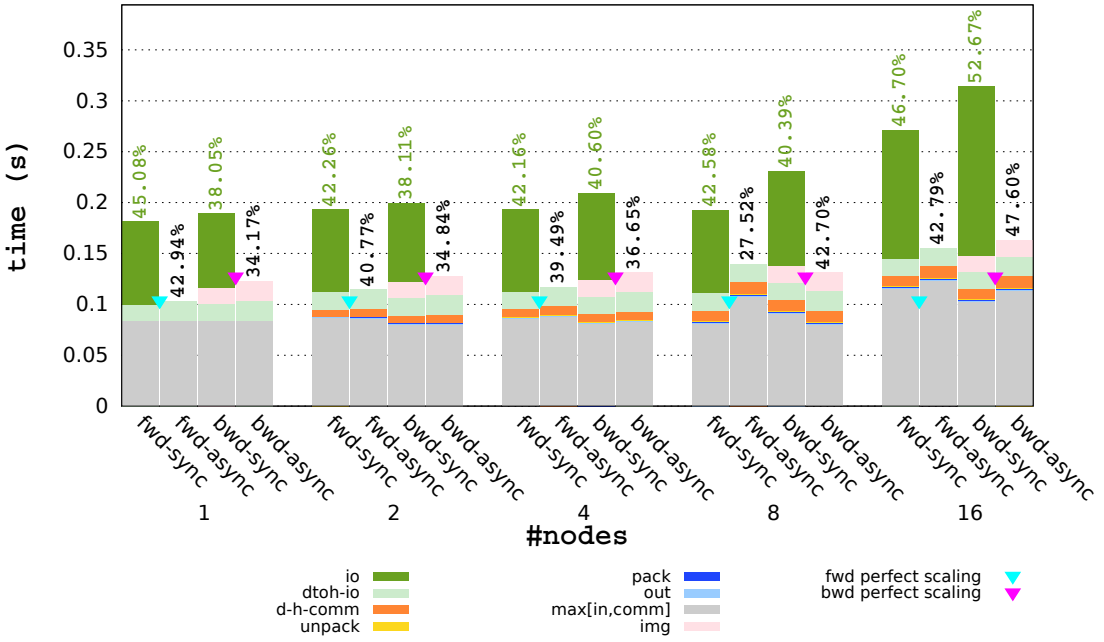



FIGURE 8.30: Performance impact of the asynchronous I/O on the **seismic migration** application on the **GPU cluster**. The **weak** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

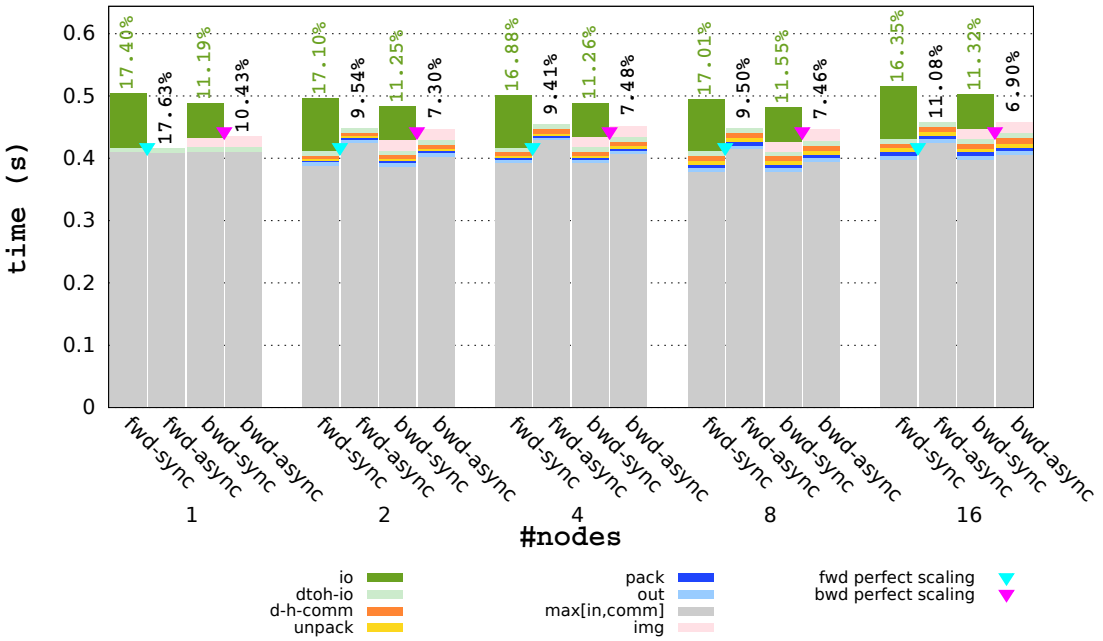



FIGURE 8.31: Performance impact of the asynchronous I/O on the **seismic migration** application on the **APU cluster** with the **cggc** DPS. The **weak** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

to the asynchronous I/O operations, the I/O overhead is overlapped since the figure reports that the `async` version offers a performance gain up to 47%. However, we notice that the scaling is not optimal especially for the test cases "8 nodes" and "16 nodes", where the times associated to communications and MPI synchronization (reported by the communication thread) were dominating the computation times (especially for the FWD stage). In other words, due to the excessive MPI communications overhead as the number of neighbors increases with respect to the nodes count (which implies more MPI communications), the weak scaling is less efficient as the node counts increases. Besides, the overhead associated to the PCI Express memory traffic represent a relatively high fraction (we recall that the discrete GPU based compute nodes are featured with a PCI Express bus gen 3) of the overall time (up to 18%), as the `d-h-comm` is proportional to the communications time. This overhead did not help to achieve an ideal scaling even in the test cases where the communication times do not dominate the computation times, i.e. "2 nodes" and "4 nodes", but thanks to the asynchronous I/O the achieved scaling is still good. As a perspective, we aim in the future to overlap the memory transfer between CPU and GPU, with the computations on the GPU, in order to reach a near to linear scaling.

The figure 8.31 shows the performance of the weak scaling test of the seismic migration on the APU cluster. We rely in this benchmark on the `cggc` DPS. Similarly to the strong scaling scenario, the I/O overhead in the FWD is more important than that in the BWD (write operations are more expensive than read operations on the APU cluster). However, the asynchronous I/O helped to completely suppress the overhead in the test case "1 node", and to reduce it in the other test cases. This offered a performance enhancement of up to 17.6%. Moreover, we point out that the overhead associated to the memory traffic between the CPU and the GPU is alleviated compared to the weak

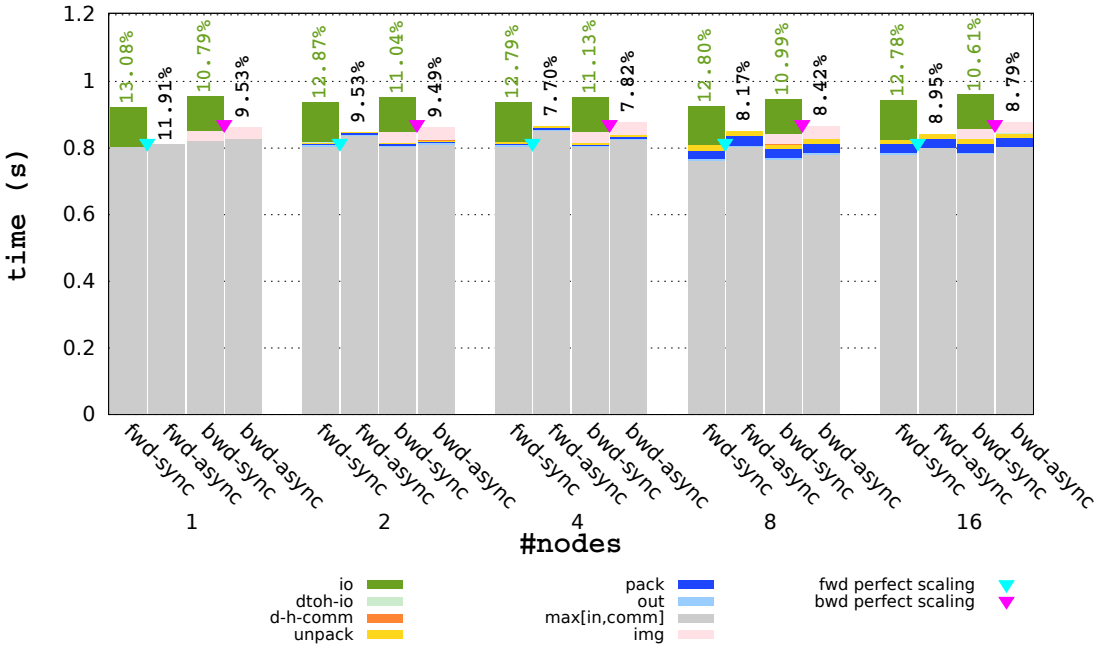



FIGURE 8.32: Performance impact of the asynchronous I/O on the **seismic migration** application on the **APU cluster** using the **zero-copy** memory objects. The **weak** scaling is considered. The $\mathcal{V}$ **dataset** is used as input data. Each execution time is the average of 1000 of simulation iterations.

scaling results on the GPU cluster. But, the scaling is not quite optimal (even though it is more efficient, in the case of high nodes count, as compared to the scaling of the discrete GPUs) because of this extra overhead.

In the figure 8.32, we illustrate the performance results of the weak scaling test on the APU cluster with the `zz` DPS. First, the asynchronous I/O is beneficial to the performance of the seismic migration on the APU cluster when using the zero-copy memory objects. Indeed, the figure reports that it allows a performance gain up to 12%. Second, the overhead related to the memory traffic between the CPU and GPU is removed, thanks to the zero-copy memory objects. Third, the times associated to data packing and data unpacking are relatively higher since those operations are performed by the CPU. One can see that the `pack` and `unpack` times are increasing as the nodes count increases. As a matter of fact, this overhead starts limiting the scaling efficiency (see the test cases "8 nodes and "16 nodes"), and unfortunately hides the enhancement offered by using zero-copy memory objects. We recall that, in the one hand the OpenCL standard did not allow us to overlap this overhead with computations, and in the other hand, we did not invest enough efforts in optimizing the subroutines $pack\_halos\_in\_linear\_buffers()$ and $unpack\_buffers\_into\_halo\_regions()$ on the CPU which is planned in our future work (i.e. efficiently parallelize the subroutines or use the MPI derived data-types). Finally, we recall that here again using the zero-copy memory objects cuts the performance to half as compared to using the `cggc` DPS.

