UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

VÍCTOR EDUARDO MARTÍNEZ ABAUNZA

Performance Optimization of Geophysics Stencils on HPC Architectures

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor:
Prof. Dr. Philippe Olivier Alexandre Navaux
Advisor during Ph.D. internship:
Dr. Fabrice Dupros

Porto Alegre
August 2018

CIP — CATALOGING-IN-PUBLICATION

Martínez Abaunza, Víctor Eduardo

Performance Optimization of Geophysics Stencils on HPC Architectures / Víctor Eduardo Martínez Abaunza. – Porto Alegre: PPGC da UFRGS, 2018.

129 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2018. Advisor: Philippe Olivier Alexandre Navaux; Advisor during Ph.D. internship: Fabrice Dupros.

I. Navaux, Philippe Olivier Alexandre. II. Dupros, Fabrice. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do PPGC: Prof. João Luiz Dihl Comba
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

«(El cabalista que ofició de numen
a la vasta criatura apodó Golem;
estas verdades las refiere Scholem
en un docto lugar de su volumen.)»

EL GOLEM — J. L. BORGES

ACKNOWLEDGMENT

I want to thank to:

My family, you are the reason
My friends, they become my family
My advisers, they become my friends
My colleagues, they become my advisers
and all the people that support me...

You know who you are.

Víctor

ABSTRACT

Wave modeling is a crucial tool in geophysics, for efficient strong motion analysis, risk mitigation and oil & gas exploration. Due to its simplicity and numerical efficiency, the finite-difference method is one of the standard techniques implemented to solve the wave propagation equations. This kind of applications is known as stencils because they consist in a pattern that replicates the same computation on a multi-dimensional domain. High Performance Computing is required to solve this class of problems, as a consequence of a large number of grid points involved in three-dimensional simulations of the underground. The performance optimization of stencil computations is a challenge and strongly depends on the underlying architecture.

In this context, this work was directed toward a twofold aim. Firstly, we have led our research on multicore architectures and we have analyzed the standard OpenMP implementation of numerical kernels from the 3D heat transfer model (a 7-point Jacobi stencil) and the Ondes3D code (a full-fledged application developed by the *French Geological Survey*). We have considered two well-known implementations (naïve, and space blocking) to find correlations between parameters from the input configuration at runtime and the computing performance; thus, we have proposed a Machine Learning-based approach to evaluate, to predict, and to improve the performance of these stencil models on the underlying architecture. We have also used an acoustic wave propagation model provided by the *Petrobras* company and we have predicted the performance with high accuracy on multicore architectures. Secondly, we have oriented our research on heterogeneous architectures, we have analyzed the standard implementation for seismic wave propagation model in CUDA, to find which factors affect the performance; then, we have proposed a task-based implementation to improve the performance, according to the runtime configuration set (scheduling algorithm, size, and number of tasks), and we have compared the performance obtained with the classical CPU or GPU only versions with the results obtained on heterogeneous architectures.

Keywords: HPC. Machine Learning. Multicore. Heterogeneous Architectures. Stencil Computations. Performance Simulation. Performance improvement.

Optimização de Desempenho de Estênceis Geofísicos sobre Arquiteturas HPC

RESUMO

A simulação de propagação de onda é uma ferramenta crucial na pesquisa de geofísica (para análise eficiente dos terremotos, mitigação de riscos e a exploração de petróleo e gáz). Devido à sua simplicidade e sua eficiência numérica, o método de diferenças finitas é uma das técnicas implementadas para resolver as equações da propagação das ondas. Estas aplicações são conhecidas como estênceis porque consistem num padrão que replica a mesma computação num domínio multidimensional de dados. A Computação de Alto Desempenho é requerida para solucionar este tipo de problemas, como consequência do grande número de pontos envolvidos nas simulações tridimensionais do subsolo. A optimização do desempenho dos estênceis é um desafio e depende do arquitetura usada. Neste contexto, focamos nosso trabalho em duas partes. Primeiro, desenvolvemos nossa pesquisa nas arquiteturas multicore; analisamos a implementação padrão em OpenMP dos modelos numéricos da transferência de calor (um estêncil Jacobi de 7 pontos), e o aplicativo Ondes3D (um simulador sísmico desenvolvido pela *Bureau de Recherches Géologiques et Minières*); usamos dois algoritmos conhecidos (nativo, e bloqueio espacial) para encontrar correlações entre os parâmetros da configuração de entrada, na execução, e o desempenho computacional; depois, propusemos um modelo baseado no Aprendizado de Máquina para avaliar, predizer e melhorar o desempenho dos modelos estênceis na arquitetura usada; também usamos um modelo de propagação da onda acústica fornecido pela empresa *Petrobras*; e predizemos o desempenho com uma alta precisão (até 99%) nas arquiteturas multicore. Segundo, orientamos nossa pesquisa nas arquiteturas heterogêneas, analisamos uma implementação padrão do modelo de propagação de ondas em CUDA, para encontrar os fatores que afetam o desempenho quando o número de aceleradores é aumentado; então, propusemos uma implementação baseada em tarefas para amelhorar o desempenho, de acordo com um conjunto de configuração no tempo de execução (algoritmo de escalonamento, tamanho e número de tarefas), e comparamos o desempenho obtido com as versões de só CPU ou só GPU e o impacto no desempenho das arquiteturas heterogêneas; nossos resultados demostram um speedup significativo (até $\times 25$) em comparação com a melhor implementação disponível para arquiteturas multicore.