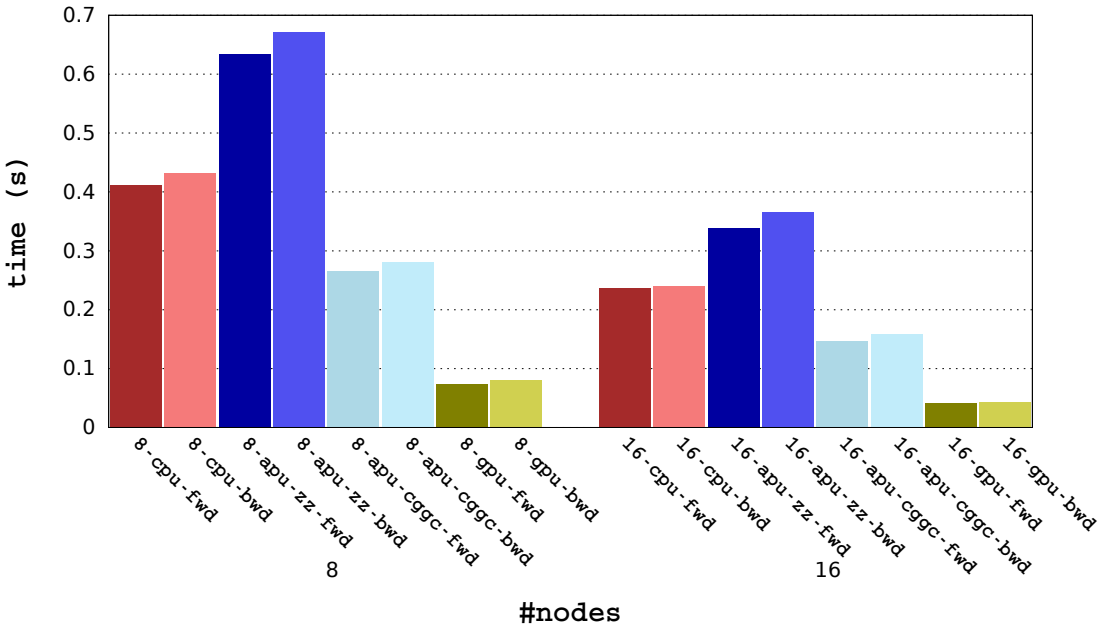



FIGURE 8.33: An **execution time** based comparison of the performance of the large scale **seismic migration** implementation on the CPU cluster, on the GPU cluster and on the APU cluster (with the `cggc` and `zz` DPSs). The comparison involves the tests run on the compute grid $\Omega_{9\times9\times9}$.

### 8.3.3 Performance comparison

After having evaluated the large scale implementations of seismic applications on the different hardware, we try in this section to present a comparison study between the CPU cluster, the APU cluster and the GPU cluster. The comparison is based on the performance results of the RTM obtained on the three clusters, with respect to the strong scaling and weak scaling scenarios, and detailed in the previous sections. To begin with, the study is based on straightforward comparisons based on the execution times for a fixed number of compute nodes. Then, we rely on a performance projection in order to compare the throughput in terms of RTM shots of the three architectures, and to compare the performance results while fixing the power consumption envelope (based on Watts of TDP for the CPU cluster, and on the maximum power consumption for the HWAs based clusters).

#### 8.3.3.1 Comparison based on measured results

In the figure 8.33 we group the performance results of the large scale implementation of the RTM application on the CPU cluster, on the GPU cluster and on the APU cluster. The results correspond to the performance numbers obtained when running the seismic migration on the compute grid $\Omega_{9\times9\times9}$ (see table 5.5). For the APU cluster we included the results obtained with the `cggc` DPS, as well as those with the `zz` DPS. Given that the strong scaling test on the HWAs based clusters was conducted only on eight and sixteen nodes, we restrict our comparison to those two test cases. The text under each histogram bar, in the figure, refers to the FWD or the BWD stage of the RTM algorithm, on a fixed number of nodes (8 or 16) of a given architecture (CPU,

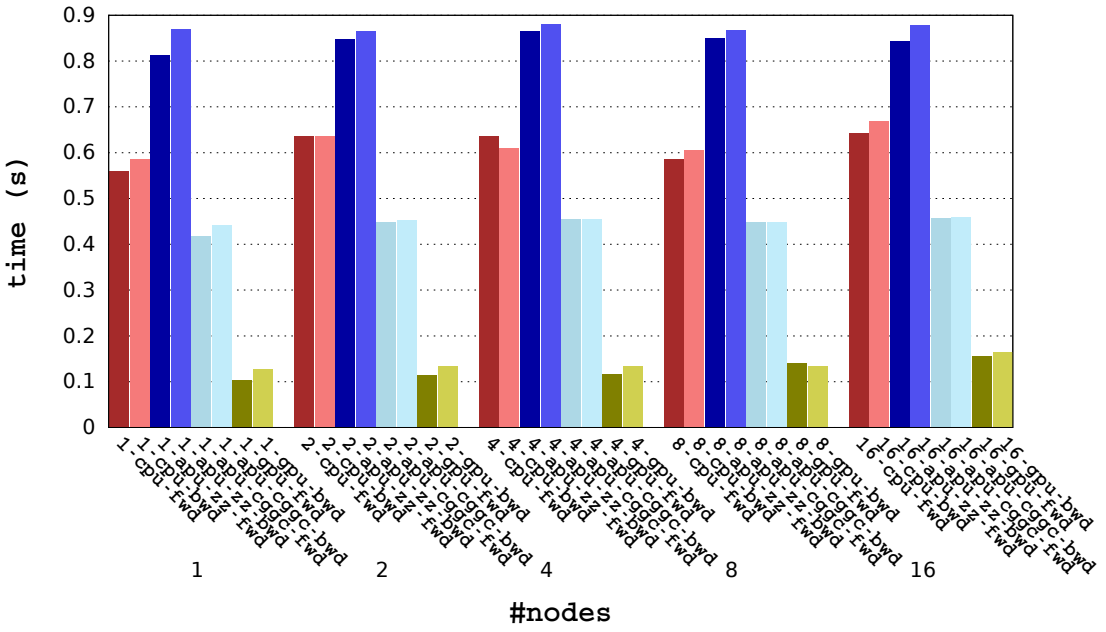



FIGURE 8.34: An **execution time** based comparison of the performance of the large scale **seismic migration** implementation on the CPU cluster, on the GPU cluster and on the APU cluster (with the `cggc` and `zz` DPSs). The comparison involves the tests run on the compute grids ranging from $\Omega_{16\times16\times16}$ to $\Omega_{8\times8\times4}$ (see table 5.7).

GPU or APU), with mentioning which DPS was applied (for the APU cluster only). For example the histogram bar `8-apu-cggc-bwd` indicates the overall execution time of the RTM backward sweep on 8 nodes of the APU cluster, while using the `cggc` DPS. We recall that the strong scaling test was performed on the compute grid $\Omega_{9\times9\times9}$, with using the $\mathcal{V}$ dataset as input data, and with a data snapshotting frequency equal to 10. The numbers reported in the figure (lower is better) indicate that the APU cluster, with the `cggc` DPS, outperforms the CPU cluster by a factor of 1.6×. We recall that each compute node on the CPU cluster comprises two sockets, which means for a socket to socket comparison the speedup ratio becomes 3.2×. Besides, we note that when using the `zz` DPS the application takes more than twice the time as when using the `cggc` DPS on the APU cluster (more precisely 2.3×). We believe that this is mainly due to the two fold difference between the sustained bandwidth of the GPU memory and that of the device-visible host memory, which would impact the memory bound applications which is the case of the RTM. Moreover, it comes with no surprise that the GPU cluster outperforms the APU cluster (with the `cggc` DPS) by a factor of 3.5×, and almost by a factor of 8.3× when the zero-copy memory objects are used. We recall that theoretically the discrete GPUs are an order of magnitude more compute powerful, and have 10× higher memory bandwidth than APUs.

We aggregate the best performance results, on each architecture, when running the seismic migration on the compute grids $\Omega_{16\times16\times16}$, $\Omega_{16\times16\times8}$, $\Omega_{16\times8\times8}$, $\Omega_{8\times8\times8}$ and $\Omega_{8\times8\times4}$ (see table 5.7) in figure 8.34. The reader can notice that the figure reports roughly the same information as the previous comparison illustrated in figure 8.33, and that for all the test cases from "1 node" to "16 nodes". However, in the case of the GPU cluster, given that the execution time slightly increases with respect to the nodes count, the GPU to APU ratio decreases and reaches 3× in the test case "16 nodes". But, this

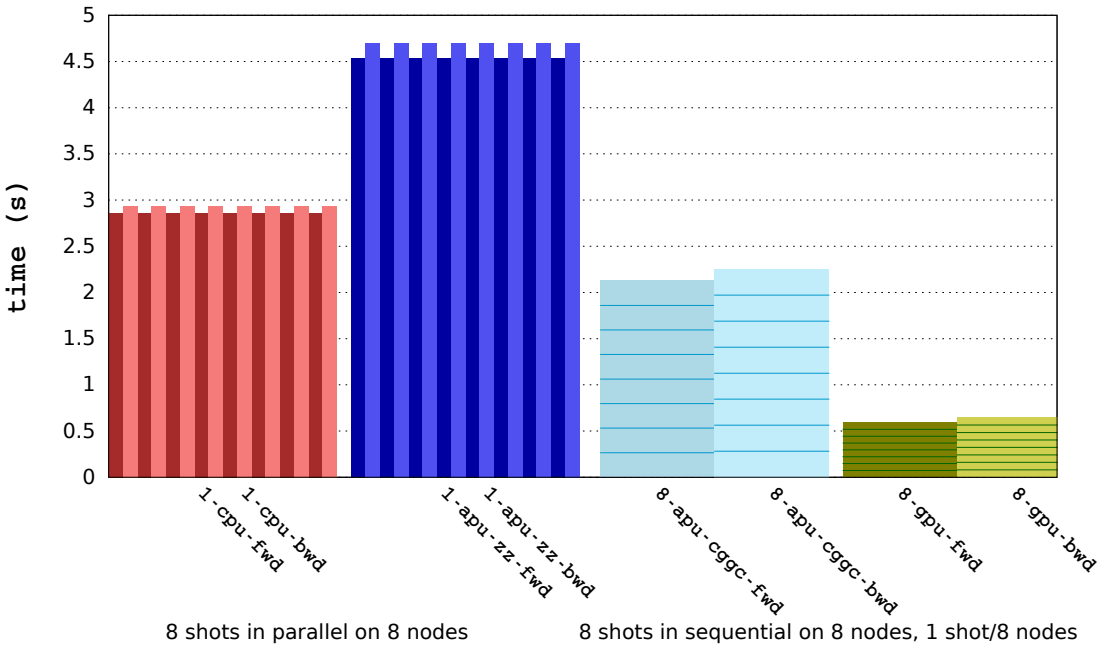



FIGURE 8.35: **Comparison** of the execution times of the large scale **seismic migration** implementation, and the **minimum required hardware configuration** to process **8 shots**, on the CPU cluster, on the GPU cluster and on the APU cluster (with the `cggc` and `zz` DPSs).

does not influence the comparison and the global findings we pointed out in the previous comparison. In addition, for the following comparisons, we will need to use a fixed size problem ($\Omega_{9\times9\times9}$) on a varying number of nodes. Therefore, for the rest of the chapter it is more relevant to rely on the comparison depicted in figure 8.33.

#### 8.3.3.2 Comparison based on performance projection

We recall that the seismic migration is subject, in a higher level, to a natural parallelism which consists of processing multiple shot experiments at a time. Based on the memory requirements of the application, we found out that the minimum configuration in the GPU cluster, in order to process one shot, is 8 compute nodes. It is the same configuration required for the APU cluster with the `cggc` DPS. However, on the CPU cluster or the APU cluster with the `zz` DPS, processing one shot requires only one compute node. In the latter case, this means that 8 shots can be processed in parallel on 8 nodes by simply placing one shot per compute node would require, in the first case, 8 sequential passes on 8 nodes of the GPU cluster or on 8 nodes on the APU cluster (with `cggc`). The figure 8.35 assimilates these particular configurations, and evaluates what we called the "production throughput" of each architecture. This production throughput is estimated as follows.

- The execution time on the GPU cluster represents 8× (we illustrate this in the figure by the green horizontal lines) the execution time of the test case "8 nodes" in the strong scaling scenario (see figure 8.27).

- The execution time of the APU cluster (with the `cggc` DPS) is the execution time obtained in the test case "8 nodes" of the strong scaling scenario (see figure 8.28) multiplied by 8 (we illustrate this in the figure by the blue horizontal lines).
- The execution time of the CPU cluster (resp. the APU cluster with the `zz` DPS) corresponds to the test case "1 node" in the figure 8.24 (resp. in the figure 8.29), but depicted eight times in the figure to illustrate that 8 nodes are running in parallel.

Therefore, this comparison has the advantage of taking into consideration the loss of parallel efficiency as the nodes count increases, especially in the case of the GPU cluster. To begin with, the figure shows that the ratio of the execution time of the APU cluster (with the `zz` DPS), to the execution time on the GPU cluster is slightly lower (7.6× compared to 8.3× in the previous configuration), thanks to the fact that when using the zero-copy memory object one shot fits on one APU node removing the need of performing any MPI communication. This could be even more significant if we had to use a higher number of GPU based compute nodes (for example with GPUs with lower memory capacities we would have needed more GPU nodes, or with 64 GB APU or CPU nodes we could have processed larger seismic shots (requiring more GPU nodes)), since the scaling efficiency on the GPU cluster tends to decrease as the nodes count increases. Furthermore, the ratio of the execution time of the APU cluster (with the `zz` DPS), to the execution time on the APU cluster (with the `cggc` DPS) is also reduced (2× compared to 2.3× in the previous configuration) because to the same reason. Finally, the figure reports roughly the same performance ratio between the performance of the GPU cluster and the APU cluster (with the `cggc` DPS) as in the figure 8.33, thanks to the relatively good scaling on the GPU cluster and on the APU cluster with the `cggc` DPS.

We have mentioned in section 3.3, that the APU is a low power chip and that it draws at most 95 W (maximum power). We also mentioned that a system with a high-end discrete GPU and a high-end CPU features, approximately, a power envelope of 400 W (maximum power). Besides, each CPU socket on a compute node of the CPU cluster consumes 150 W TDP.
We would like to take into consideration those details, related to the power consumption, in our comparative study. However, unlike the single node study in chapters 6 and 7, we do not have the appropriate meters that are used to measure the real power consumption of racks and clusters. Therefore, we rely on the theoretical maximum power consumption of each hardware configuration, in order to estimate the power efficiency of the seismic migration on the CPU and HWAs based clusters.
To this purpose, we consider gathering the execution times of each cluster on a fixed power envelope that corresponds to the estimated maximum power of the 16-nodes APU cluster, i.e. approximately 1600 W. This would correspond to roughly four nodes on the GPU cluster, and roughly to eight compute nodes on the CPU cluster (we underestimate the maximum power of a socket to 100 W). Given that the scaling test was restricted to only eight and sixteen nodes on the GPU cluster, we extrapolate the performance numbers of the test case "4 nodes" on the GPU cluster based on the numbers of the test cases "8 nodes" and 16 nodes" ($time\ on\ 4\ nodes = time\ on\ 8\ nodes * ((time\ on\ 8\ nodes)/(time\ on\ 16\ nodes))$).
We illustrate this comparison in the figure 8.36: the reader can notice that, under these considerations, the APU cluster (with the `cggc` DPS) offers the same performance as the GPU cluster. Besides, one can notice the 3× fold (mentioned earlier in the previous

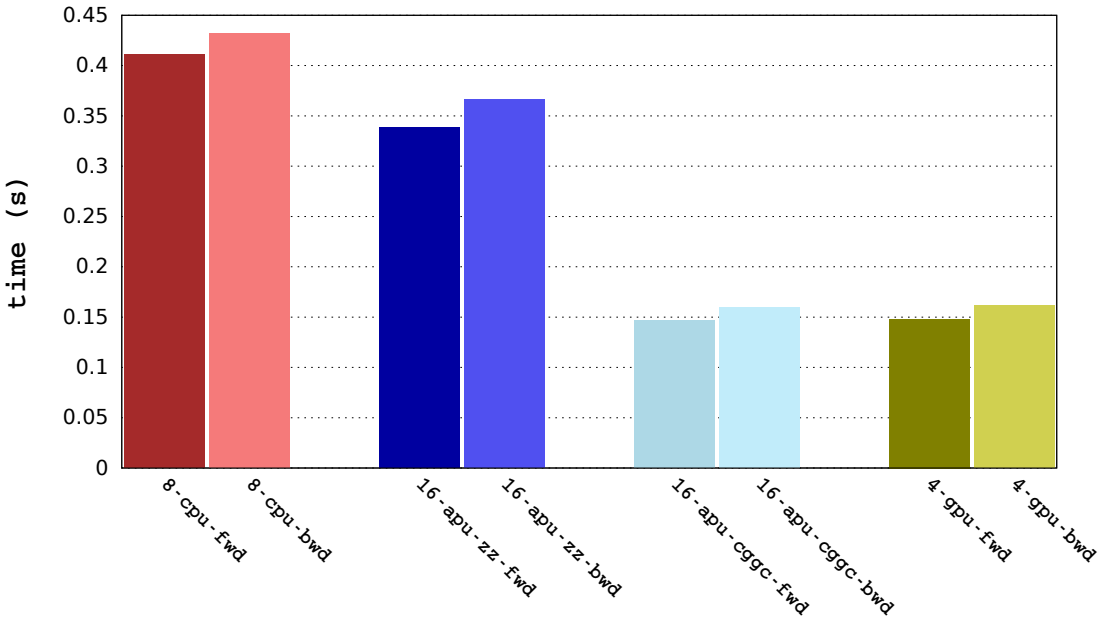



FIGURE 8.36: **Comparison** of the execution times of the large scale **seismic migration** implementation on the CPU cluster, on the GPU cluster and on the APU cluster (with the `cggc` and `zz` DPSs), based on an estimation of maximum **power consumption**.

comparison, regarding the socket to socket comparison) between the performance of the APU cluster (with the `cggc`) and the CPU cluster. Moreover, even with the `zz` DPS, the APU cluster outperforms the CPU cluster from a power efficiency standpoint (we recall that the power consumption of the CPU cluster is underestimated in this comparison).

## 8.4 Conclusion

After going through the performance results of the seismic modeling and of the seismic migration (RTM) on the CPU cluster (flat MPI+Fortran implementation) and on the HWAs based cluster (MPI+Fortran+OpenCL implementation), we can conclude that optimizing and accelerating the wave equation solver on the node level is not enough to achieve a good scaling on the large scale. The different implementations of the seismic applications, were subject to a high communications rate, and to an extensive I/O usage (in the case of the RTM). Adapting these implementations to the HWAs, had introduced additional complexities to the application workflows, such as managing the GPUs, the traffic between the CPU and GPU, etc. Therefore, a special care had to be put in order to reduce the OpenCL kernels overheads, to improve the CPU code managing the GPUs and more importantly to optimize the MPI communications since accelerating the computations on HWAs makes the communications become a bottleneck.
The performance numbers have shown that overlapping the MPI communications with the computation (on CPU, on APU or on GPU) improved the performance of the seismic modeling, and thus that of the RTM, in most the test cases. More importantly, the overlap technique was more efficient on the GPU and the APU clusters, as the CPUs on those clusters were often idle and fully dedicated to the MPI communications and

to drive the computations on the accelerators. Besides, the results have reported that relying on the Fortran 2003 asynchronous I/O engine (only with Intel compilers), had strongly alleviated one of the performance bottlenecks of the RTM which is the extensive I/O usage (depending on the data snapshotting frequency), especially on the HWAs based clusters thanks to lightweight load on the CPUs that were fully dedicated to MPI communications and to the asynchronous I/O engine (in background). Moreover, the performance numbers of the seismic applications on the GPU cluster have informed that the memory traffic between the CPU and the GPU may hinder achieving an optimal scaling when the number of nodes increases. We recall that optimizations techniques such as temporal blocking and overlapping the PCI memory transfers with computations, could have been used in order to mitigate this overhead, but those are not covered by the scope of this work.

Thanks to the APU, the overhead associated to transferring data between the CPU and the GPU is reduced. When using zero-copy memory objects, the overhead is totally suppressed resulting in achieving an optimal scaling in general, at the expense of lower computation performance as compared to the performance of the GPU cluster.

Finally, we have shown that as far as both the performance and the power efficiency are concerned, the APU delivers the same performance results as the discrete GPU.