Palavras-chave: HPC, aprendizado de máquina, multicore, arquiteturas heterogêneas, computação de estênceis, simulação de desempenho, ganho de desempenho.

LIST OF ABBREVIATIONS AND ACRONYMS

ANOVA	Analysis of Variance
APU	Accelerated Processing Architecture
BRGM	French Geological Survey
CM	Cache Misses
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DAG	Directed Acyclic Graph
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DP	Data Parallelism
FDM	Finite Difference Method
EDP	Energy Delay Product
FPGA	Field Programmable Gate Arrays
GPU	Graphics Processing Unit
HC	Heterogeneous Computing
HCS	Heterogeneous Computing System
HT	HyperThreading
HPC	High Performance Computing
ILP	Instruction Level Parallelism
ISA	Instruction-Set Architectures
LLC	Last Level Cache
LU	Lower Upper Decomposition
MIPS	Microprocessors with Interlocked Pipeline Stages
ML	Machine Learning

MLP	Memory Level Parallelism
MPI	Message Passing Interface
MSE	Mean Squared Error
NUMA	Non-Uniform Memory Access
PCIe	Peripheral Component Interconnect Express
PTX	Parallel Thread Execution
PU	Processing Unit
RISC	Reduced Instruction Set Computer
RMSE	Root Mean Square Error
S2S	Source-to-source
SCHP	Single-Chip Heterogeneous Processor
SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessors
SMP	Symmetric Multi-Processing
SMT	Simultaneous Multithreading
SoC	System-on-a-Chip
SVM	Support Vector Machine
TLB	Translation Lookaside Buffer
TLP	Thread-level Parallelism

LIST OF FIGURES

Figure 2.1 Example of the Roofline model for an Intel Xeon (Clovertown) multicore architecture. 32
Figure 2.2 Architeture of Viking multicore machine presented in Table 2.1. 33

Figure 3.1 Size of 7-point Jacobi stencil and its neighbor points. 36
Figure 3.2 Size of seismic stencil to calculate velocity and stress components. 39
Figure 3.3 Size of acoustic wave propagation stencil and its neighbor points. 40
Figure 3.4 Representation of solution by thread, for naive and space tiling algorithms. 42
Figure 3.5 Impact of performance measures by number of threads. 44
Figure 3.6 Impact of performance measures by problem size. 45
Figure 3.7 Impact of performance measures by code optimization. 45
Figure 3.8 Traces of one-time iteration for 7-point stencil execution. 46
Figure 3.9 Impact of performance measures by scheduling policy. 48
Figure 3.10 Impact of performance measures by scheduling and chunk size on Turing machine. 49
Figure 3.11 Impact of performance measures by stencil algorithm. 49
Figure 3.12 Performance vs cache misses by input parameters in Turing machine 50
Figure 3.13 Linear fitting of cache memory by problem size and machine 51
Figure 3.14 Exponential fitting for problem size 52

Figure 4.1 Flowchart of General Model for Performance Prediction of Geophysics Stencils based on Machine Learning (Golem). 56
Figure 4.2 Flowchart of Golem on multicore architectures. 58
Figure 4.3 Hardware counter behavior of 7-point Jacobi on Node Cryo. 61
Figure 4.4 Normalized performance comparison between predicted results from the ML-algorithm and results from the best performance experimens on multicore architectures 64
Figure 4.5 Flowchart of Golem on manycore architectures. 65
Figure 4.6 Hardware counters behavior on Manycore Node. 68
Figure 4.7 Normalized performance comparison between predicted results from the ML-algorithm and results from the best performance experimens on manycore architectures. 70