# Chapter 9

# Conclusions and perspectives

**Contents**



This chapter provides a succinct review of the thesis objectives as well as a summary of our contributions, before closing with a mention to some perspective points and the future work related to this research.

## 9.1 Conclusions

In their, more complex and more challenging than ever, search for new hydrocarbons deposits, Oil and Gas big players are constantly seeking for compute powerful infrastructures and highly optimized software tools to deploy sophisticated and compute-intensive seismic workloads in the most efficient way possible. The seismic imaging algorithms are, and will continue, to be driven by the technology advances offered by HPC. Solutions based on hardware accelerators such as GPUs are widely embraced by the industry, to enhance the accuracy and the performance of imaging workflows such as the RTM. Such workloads are known to have extensive computational demands, to have a high memory requirement and to be I/O-intensive. GPU solutions have shown some weaknesses, in terms of memory limitation, possible restraining overhead due to the PCI Express bus and high power consumption in some cases.

The main objective of this thesis was to assess the relevance of the APU, a new hardware accelerator with a brand new design that fuses a CPU and a GPU in the same socket, in a seismic exploration context. Besides, this work tended to classify the APU in the HPC ecosystem, by means of a thorough study of a selection of seismic applications (modeling and RTM) on the node level, and on the large scale level (clusters of up to 64 nodes). Ultimately, this study had shown that the Kaveri APU (A10-7850K) offers a 3.2× (socket to socket comparison) speedup of the large scale RTM implementation as compared to the high-end Intel Xeon E5-2670 CPU. Besides, albeit an order of magnitude behind recent high-end GPUs, at the time of writing (AMD Tahiti GPU, and NVIDIA Tesla K40s GPU), in terms of compute power and memory bandwidth, the APU falls behind the discrete GPU only by a factor of 3.5× (as compared

to the NVIDIA Tesla K40s GPU). Moreover, when both the performance and power consumption are concerned in the large scale, the APU had shown a great interest as it delivers the same performance as the GPU in this case.

In details, throughout this work we have demystified the new memory model of the APU and introduced practical recommendations in terms of data placement strategies. We have shown, with the help of memory and applicative benchmarks that applications performance could be cut to half when using Onion bus instead of Garlic bus. In particular, we underlined the impact of the data snapshotting frequency, a performance factor that is relevant to the RTM algorithms with selective checkpointing, on the OpenCL performance of the stencil computations (a building block of the seismic applications). We had seen that the APU offers more than an order of magnitude higher performance than the CPU, and that for a high frequency of data snapshotting, which would help deliver more accurate final output images of the RTM algorithm, the APU outperforms the GPU. Besides, we had shown that the APU is more power efficient than both the CPU and the GPU when benchmarking a 3D finite difference application and a matrix multiply kernel. Moreover, we proposed a hybrid (CPU+GPU) strategy to deploy OpenCL workloads on the APU, and the interest of the hybrid utilization has been justified, as we showed a 30% of performance enhancement for a high order 3D finite difference application, with a high rate of memory and computation divergence.

At the node level, the seismic modeling and the RTM applications were highly optimized in OpenCL. The computation kernels of the seismic modeling delivered performance results on the Kaveri APU 4.5× lower, than the performance on the Tahiti GPU, but outperformed the Phenom CPU by a factor of 13×. However, when the power consumption is taken into consideration, we had found the APU 1.8× more power efficient than the GPU when running the RTM on the node level. We also evaluated a directive based approach of the seismic applications on the node level, using OpenACC and HMPPcg. We came to the conclusion that OpenACC (with the help of the HMPPcg workbench) delivered half the performance obtained using OpenCL, but with having added 26× less lines of code than in the OpenCL implementations.

At the large scale, MPI+OpenCL implementations of the seismic modeling and the RTM applications were given. A comparative study between an Intel Xeon E5-2670 CPU cluster (64 nodes), an NVIDIA Tesla K40s GPU cluster (16 nodes), and an AMD A10-7850K APU cluster (16 nodes) was conducted. We had shown that overlapping MPI communications with computations has resulted in improving the scaling of the seismic modeling on the three clusters. The improvement was more significant on the GPU and APU clusters, where in the one hand the computations time was reduced letting the communications time becoming a higher fraction of the overall time, and where the CPUs were idle most of the time, and thus were fully dedicated to MPI communications while the computations were carried out on the GPUs, on the other hand. Consequently, the scaling of the RTM on the hardware accelerators based clusters is even better than the scaling of the CPU cluster (with placing 16 MPI process per compute node). Besides, in addition to the communication-computation overlap, we demonstrated that asynchronous I/O (with the Intel Fortran compiler only) had helped enhancing the performance and the scaling of the large scale RTM (with a frequency of data snapshotting equal to 10) on the different hardware, the hardware accelerators based clusters in particular since the percentages of the I/O time with respect to the overall time were higher than those reported in the CPU cluster (as the computations are accelerated), and since the CPUs of these clusters had a lightweight load and were

fully used for communications and by the asynchronous I/O engine. Moreover, the performance numbers of the seismic applications on the GPU cluster showed that the memory traffic between the CPU and the GPU, via the PCI Express bus, can hinder achieving an optimal scaling as the nodes count increases. Thanks to the APU, we had shown that the overhead associated to transferring data between the CPU and the GPU is reduced without even using the zero-copy buffers. Moreover, when using zero-copy memory objects the overhead is totally suppressed resulting in an optimal scaling in general, even though the overall performance results of the RTM on the APU cluster was lower than on the GPU cluster. However, when considering the maximum power consumption, on the large scale, the power efficiency of the APU cluster equaled that of the GPU cluster, which we consider a valuable asset, not to mention the gap between the two hardware in terms of cost (at the time of writing, an A10-7850K APU costs roughly $150 whereas the price at launch of the NVIDIA Tesla K40s GPU is $7700).
To the best of our knowledge, this research is the first thorough comparison of APUs against other HPC based systems with GPUs and CPUs at the node level as well as at the large scale, which allowed to validate its feasibility on Oil and Gas applications.

## 9.2 Perspectives

In the near future, one of the perspectives of this research is to survey the upcoming roadmap of the APU technology. Indeed, the "Carrizo" APU is already shipping and the "Zen APU" (also referred to as the "Big APU") is around the corner (scheduled for 2016), with promising technical features. The Carrizo APU incorporates the Onion 3 bus which perfectly unifies, on the hardware level, the memory space between the CPU and GPU, reducing the bandwidth gap between cache-coherent and cache non-coherent memory accesses (today the gap is about 50% and it is rumoured that it will be reduced to 10%), which would reduce the performance gap between using and not using the zero-copy buffers. The Big APU will have a higher count, in terms of compute units, and will also feature HBM memory[1], which would be used as a fourth level of cache and would enhance the performance of memory bound workloads such as the RTM. It is then worthwhile to evaluate the performance of our implementations on the new APUs (evaluating theses APUs would require the use of the OpenCL 2.0 standard, which would also allow to exploit larger amount of memory per compute node).
Besides, a continuation to this work would involve improving the APU large scale implementations of the seismic application by overlapping the data packing and unpacking (this also would be only possible with the OpenCL 2.0 standard) with the computations, and by reducing the temporary data copies, during the latter operations, by means of the MPI derived data types for example. Enhancements to the GPU implementations can also be added, namely temporal blocking and overlapping the PCI memory transfers with computations.
Furthermore, as perspective we can evaluate the large scale implementations on a larger number of compute nodes, and with different performance parameters (namely with higher and lower data snapshotting frequencies), which would impact the parallel efficiency of the GPU cluster, due to the PCI transfers, and would possibly change the conclusions of our comparative study.

[1] High Bandwidth Memory which is a RAM interface for 3D-stacked DRAM memory that sits on top of a processor socket.

In addition, with the continuous advances that are integrated to the OpenACC standard (the standard 3.0 is expected for the beginning of 2016), it would be interesting to ameliorate our directive based approach, given the interesting trade-off between the performance results and the programming efforts that it presents.

When it comes to long-term perspectives, HPC architectures are in a constant evolution in the pursuit of exascale computing. While the path seems difficult for CPU based solutions, hardware accelerators are in their way to be an essential ingredient in today's and future supercomputers, and are even tending to be used as standalone processors (except for the discrete GPUs). For example, NVIDIA is incorporating the NVLink technology, to its Tesla GPU lineup. IBM is also adding the technology to the POWER CPUs. According to NVIDIA, the NVLink is a high bandwidth and energy efficient interconnect that replaces the PCI technology and enables fast interactions, 5 to 12 times faster than the PCI Express Gen 3 interconnect, between CPU and GPU and even between GPUs in the same compute node. As a matter of fact the U.S. Department of Energy has already unveiled its ambition to build two GPU and NVLink powered supercomputers, "Sierra" and "Summit", promising a breakthrough in terms of performance (estimated to achieve between 100 and 300 petaflops of peak performance) and energy efficiency, and which will likely top the list of the TOP500 ranking. The NVLink technology will probably alleviate the PCI bottleneck that restrains many scientific codes, seismic applications included, which will be considered as an important asset for GPU based systems.
Furthermore, the OpenPOWER foundation is a an emerging collaboration between 113 members such as IBM, NVIDIA, Google, Mellanox and Altera to name few, which mission is to conceive and build future hybrid architectures around the IBM POWER processor technology. Systems with POWER CPUs connected to NVIDIA GPUs via the NVLink interconnect, and others combining POWER processors with Altera FPGAs (leveraging OpenCL) are already announced. Those may be profitable to the RTM workload since the overhead due to transferring data back and forth between the CPU and the hardware accelerator (a GPU or an FPGA in this case) is expected to be considerably reduced.
That being said, for the APU to become a competitive solution in the HPC ecosystem, we believe that AMD has to see bigger than the "Big APU". The APU has already the advantage of being a standalone processor, has a memory design that tend to unify the CPU and the GPU memory spaces, and is already considered as an energy efficient chip compared to the other HPC processors. However, the APU roadmap should follow the advances of hybrid architectures based on discrete GPUs especially in terms of memory bandwidth. As far as the seismic workloads are concerned, in addition to most scientific codes that are based on explicit numerical solvers, the computations are suitable to massively parallel architectures. Therefore, it would be no waste if APUs will feature a higher density of GPU compute units rather than adding CPU cores from an APU generation to another (it is rumoured that the Big APU would incorporate 16 *CPU* cores and about 40 GPU compute units, while 4 cores and almost 50 GPU compute units would be a more suitable configuration for the explicit numerical solvers). In this thesis, it had been shown that in spite of the small number of cores and the relatively low frequency, CPUs played their role perfectly since one core was dedicated to MPI communications, one core to asynchronous I/O, and one core to drive the GPUs computations (the APU we used has only 4 CPU cores). It seems to me that future APUs should dedicate the major piece of silicon to the GPU compute units and a minor proportion to the CPU cores (maximum of 4 cores), while lowering the cores frequencies if necessary