Figure 5.1 Block diagram of a NVIDIA GPU SM. 74
Figure 5.2 Representation of a grid with the thread blocks. 75
Figure 5.3 Representation of memory hierarchy for threads, blocks and grids. 76
Figure 5.4 Heterogeneous CUDA programming model. 77
Figure 5.5 Architecture of the Fermi SM. 79
Figure 5.6 Architecture of the Kepler SM. 80
Figure 5.7 Architecture of the Maxwell SM. 81
Figure 5.8 Architecture of the Pascal SM. 82
Figure 5.9 Architecture of Tegra machine presented in Table 5.1. 84

Figure 6.1 Tiling division of 3D data domain, each slice is computed by the GPU. 87
Figure 6.2 Timeloop and speedup measures of Ondes 3D application when increasing the number of GPUs. 88
Figure 6.3 Kernel execution and communication time of Ondes 3D application when increasing the number of GPUs. 89

Figure 6.4 Measures (time, load and memory consumption) of Ondes 3D on a cluster of GPUs. 90
Figure 6.5 Statistical estimators (average and standard deviation) of Ondes 3D for GPU load and memory usage. 91

Figure 7.1 Grid of blocks including inner grid-points corresponding to the physical domain and outer ghosts zones 96
Figure 7.2 Tasks dependency on a grid of 3×3 blocks 97
Figure 7.3 Impact of the scheduling algorithms for experiments on heterogeneous platforms. Relative speedup over the worst situation on each platform. 99
Figure 7.4 Impact of the granularity on the efficiency of seismic wave modeling on GPUs+CPUs. 100
Figure 7.5 Speedup on the HPC node (in-core dataset) over multicore execution 102
Figure 7.6 Speedup for out-of-core dataset when running on the HPC node over multicore execution. 103
Figure 7.7 Comparison of Time-to-solution metrics for Tegra K1 (Jetson), Guane-1 (HPC server) and BRGM node (Desktop platform). 105
Figure 7.8 Energy measures for the earthquake modeling on three heterogeneous architectures. 106

LIST OF TABLES

Table 2.1 Configurations of multicore machines. 33

Table 3.1 Measures for the parameters of input vector. 43
Table 3.2 Correlation coefficient of cache access vs cache misses 51
Table 3.3 RMSE, SD and R-SQUARE of cache fitting 52
Table 3.4 RMSE, SD and R-SQUARE of exponential fitting 53

Table 4.1 Parameters of input vector for each algorithm. 58
Table 4.2 Optimizations set for multicore architectures 59
Table 4.3 *p-value* of one-way ANOVA for the GFLOPS variable in the naive algorithm experiments. 60
Table 4.4 *p-value* of two-way ANOVA for the seismic wave kernel. 60
Table 4.5 *p-value* of one-way and two-way ANOVA for the GFLOPS variable in Space tiling algorithm experiments. 61
Table 4.6 Total number of experiments. 62
Table 4.7 RMSE and R-square for predicted values of the 7-point Jacobi and the Seismic Wave kernels. 63
Table 4.8 Available configurations for optimization procedure. 66
Table 4.9 *p-value* of one-way ANOVA for the manycore architecture. 66
Table 4.10 *p-value* of two-way ANOVA for the seismic wave kernel. 67
Table 4.11 Number of experiments 69
Table 4.12 Statistical estimators of our prediction model 69

Table 5.1 Heterogeneous architecture configurations. 84

Table 7.1 Memory consumption, number of blocks and number of parallel task for the simulated scenarios 98
Table 7.2 Speedup on the commodity-based hardware configuration (in-core dataset) over multicore execution 101
Table 7.3 Speedup on the commodity-based configuration (out-of-core dataset) over multicore execution 102
Table 7.4 List of runtime parameters that affect the performance of task-based implementation on heterogeneous architectures 104