to remain energy efficient. Besides, since adding more compute units implies more memory contentions, we believe that the APU technology should incorporate more memory channels (say 4 for example) featuring faster memory such as DDR4, and should find the right trade-off between the number of GPU cores and the memory latency. This, in addition to the HBM memory, would be very profitable to the seismic workloads whose performance is driven by the maximum memory bandwidth. Moreover, we had shown that APU based solutions allows a higher density, in terms of processors, compared to GPU based solutions for the same energy envelope. Putting together multiple APUs on a server blade in an SMP fashion (with sharing a high level cache if possible) would enhance the power consumption because some hardware components would be common to all the APUs on the blade, and would also impact the seismic workloads by offering the ability to process a large shot per server blade for example.
On the programmability and software engineering related to APUs, we hope that the OpenACC standard will include a set of directives that takes into consideration the nature of the zero-copy memory buffers (especially that other architectures similar to the APU are emerging), which would considerably ease programming on such processors and make it as simple as OpenMP. As a matter of fact, OpenMP 4.0 is meant to target, in addition to multicore CPUs, the accelerators which raises the question about merging the two specifications in order to target a wider range of users.
From an algorithmic standpoint, it is rather preferable to adapt more realistic RTM algorithms to the hardware accelerators based solutions. The elastic RTM algorithm (isotropic or anisotropic), for example, has the advantage to deliver images with an increased continuity and with sharpened dipping subsalt events, which better represents the elastic property of the earth. This kind of algorithms has a higher compute intensity and different memory requirements, and it would be interesting to explore how applicable it is to APUs. We believe that it would raise much more enthusiasm within the HPC community and within the Oil and Gas industry.

# Bibliography


[1] AMD "Carrizo" architecture. AMD. URL http://www.amd.com/en-us/who-we-are/corporate-information/events/isscc.

[2] State of the Part: CPUs. AnandTECH. URL http://www.anandtech.com/show/8312/state-of-the-part-cpus.

[3] Coarrays Fortran. URL http://www.g95.org/.

[4] CUDA Parallel Computing Platform. NVIDIA. URL http://www.nvidia.com/object/cuda_home_new.html.

[5] Forget Moore's law: Hot and slow DRAM is a major roadblock to exascale and beyond. EXTREMETECH. URL http://www.extremetech.com/computing/185797-forget-moores-law-hot-and-slow-dram-is-a-major-roadblock-to-exascale-and-beyond.

[6] The GREEN 500. URL http://www.green500.org.

[7] Intel 'Haswell' Xeon E5s Aimed Squarely at HPC. HPCWire. URL http://www.hpcwire.com/2014/09/08/intel-haswell-xeon-e5s-aimed-squarely-hpc/.

[8] Kalray. URL http://www.kalrayinc.com/kalray/products/.

[9] TOP 500, The list. URL http://www.top500.org.

[10] The UPC Language. URL http://upc.lbl.gov/.

[11] The OpenMP API specification for parallel programming, version 3.0, 2008. URL http://www.openmp.org/mp-documents/spec30.pdf.

[12] MPI: A message-passing interface standard. version 2.2, 2012. URL http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[13] OpenACC: Directives for Accelerators, version 2.0, 2013. URL http://www.openacc.org/.

[14] The OpenMP API specification for parallel programming, version 4.0, 2013. URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[15] Param Yuva-II: India's fastest supercomputer, 2013. URL http://cdac.in/index.aspx?id=pk_itn_spot849.

[16] Understanding the 80 Plus Certification, 2015. URL http://www.hardwaresecrets.com/understanding-the-80-plus-certification.

[17] Altera FPGAs, 2015. URL https://www.altera.com/products/fpga/overview.html.

[18] CUDA C programming guide. NVIDIA, 2015. URL https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

[19] OpenCL: The open standard for parallel programming of heterogeneous systems, version 2.0. Khronos, 2015. URL https://www.khronos.org/opencl/.

[20] Xilinx, all programmable FPGAs, 2015. URL http://www.xilinx.com/products/silicon-devices/fpga.html.

[21] AAPG. Cutting Costs Without Quality Cuts, 2007. URL https://www2.aapg.org/explorer/2007/06jun/montana_seismic.cfm.

[22] R. Abdelkhalek. *Accélération matérielle pour l'imagerie sismique : modélisation, migration et interprétation.* PhD thesis, Université Sciences et Technologies - Bordeaux I, 2013.

[23] R. Abdelkhalek, H. Calandra, O. Coulaud, G. Latu, and J. Roman. Fast seismic modeling and reverse rime migration on a graphics processing unit Cluster. *Concurrency and Computation: Practice and Experience*, 24(7):739–750, 2012.

[24] K. Aki and P.G. Richard. *Quantitative Seismology : Second Edition.* University Science Books, 2002.

[25] M. Al-husseini. The debate over Hubbert's Peak: a review. *GeoArabia*, 11(2), 2006.

[26] F. Alted. Why modern CPUs are starving and what can be done about it, 2010. URL http://www.blosc.org/docs/StarvingCPUs-CISE-2010.pdf.

[27] Z. Alterman and F. C. Karal. Propagation of elastic waves in layered media by finite difference methods. 58(1):367–398, 1968.

[28] AMD. A Brief History of General Purpose (GPGPU) Computing. URL http://www.amd.com/uk/products/technologies/stream-technology/opencl/pages/gpgpu-history.aspx.

[29] AMD. Accelerated Parallel Processing OpenCL User Guide, 2014.

[30] J.E. Anderson, L. Tan, and D. Wang. Time-Reversal Methods for RTM and FWI. *Society of Exploration Geophysicists*, 2011.

[31] J.E. Anderson, L. Tan, and D. Wang. Time-reversal checkpointing methods for RTM and FWI. *Geophysics*, 77(4):S93, 2012.

[32] M. Araya-Polo, F. Rubio, R. de la Cruz, M. Hanzich, J.M. Cela, and D.P. Scarpazza. High-Performance Seismic Acoustic Imaging by Reverse-Time Migration on the Cell/B.E. Architecture. *ISCA2008 - WCSA2008 / Scientific Programming Special Issue on High Performance Computing on Cell B.E. Processors*, 2008.

[33] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericas, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguade, J.M. Cela, and M. Valero. Assessing Accelerator-Based HPC Reverse Time Migration. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):147–162, 2011.

[34] V. Arslan, J.Y. Blanc, M. Tchiboukdjian, P. Thierry, and G. Thomas-Collignon. Design and Performance of an Intel Xeon Phi based Cluster for Reverse Time Migration. *EAGE*, 2014.

[35] C. P. Ashton, B. Bacon, A. Mann, N. Moldoveanu, T. Sinclair, and G. Redekop. 3D Seismic Survey Design. *Oilfield Review*, 1994.

[36] C. Baldassari. *Modelling and numerical simulation for land migration by wave equation*. PhD thesis, Université de Pau et des Pays de l'Adour, 2009.

[37] E. Baysal, D. Kosloff, and J. Sherwood. Reverse time migration. *Geophysics*, 48 (11):1514–1524, 1983.

[38] H. Ben Hadj Ali. *Three dimensional visco-acoustic frequency-domain full waveform inversion*. PhD thesis, Université Nice-Sophia-Antipolis, 2009.

[39] J.-P. Berenger. A perfectly matched layer for the absorption of electromagnetic waves. *J. Comput. Phys.*, 114(2):185–200, 1994.

[40] S. Bihan, Moulard G.-E., R. Dolbeau, H. Calandra, and R. Abdelkhalek. Directive-based Heterogeneous Programming A GPU-Accelerated RTM Use Case. Technical report, Total S.A., 2009.

[41] L.B. Biondi. *3D Seismic Imaging*. Society of Exploration Geophysicists, 2006.

[42] R. Bording and L. Lines. *Seismic Modeling and Imaging with the Complete Wave Equation*. Society of Exploration Geophysicists, 1997.

[43] S. Breuer, M. Steuwer, and S. Gorlatch. Extending the SkelCL Skeleton Library for Stencil Computations on Multi-GPU Systems. Departement of Mathematics and Computer Science University of Münster, 2014.

[44] J. Brittan, J. Bai, H. Delome, C. Wang, and D. Yingst. Full waveform inversion – the state of the art. *SEG Technical Program Expanded Abstracts*, 31:75–81, 2013.

[45] R. J. Brown, R. R. Stewart, J. E. Gaiser, and D. C. Lawton. An acquisition polarity standard for multicomponent seismic data. 12(1), 2000.

[46] C. Bryan. OpenCL optimization case study support vector machine training. AMD, 2011.

[47] I. Buck. The Evolution of GPUs for General Purpose Computing. NVIDIA, 2010. URL http://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf.

[48] H. Calandra. La puissance de calcul au service de la géophysique. *Technoscoop*, 44:85–89, 2006.

[49] G. Calandrini, A. Gardel, I. Bravo, P. Revenga, et al. Power Measurment Methods for Energy Efficient Applications. *sensors*, 2013.

[50] J. Carcione, G. Herman, and A. ten Kroode. Seismic modeling. *GEOPHYSICS*, 67(4):1304–1325, 2002.

[51] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 213–222. ACM, 2004.

[52] A.J. Chorin. Numerical solution of the Navier-Stokes Equations. *Math. Comp.*, 22:745–762, 1968.

[53] G. Chow, A. Tse, Q. Jin, W. Luk, P. Leong, and D.B. Thomas. A Mixed Precision Monte Carlo Methodology for Reconfigurable Accelerator Systems. ASSOC COMPUTING MACHINERY, 2012.

[54] M. Christen, S. Olaf, P. Messmer, E. Neufeld, and H. Burkhart. Accelerating Stencil-Based Computations by Increased Temporal Locality on Modern Multi- and Many-Core Architectures. In *Proc. of First International Workshop on New Frontiers in High-performance and Hardware-aware Computing*, 2008.

[55] C. Chu and P. Stoffa. Implicit finite-difference simulations of seismic wave propagation. *GEOPHYSICS*, 77(2):T57–T67, 2012.