CONTENTS

1 INTRODUCTION....14
1.1 Research issues....15
1.1.1 Exploiting multicore architectures....15
1.1.2 Machine Learning approaches on HPC platforms....16
1.1.3 Heterogeneous architectures....16
1.1.4 Research context....17
1.2 Objectives and contributions....18
1.3 Outline....21
2 MULTICORE ARCHITECTURES AND PROGRAMMING MODELS....24
2.1 Parallelism....24
2.2 Multicore architectures....25
2.2.1 Manycore architectures....26
2.3 Programming models on HPC architectures....26
2.3.1 Message Passing Interface....27
2.3.2 Shared-Memory programming....27
2.3.3 Impact of compilers....29
2.4 Performance evaluation....31
2.4.1 Hardware performance counters....31
2.4.2 Roofline model....32
2.5 Target machines....33
2.6 Concluding remarks....34
3 NUMERICAL BACKGROUND: GEOPHYSICAL KERNELS ON MULTI-CORE PLATFORMS....35
3.1 Stencil applications....35
3.1.1 7-point Jacobi stencil....36
3.1.2 Seismic wave propagation stencil....37
3.1.3 Acoustic wave propagation stencil....40
3.2 Standard implementations of numerical stencil....41
3.2.1 Naïve....41
3.2.2 Space Tiling....41
3.3 Performance charaterization of numerical stencils....42
3.3.1 Scalability....44
3.4 Concluding remarks....53
4 MACHINE LEARNING STRATEGY FOR PERFORMANCE IMPROVE-MENT ON MULTICORE ARCHITECTURES....54
4.1 Performance improvement by Machine Learning models....54
4.2 General model for performance prediction....55
4.2.1 Architecture of geophysics prediction model....55
4.2.2 Performance prediction on multicore architectures....57
4.2.3 Performance prediction on manycore architectures....65
4.3 Concluding remarks....70
5 HETEROGENEOUS ARCHITECTURES AND PROGRAMMING MODELS.72
5.1 Streaming Multiprocessors....73
5.2 Programming models on heterogeneous architectures....74
5.2.1 OpenCL programming model....77
5.3 Evolution of NVIDIA GPU Architectures....78
5.4 Assymetric low power architectures....82
5.5 Target machines....83

5.6 Concluding remarks **84**
6 NUMERICAL IMPLEMENTATION OF GEOPHYSICS STENCILS ON HETEROGENEOUS PLATFORMS **86**
6.1 Setup and Performance Measurement **87**
6.2 Elapsed Time **88**
6.3 GPU Load and Memory Usage **91**
6.4 Concluding Remarks **92**
7 TASK-BASED APPROACH FOR PERFORMANCE IMPROVEMENT ON HETEROGENEOUS PLATFORMS **93**
7.1 Runtime systems for task-based programming **93**
7.2 StarPU runtime system **94**
7.3 Elastodynamics over runtime system **96**
7.4 Experiments **97**
7.4.1 Scheduling strategies 98
7.4.2 Size of the block 99
7.4.3 In-core dataset 100
7.4.4 Out-of-core dataset 101
7.5 Summary of runtime parameters **103**
7.6 Task-based implementation for energy efficiency **104**
7.6.1 Computing time 105
7.6.2 Energy efficiency 106
7.7 Concluding remarks **107**
8 RELATED WORK **109**
8.1 Perfomance improvement of stencil applications on multicore architectures . **109**
8.2 Advanced optimizations, low-level and auto-tuning strategies **110**
8.3 Machine Learning approaches **111**
8.4 Heterogeneous computing **113**
9 CONCLUSION AND PERSPECTIVES **116**
9.1 Contributions **117**
9.2 Future Work **118**
REFERENCES **120**

1 INTRODUCTION

The behavior of scientific applications related to High Performance Computing (HPC) depends on many factors (non-uniform memory access, vectorization, compiler optimizations, memory policies, scheduling algorithms, etc.) that may severely influence the performance. At the hardware level, the complexity of available computing nodes is increasing, this includes several levels of hierarchical memories and more heterogeneous cores. At the software level, there are currently several programming models to exploit the architecture (shared memory, message passing, parallel tasks, etc.). These evolutions lead to redesign the code of scientific applications to obtain the best performance for a specific architecture with a specific programming model (BUCHTY et al., 2012; MITTAL; VETTER, 2015).

Examples of HPC applications are the stencil-based applications (a nearest-neighbor pattern replicated in a data domain), which are used to solve many problems related to Partial Differential Equations (PDE), the heart of many problems in areas as diverse as electromagnetics, fluid dynamics or geophysics. This is particularly true in the case of three-dimensional waves propagation in complex media, it is still one of the main challenges in geophysics and the Finite-Difference Method (FDM) is a standard technique implemented to solve these equations (MOCZO; ROBERTSSON; EISNER, 2007; DATTA et al., 2008; NGUYEN et al., 2010).