[56] H. Chuan, Z. Wei, and L. Mi. Time Domain Numerical Simulation for Transient Waves on Reconfigurable Coprocessor Platform. In *In proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2005*, pages 127–136, 2005.

[57] C. Chunlei, P.L. Stoffa, and S. Roustam. 3D Seismic Modeling And Reverse-Time Migration With the Parallel Fourier Method Using Non-blocking Collective Communications. *SEG*, 2009.

[58] J.F. Claerbout. Toward a unified theory of reflector mapping. *Geophysics*, 36(3): 467–481, 1971.

[59] J.F. Claerbout. *Fundamentals of Geophysical Data Processing*. Blackwell, 1976.

[60] J.F. Claerbout. *Imaging the Earth's Interior (IEI)*. Blackwell, 1985.

[61] R.G. Clapp. Reverse time migration with random boundaries. In *79th Annual International Meeting, SEG Expanded Abstracts*, volume 28, pages 2809–2813, 2009.

[62] R.G. Clapp. Reverse time migration: Saving the boundaries, 2009.

[63] Robert G. Clapp, F. Haohuan, and O. Lindtjorn. Selecting the right hardware for Reverse Time Migration. *Society of Exploration Geophysicists*, (January), 2010.

[64] G. Cocco and A. Cisternino. Device specialization in heterogeneous multi-GPU environments. In *2012 Imperial College Computing Student Workshop*, 2012.

[65] J.A. Coffeen. *Seismic Exploration Fundamentals*. Pennwell Books, 1986.

[66] Colorado School of Mines. The New SU User's Manual, 2008. URL `ftp://ftp.cwp.mines.edu/pub/cwpcodes/sumanual_600dpi_a4.pdf`.

[67] M.. Commer and G. Newman. A parallel finite-difference approach for 3D transient electromagnetic modeling with galvanic sources. *Geophysics*, 69(5):1192–1202, 2004.

[68] Platform Computing Corporation. GreenHPC - Dynamic Power Management in HPC. Technical report, 2010.

[69] C. Couder-Castañeda, C. Ortiz-Alemán, M. Gabriel, M. Orozco-del Castillo, and M. Nava-Flores. TESLA GPUs versus MPI with OpenMP for the ForwardModeling of Gravity and Gravity Gradient of Large Prisms Ensemble. *Journal of Applied Mathematics*, 2013.

[70] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM J. Res. Dev.*, 11(2):215–234, 1967.

[71] M. Daga, A.M. Aji, and W. Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *Symposium on Application Accelerators in High-Performance Computing*, 2011.

[72] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12, 2008.

[73] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and Y. KYelick. Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[74] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning the 27-point stencil for multicore. In *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[75] M.C. Delorme, T.S. Abdelrahman, and C. Zhao. Parallel radix sort on the AMD Fusion Accelerated Processing Unit. In *International Conference on Parallel Processing, 2013. ICPP 2013*, 2013.

[76] D. A. Donzis and K. Aditya. Asynchronous Finite-difference Schemes for Partial Differential Equations. *J. Comput. Phys.*, 274:370–392, 2014.

[77] Douglas N.A. Stability, consistency, and convergence of numerical discretizations, 2014. URL http://www.ima.umn.edu/~arnold//papers/stability.pdf.

[78] Drijkoningen, G.G. Exploration seismics. URL http://geodus1.ta.tudelft.nl/PrivatePages/G.G.Drijkoningen/LectureNotes/SeismicProcessing2%28tg001_ta3600%29.pdf.

[79] J.P. Durbano and F.E. Ortiz. FPGA-based acceleration of the 3D finite-difference time-domain method. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 156–163. IEEE, 2004.

[80] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, Rajiv K. K., A. Nakano, and P. Vashishta. A Multilevel Parallelization Framework for High-Order Stencil Computations High-Order Stencil Application. *europa*, 2009.

[81] H. Dursun, K.-I. Nomura, L. Peng, R. Seymour, W. Wang, Rajiv K. K., A. Nakano, P Vashishta, K. Manaschai, and P. Liu. In-Core Optimization of High-Order Stencil Computations. In Hamid R. Arabnia, editor, *PDPTA*, pages 533–538. CSREA Press, 2009.

[82] E. Dussaud, W.W. Symes, and P. Williamson. Computational strategies for reverse-time migration. *78th SEG Annual Meeting*, pages 2267–2271, 2008.

[83] CAPS Entreprise. HMPP Workbench, a directive-based compiler for hybrid computing, 2007. URL http://caps-entreprise.com/fr/page/index.php?id=49&p_p=36.

[84] A. Farjallah. *Preparing Depth Imaging Applications for Exascale Challenges and Impacts*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2014.

[85] R.P. Fletcher and O.A. Robertsson. Time-varying boundary conditions in simulation of seismic wave propagation. *Geophysics*, 76(1):A1–A6, 2011.

[86] D. Foltinek, D. Eaton, J. Mahovsky, P. Moghaddam, and R. McGarry. Industrial-scale reverse time migration on gpu hardware. In *2009 SEG Annual Meeting*, 2009.

[87] M. Frigo and V. Strumpen. Cache Oblivious Stencil Computations. IBM, 2005.

[88] H. Fu and R.G. Clapp. Eliminating the memory bottleneck: an FPGA-based solution for 3d reverse time migration. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 65–74. ACM, 2011.

[89] G.H.F. Gardner. *MIGRATION OF SEISMIC DATA*. Society Of Exploration Geophysicists, 1985.

[90] J. Gazdag. Wave-equation migration with the phase-shift method. *Geophysics*, 43:1342–1351, 1978.

[91] R. Ge, X. Feng, H. Pyla, K. Cameron, and W. Feng, 2007. Power Measurement Tutorial for the Green500 List.

[92] S. Ghosh, S. Chandrasekaran, and B. Chapman. Energy Analysis of Parallel Scientific Kernels on Multiple GPUs. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, 2012.

[93] S. Ghosh, T. Liao, H. Calandra, and B. M. Chapman. Performance of CPU/GPU compiler directives on ISO/TTI kernels. *Computing*, 96(12):1149–1162, 2013.

[94] S.H. Gray, J. Etgen, J. Dellinger, and D. Whitmore. Seismic migration problems and solutions. *Geophysics*, 66:1640, 2001.

[95] A. Griewank. Achieving Logarithmic Growth Of Temporal And Spatial Complexity In Reverse Automatic Differentiation, 1992.

[96] A. Griewank and A. Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, 2000.

[97] The Portland Group. PGI Fortran and C Accelarator Programming Model, 2010. URL http://www.pgroup.com/resources/accel.htm.

[98] H. Guan, Z. Li, B. Wang, and Y. Kim. A Multi-Step Approach for Efficient Reverse-Time Migration. *Expanded Abstracts of 78th Annuel SEG Mtg*, pages 2341–2345, 2008.

[99] A. Guitton. Shot-profile migration of multiple reflections. *SEG Technical Program Expanded Abstracts*, 21(1):1296–1299, 2002.

[100] Chung-Hsing H. and S.W. Poole. Power measurement for high performance computing: State of the art. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –6, 2011.

[101] E. Hager. Full Azimuth Seismic Acquisition with Coil Shooting. $8^{th}$ *Biennial International Conference & Exposition on Petroleum Geophysics*, 2010.

[102] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2011.

[103] G. Hager, H. Stengel, G. Wellein, J. Treibig, M. Wittmann, and M. Meier. MPI+OpenMP hybrid computing (on modern multicore systems). Technical report, 39th Speedup Workshop on High-Performance Computing, 2010.

[104] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein. Prospects for Truly Asynchronous Communication with Pure MPI and Hybrid MPI/ OpenMP on Current Supercomputing Platforms. 2011.

[105] D. Hale. Migration by the Kirchhoff, slant stack and Gaussian beam methods. Technical report, 1999. URL http://cwp.mines.edu/Documents/cwpreports/cwp-126.pdf.

[106] M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, B. Paven, P.C. Diniz, et al. ASCR Programming Challenges for Exascale Computing. 2011.

[107] B. Hamilton and C.-J. Webb. *Room acoustics modelling using GPU-accelerated finite difference and finite volume methods on a face-centered cubic grid*. 2013.

[108] T.D. Han and T.S. Abdelrahman. hiCUDA: High-Level GPGPU Programming. *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):78–90, 2011.

[109] S. Hauck and A. DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., 2007.

[110] J.L. Hennessy and D.A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011.

[111] L.I.U Hong-wei, L.I. Bo, L.I.U. Hong, T. Xiao-long, and L.I.U. Qin. The algorithm of high order finite difference pre-stack reverse time migration and gpu implementation. *Chinese Journal of Geophysics*, 53(4):600–610, 2010.

[112] L. Hongwei, D. Renwei, L. Lu, and L. Hong. Wavefield reconstruction methods for reverse time migration. *Journal of Geophysics and Engineering*, 10(1):015004, 2013.

[113] C.-H. Hsu, W.-C. Feng, and J.S. Archuleta. Towards Efficient Supercomputing: A Quest for the Right Metric. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11 - Volume 12*, IPDPS '05, pages 230.1–. IEEE Computer Society, 2005.

[114] S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. *International Symposium on Parallel & Distributed Processing*, 2009.

[115] IAGC. An overview of marine seismic operations. Technical report, 2011.

[116] IFP. Marmousi model and data set, 1993. URL http://sw3d.cz/software/marmousi/marmousi.htm.

[117] Intel. Threading Building Blocks (TBB). URL https://www.threadingbuildingblocks.org/.

[118] D. Jacobsen, J. Thibault, and I. Senocak. An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. American Institute of Aeronautics and Astronautics, 2010.

[119] Z. Jiang, K. Bonham, J.C. Bancroft, and L.R. Lines. Overcoming computational cost problems of reverse-time migration. *GeoCanada*, 2010.

[120] G. Jin, T. Endo, and S. Matsuoka. A Multi-Level Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPU. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1080–1087, 2013.

[121] Guanghao Jin, Toshio Endo, and Satoshi Matsuoka. A Multi-Level Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPU. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1080–1087, 2013.

[122] J. Jossey and A.N. Hirani. Equivalence theorems in numerical analysis: integration, differentiation and interpolation, 2000.

[123] B. Kaelin and A. Guitton. Illumination effects in reverse time migration. *EAGE*, (June), 2007.

[124] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM J. Res. Dev.*, 49(4/5), 2005.

[125] T. H. Kaiser and S. B. Baden. Overlapping Communication and Computation with OpenMP and MPI. *Sci. Program.*, 9(2,3):73–81, 2001.

[126] S. Kane. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Trans. Antennas and Propagation*, pages 302–307, 1966.

[127] P. Kearey, M. Brooks, and I. Hill. *An Introduction to Geophysical Exploration.* Blackwell Science, 2002.

[128] K. Kelly, R. Ward, S. Treitel, and R. Alford. Synthetic Seismograms: A Finite-Difference Approach. *GEOPHYSICS*, 41(1):2–27, 1976.

[129] Khronos Group. The OpenGL Specification version 4.3, 2012.

[130] Y. Kim, Y. Cho, U. Jang, and C. Shin. Acceleration of Stable TTI P-wave Reverse-time Migration with GPUs. *Comput. Geosci.*, 52:204–217, 2013.

[131] M. Klemm. 23 tips for performance tuning with the Intel MPI Library, 2010. URL https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/03%20MPI%20Tuning.pdf.

[132] M. Klemm. Intel Hyper-Threading Technology Technical User's Guide, 2012. URL https://www.utdallas.edu/~edsha/parallel/2010S/Intel-HyperThreads.pdf.

[133] S. Koliaï, Z. Bendifallah, M. Tribalat, C. Valensi, J.-T. Acquaviva, and W. Jalby. Quantifying Performance Bottleneck Cost Through Differential Analysis. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 263–272. ACM, 2013.

[134] D. Komatitsch and R. Martin. An unsplit convolutional perfectly matched layer improved at grazing incidence for the seismic wave equation. *Geophysics*, 72(5): SM155–SM167, 2007.

[135] D. Kosloff and E. Baysal. Forward modeling by a Fourier method. *GEOPHYSICS*, 47(10):1402–1412, 1982.

[136] R. Kosloff and D. Kosloff. Absorbing Boundaries for Wave Propagation Problems. *J. Comput. Phys.*, 63(2):363–376, 1986.

[137] H. Kronawitter, S. Stengel, G. Hager, and C. Lengauer. Domain-Specific Optimization of Two Jacobi Smoother Kernels and Their Evaluation in the ECM Performance Model. *Parallel Processing Letters*, 24(3), 2014.

[138] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[139] A. Lemmer and R. Hilfer. Parallel domain decomposition method with non-blocking communication for flow through porous media. *Journal of Computational Physics*, 281:970–981, 2015.

[140] W. Liang, Y. Wang, and C. Yang. Determining finite difference weights for the acoustic wave equation by a new dispersion-relationship-preserving method. *Geophysical Prospecting*, 63(1):11–22, 2015.

[141] L.R. Lines, R. Slawinski, and R.P. Bording. A recipe for stability of finite-difference wave-equation computations. *Geophysics*, 64(3):967–969, 1999.

[142] H. Liu, Bo Li, Hong Liu, X. Tong, Q. Liu, X. Wang, and W. Liu. The issues of prestack reverse time migration and solutions with Graphic Processing Unit implementation. *Geophysical Prospecting*, 60(5):906–918, 2012.

[143] L.T. Lkelle and L. Amundsen. *Introduction to Petroleum Seismology (Investigations in Geophysics)*. Society Of Exploration Geophysicists, 2005.

[144] S.A. Long, R. Van Borselen, and L. Fountain. Surface-Related Multiple Elimination - Applications to an offshore Australia data set. *ASEG Extended Abstracts*, pages 1–4, 2001.

[145] L. Lu and K. Magerlein. Multi-level Parallel Computing of Reverse Time Migration for Seismic Imaging on Blue Gene/Q. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 291–292. ACM, 2013.

[146] E. Lusk and A. Chan. Early Experiments with the OpenMP/MPI Hybrid Programming Model. In R. Eigenmann and R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 36–47. Springer Berlin Heidelberg, 2008.

[147] T. Lutz, C. Fensch, and M. Cole. PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems. *ACM Trans. Archit. Code Optim.*, 2013.

[148] MADAGASCAR. Guide to RSF file format, 2004. URL http://www.reproducibility.org/wiki/Guide_to_RSF_file_format.

[149] T.M. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D.E. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *CoRR*, abs/1410.3060, 2014.

[150] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-scale GPU-accelerated Supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 11:1–11:12. ACM, 2011.

[151] G. A. McMechan. Migration by extrapolation of time-dependent boundary values. *Geophysical Prospecting*, 31:413–420, 1983.

[152] Mellanox Technologies Inc. Introduction to InfiniBand, 2003. URL http://www.mellanox.com/pdf/whitepapers/IB_Intro_WP_190.pdf.

[153] Mellanox Technologies Inc. Mellanox IB-Verbs API (VAPI), 2012. URL http://nuweb12.neu.edu/rc/wp-content/uploads/2013/09/MellanoxVerbsAPI.pdf.

[154] K.C. Meza-Fajardo and A.S. Papageorgiou. A Nonconvolutional, Split-Field, Perfectly Matched Layer for Wave Propagation in Isotropic and Anisotropic Elastic Media: Stability Analysis. *Bulletin of the Seismological Society of America*, 98(4): 1811–1836, 2008.

[155] D. Michéa and D. Komatitsch. Accelerating a 3D finite-difference wave propagation code using GPU graphics cards. *Geophys. J. Int.*, 182(1):389–402, 2010.

[156] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[157] Microsoft. DirectX. URL http://www.microsoft.com/en-us/download/search.aspx?q=directx.

[158] P. Moczo, J. OA. Robertsson, and L. Eisner. The finite-difference time-domain method for modeling of seismic wave propagation. *Advances in Geophysics*, 48: 421–516, 2007.

[159] W.A. Mulder and R.-E. Plessix. A comparison between one-way and two-way wave-equation migration. *Geophysics*, 69(6), 2004.

[160] Muranushi, T. and Makino, J. Optimal Temporal Blocking for Stencil Computation. *Procedia Computer Science*, 2015.

[161] C. Märtin. Post-Dennard Scaling and the final Years of Moore ' s Law Consequences for the Evolution of Multicore-Architectures. *Informatik und Interaktive Systeme*, 2014.

[162] A. Narang, S. Kumar, A.S. Das, M. Perrone, D. Wade, K. Bendiksen, V. Slåtten, and T.E. Rabben. Performance Optimizations for TTI RTM on GPU based Hybrid Architectures. *Biennial International Conference & Exposition*, 2013.

[163] R. Nath, S. Tomov, and J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR'10, 2011.

[164] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 3.5d blocking optimization for stencil computations on modern CPUs and GPUs. In *in Proc. of the 2010 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis, 2010*, pages 1–13, 2010.

[165] U.S. Department of Energy. The Opportunities and Challenges of Exascale Computing. Technical report, 2010. URL http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf.

[166] T. Okamoto, H. Takenaka, T. Nakamura, and T. Aoki. Accelerating Large-Scale Simulation of Seismic Wave Propagation by Multi-GPUs and Three-Dimensional Domain Decomposition. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 375–389. Springer Berlin Heidelberg, 2013.

[167] S. Operto, J. Virieux, and A. Ribodetti. Finite-difference frequency-domain modeling of viscoacoustic wave propagation in 2D tilted transversely isotropic (TTI) media. *Geophysics*, 74(5), 2009.

[168] J. Panetta, T. Teixeira, P.R.P. de Souza Filho, C.A. da Cunha Finho, D. Sotelo, F. da Motta, S.S. Pinheiro, I. Pedrosa, A.L.R. Rosa, L.R. Monnerat, L.T. Carneiro, and C.H.B. de Albrecht. Accelerating Kirchhoff Migration by CPU and GPU Cooperation. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 26–32, 2009.

[169] I. Panourgias. NUMA effects on multicore, multi socket systems, 2011. URL http://static.ph.ed.ac.uk/dissertations/hpc-msc/2010-2011/IakovosPanourgias.pdf.

[170] A. Pedram, R.A. van de Geijn, and A. Gerstlauer. Codesign Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures. *Computers, IEEE Transactions on*, 2012.

[171] M.P. Perrone, L. Lu, L. Liu, I. Fedulova, A. Semenikhin, and V. Gorbik. High Performance RTM Using Massive Domain Partitioning. *EAGE*, 2011.

[172] M.P. Perrone, L. Lu, L. Liu, K. Magerlein, K. Changhoan, I. Fedulova, and A. Semenikhin. Fast Scalable Reverse Time Migration Seismic Imaging on Blue Gene/P. *SC'11*, 2011.

[173] M. Peter, O.A. Johan, and E. Leo. The Finite-Difference Time-Domain Method for Modeling of Seismic Wave Propagation. In *Advances in Wave Propagation in Heterogenous Earth*, volume 48. Elsevier, 2007.

[174] PGS. Multi-Azimuth 3-D Surface-Related Multiple Elimination – Application to Offshore Nile Delta. *Petroleum Geo-Services*, 9, 2009.

[175] S. Phadke, D. Bhardwaj, and S. Yerneni. 3D Seismic Modeling in a Message Passing Environment. Centre for Development of Advanced Computing, Pune University.

[176] R.-E. Plessix. A review of the adjoint-state method for computing the gradient of a functional with geophysical applications. *Geophysical Journal International*, 167(2):495–503, 2006.

[177] R.G. Pratt. Seismic waveform inversion in the frequency domain, Part 1: Theory and verification in a physical scale model. *Geophysics*, 64(3):888–901, 1999.

[178] R. Rastogi, A. Srivastava, K. Sirasala, H. Chavhan, and K. Khonde. Experience of Porting and Optimization of Seismic Modelling on Multi and Many Cores of Hybrid Computing Cluster. *EAGE*, 2015.

[179] G. Ritter and K. Waddell. Resolving Complex Salt Geometry: Iterative Salt Imaging and Interpretation. Technical report, 2012.

[180] G. Rivera and C.-W. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00. IEEE Computer Society, 2000.

[181] E. Robein. *Vitesses et techniques d'imagerie en sismique réflexion*. TEC & DOC Lavoisier, 1999.

[182] E. Robein. *Seismic Imaging: A Review of the Techniques, their Principles, Merits and Limitations*. EAGE Publications bv, 2010.

[183] A.J. Roden and D. Stephen. Convolution PML (CPML): An efficient FDTD implementation of the CFS–PML for arbitrary media. *Microwave and Optical Technology Letters*, pages 334–339, 2000.

[184] Y. Ruan, V. S. Pai, E. Nahum, and J. M. Tracey. Evaluating the Impact of Simultaneous Multithreading on Network Servers Using Real Hardware. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 315–326. ACM, 2005.

[185] I. Said, H. Calandra, and T. Liao. FPGA based technology evaluation: a case study of seismic depth imaging application. Technical report, Total S.A., 2010.

[186] W. A. Schneider. Integral formulation for the migration in two and three dimensions. *Geophysics*, 43:49–76, 1978.

[187] G. Schubert, G. Hager, H. Fehske, and G. Wellein. Parallel Sparse Matrix-Vector Multiplication as a Test Case for Hybrid MPI + OpenMP Programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1751–1758, 2011.

[188] G. T. Schuster. *Basics of Seismic Imaging*. Cambridge University Press, 2010.

[189] G.T. Schuster. Basics of Seismic Wave Theory. Technical report, 2007.

[190] SEG. The 2004 BP Velocity-Analysis Benchmark, 2004. URL http://software.seg.org/datasets/2D/2004_BP_Vel_Benchmark/.

[191] SEG Technical Standards Committee. SEG Y rev 1 Data Exchange format, 2002. URL http://www.seg.org/documents/10161/77915/seg_y_rev1.pdf.

[192] R.L. Sengbush. *Seismic Exploration Methods*. Springer Netherlands, 2012.

[193] Sercel. What on Earth is Geophysics?, 2013. URL http://www.sercel.com/about/Pages/what-is-geophysics.aspx.

[194] P. Sguazzero and J. Gazdag. Migration of Seismic Data. volume 72, 1984.

[195] M. Shafiq, M. Pericas, R. de la Cruz, M. Araya-Polo, N. Navarro, and E. Ayguadé. Exploiting memory customization in fpga for 3d stencil computations. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 38–45. IEEE, 2009.

[196] P.M. Shearer. *Introduction to Sismology*. Cambridge University Press, 2009.

[197] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, and S. Matsuoka. Peta-scale Phase-field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 3:1–3:11. ACM, 2011.

[198] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. MT-MPI: Multithreaded MPI for Many-core Environments. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 125–134. ACM, 2014.

[199] M.R. Simmons. The world's giant oilfields. Technical report, SIMMONS & COMPANY INTERNATIONAL, 2001.

[200] Stanford Exploration Project. SEP Manual, 2004. URL http://sepwww.stanford.edu/lib/exe/fetch.php?media=sep:software:sepman.pdf.

[201] H. Stengel, J. Treibig, G. Hager, and G. Wellein. Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. *CoRR*, abs/1410.5010, 2014.

[202] R.H. Stolt. Migration by Fourier transform. *Geophysics*, 43:23–48, 1978.

[203] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache Accurate Time Skewing in Iterative Stencil Computations. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 571–581. IEEE Computer Society, 2011.

[204] R. Suda and Da Qi R. Accurate Measurements and Precise Modeling of Power Dissipation of CUDA Kernels toward Power Optimized High Performance CPU-GPU Computing. In *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*, 2009.

[205] M. Sugawara, S. Hirasawa, K. Komatsu, H. Takizawa, and H. Kobayashi. A Comparison of Performance Tunabilities between OpenCL and OpenACC. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 147–152, 2013.

[206] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 32–39. ACM, 2006.

[207] W.W. Symes. Reverse time migration with optimal checkpointing. *Geophysics*, 72 (5):SM213–SM221, 2007.

[208] R. Thakur and W. Gropp. Test Suite for Evaluating Performance of Multithreaded MPI Communication. *Parallel Computing*, 35:608–617, 2008.

[209] J.I. Toivanen, T.P. Stefanski, N. Kuster, and N. Chavannes. Comparison of CPML implementations for the GPU-accelerated FDTD solver. *Progress in Electromagnetics Research M*, pages 61–75, 2011.

[210] Top500. The Tsubame-2.0 grid cluster. URL http://www.top500.org/system/176927.

[211] T. Udagawa and M. Sekijima. The Power Efficiency of GPUs in Multi Nodes Environment with Molecular Dynamics. In *Proceedings of the 2011 40th International Conference on Parallel Processing Workshops*, 2011.

[212] O. Villa, D.R. Johnson, M. O'Connor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S.W. Keckler, and W.J. Dally. Scaling the Power Wall: A Path to Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[213] A. Villarreal and J.A. Scales. Distributed three-dimensional finite-diffrence modeling of wave propagation in acoustic media. *COMPUTERS IN PHYSICS*, 1997.

[214] J Virieux and S Operto. An overview of full-waveform inversion in exploration geophysics. *Geophysics*, 74(6):WCC1, 2009.

[215] J. Virieux, H. Calandra, and R.É. Plessix. A review of the spectral, pseudo-spectral, finite-difference and finite-element modeling techniques for geophysical imaging. *Geophysical Prospecting*, 59(5):794–813, 2011.

[216] V. Volkov. Better Performance at Lower Occupancy. GPU Technology Conference, 2010.

[217] V. Volkov. Better performance at lower occupancy. GTC, 2010.

[218] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, 2008.

[219] D. Wade, N. Ankur, J. Soman, K. Bendiksen, and M.P. Perrone. Maximizing TTI RTM Throughput for CPU+GPU. *Biennial International Conference & Exposition*, 2013.

[220] S. Weijia and F. Li-Yun. Two effective approaches to reduce data storage in reverse time migration. *Computers & Geosciences*, 56(0):69–75, 2013.

[221] N. D Whitmore. Iterative depth migration by backward time propagation. *SEG Annual Meeting*, 1983.

[222] M. Wittmann, G. Hager, and G. Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–7, 2010.

[223] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein. Asynchronous MPI for the Masses. *CoRR*, abs/1302.4280:12, 2013.

[224] P. Zandevakili, M. Hu, and Z. Qin. GPUmotif: An Ultra-Fast and Energy-Efficient Motif Analysis Program Using Graphics Processing Units. *PLoS ONE*, 2012.

[225] G. Zumbusch. Tuning a Finite Difference Computation for Parallel Vector Processors. *Parallel and Distributed Computing, International Symposium on*, 0:63–70, 2012.

# List of Figures

# List of Tables

# Appendix A

# List of publications

- H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, I. Said, Assessing the relevance of APU for high performance scientific computing, AMD Fusion Developer Summit (AFDS), 2012.
- H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, I. Said, Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil, 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2013.
- H. Calandra, R. Dolbeau, P. Fortin, J.-L. Lamotte, I. Said, Forward seismic modeling on AMD Accelerated Processing Unit, 2013 Rice Oil & Gas HPC Workshop.
- P. Eberhart, I. Said, P. Fortin, H. Calandra, Hybrid strategy for stencil computations on the APU, The 1st International Workshop on High-Performance Stencil Computations, 2014.
- F. Jézéquel, J.-L. Lamotte, I. Said, Estimation of numerical reproducibility on CPU and GPU, Federated Conference on Computer Science and Information Systems, 2015.
- I. Said, P. Fortin, J.-L. Lamotte and H. Calandra. Leveraging the Accelerated Processing Units for seismic imaging: a performance and power efficiency comparison against CPUs and GPUs. (submitted on October 2015 to an international journal).
- I. Said, P. Fortin, J.-L. Lamotte, H. Calandra, Efficient Reverse Time Migration on APU clusters, 2016 Rice Oil & Gas HPC Workshop (submitted on November 2015).

**Résumé**
Les compagnies pétrolières s'appuient sur le HPC pour accélérer les algorithmes d'imagerie profondeur. Les grappes de CPU et les accélérateurs matériels sont largement adoptés par l'industrie. Les processeurs graphiques (GPU), avec une grande puissance de calcul et une large bande passante mémoire, ont suscité un vif intérêt. Cependant le déploiement d'applications telle la Reverse Time Migration (RTM) sur ces architectures présente quelques limitations. Notamment, une capacité mémoire réduite, des communications fréquentes entre le CPU et le GPU présentant un possible goulot d'étranglement à cause du bus PCI, et des consommations d'énergie élevées. AMD a récemment lancé l'*Accelerated Processing Unit (APU)* : un processeur qui fusionne CPU et GPU sur la même puce via une mémoire unifiée.
Dans cette thèse, nous explorons l'efficacité de la technologie APU dans un contexte pétrolier, et nous étudions si elle peut surmonter les limitations des solutions basées sur CPU et sur GPU. L'APU est évalué à l'aide d'une suite OpenCL de tests mémoire, applicatifs et d'efficacité énergétique. La faisabilité de l'utilisation hybride de l'APU est explorée. L'efficacité d'une approche par directives de compilation est également étudiée. En analysant une sélection d'applications sismiques (modélisation et RTM) au niveau du noeud et à grande échelle, une étude comparative entre CPU, APU et GPU est menée. Nous montrons la pertinence du recouvrement des entrées-sorties et des communications MPI par le calcul pour les grappes d'APU et de GPU, que les APU délivrent des performances variant entre celles du CPU et celles du GPU, et que l'APU peut être aussi énergétiquement efficace que le GPU.
**Mots-clés :** HPC, calcul GPU, architectures hybrides, APU, géophysique, RTM.

**Contributions of hybrid architectures to depth imaging: a CPU, APU and GPU comparative study**

**Abstract**
In an exploration context, Oil and Gas (O&G) companies rely on HPC to accelerate depth imaging algorithms. Solutions based on CPU clusters and hardware accelerators are widely embraced by the industry. The Graphics Processing Units (GPUs), with a huge compute power and a high memory bandwidth, had attracted significant interest. However, deploying heavy imaging workflows, the Reverse Time Migration (RTM) being the most famous, on such hardware had suffered from few limitations. Namely, the lack of memory capacity, frequent CPU-GPU communications that may be bottlenecked by the PCI transfer rate, and high power consumptions. Recently, AMD has launched the Accelerated Processing Unit (APU): a processor that merges a CPU and a GPU on the same die, with promising features notably a unified CPU-GPU memory.
Throughout this thesis, we explore how efficiently may the APU technology be applicable in an O&G context, and study if it can overcome the limitations that characterize the CPU and GPU based solutions. The APU is evaluated with the help of memory, applicative and power efficiency OpenCL benchmarks. The feasibility of the hybrid utilization of the APUs is surveyed. The efficiency of a directive based approach is also investigated. By means of a thorough review of a selection of seismic applications (modeling and RTM) on the node level and on the large scale level, a comparative study between the CPU, the APU and the GPU is conducted. We show the relevance of overlapping I/O and MPI communications with computations for the APU and GPU clusters, that APUs deliver performances that range between those of CPUs and those of GPUs, and that the APU can be as power efficient as the GPU.
**Keywords :** HPC, GPU computing, hybrid architectures, APU, geophysics, RTM.