Additionally, this class of modeling heavily relies on parallel architectures in order to tackle large scale geometries including a detailed description of the physics. Last decade, significant efforts have been devoted towards efficient implementation of the FDM on emerging architectures. These contributions have demonstrated their efficiency leading to robust scientific applications (DATTA et al., 2009; MICHÉA; KOMATITSCH, 2010).

Although a large literature on the optimization of stencil numerical kernels is available, predicting and optimizing its performance remains a challenge, because many input parameters are involved and affect the performance. In terms of computational efficiency, one of the main difficulties is to deal with the disadvantageous ratio between the limited pointwise computation and the intensive memory access required, leading to a memory-bound situation (DUPROS et al., 2008; DUPROS; DO; AOCHI, 2013).

1.1 Research issues

This thesis focuses on the performance optimization of stencil applications on HPC architectures. We consider classical geophysics numerical stencils that lie at the heart of earthquake modeling, and 3D underground imaging for the oil and gas industry.

1.1.1 Exploiting multicore architectures

In addition to the challenge of geophysical modeling and the mathematical problems behind seismic and acoustic wave propagation modeling, one major challenge is to leverage the various levels of parallelism involved. HPC is actually facing key challenges on both the hardware and the software sides. Machines are reaching several millions of cores and the scalability of the applications could become a bottleneck. As reported in several recent research papers (ROTEN et al., 2016; TSUBOI et al., 2016; BREUER; HEINECKE; BADER, 2016), various geophysical applications show the variability of scaling up to thousands of cores. Indeed, regardless of the numerical method involved (Finite-Difference, Finite-Elements or Spectral-Elements), such applications benefit from the limited amount of point-to-point communications between neighboring subdomains.

Traditional methods to improve the performance of computing architectures were to increase the clock frequency, add high-speed, on-chip cache, and to optimize instructions. Nowadays, companies have turned to offer parallel machines, these architectures include several processing units on smaller chips to provide several executions of instructions in the same cycle (BLAKE; DRESLINSKI; MUDGE, 2009). Numerical implementations are focused on automatic parallelization of stencil codes (SPAZIER; CHRISTGAU; SCHNOR, 2016), analysis of the stencil performance on shared memory systems by compiler optimizations (ZHU et al., 2015), and to apply computational optimizations that scale the performance over cores and vector units (GAN et al., 2014). At the shared-memory level, efficient exploitation of the growing number of computing cores available remains a challenge.

1.1.2 Machine Learning approaches on HPC platforms

Application tuning represents one methodology to improve the performance on HPC architectures. In this case, several parameters such as machine architecture, domain decomposition, compiler flags, scheduling or load balancing algorithms are considered to achieve the best performance. Unfortunately, this approach lead to the exploration of a huge set of parameters, thus limiting its interest in complex platforms. Finding the optimal value for each parameter requires to search on a large configuration set, and several heuristics or frameworks have been proposed to speed up the process of finding the best configuration in various contexts (DATTA et al., 2010; CHRISTEN; SCHENK; BURKHART, 2011; TANG et al., 2011; MIJAKOVIC; FIRBACH; GERNDT, 2016).

At this point, Machine Learning (ML) is a methodology for optimization that could be applied to find patterns on a large set of input parameters. Recently, ML algorithms have been used on HPC systems under different situations and for various workloads such as threads mapping and memory accesses (CASTRO; GÓES; MÉHAUT, 2014), I/O scheduling (BOITO et al., 2016; LI et al., 2017), or performance improvement (GANAPATHI, 2009). Building a suitable ML-based performance model for geophysics numerical kernels remains a challenge, because the accuracy of current models don't achieve the target behavior. In this sense, a model may allow us to predict, to simulate and to optimize the performance behavior on HPC architectures with a limited amount of experiments.

1.1.3 Heterogeneous architectures

The importance of Heterogeneous Computing (HC) comes from the fact that a large fraction of main Top500 and Green500 lists of supercomputers use processors with both CPUs and coprocessors (TOP500, 2017; GREEN500, 2017). The different architectures and programming models on heterogeneous architectures also present several challenges in achieving optimal performance. HC approaches have also been referred to as collaborative, hybrid, co-operative or synergistic execution, co-processing, divide and conquer approach and others (MITTAL; VETTER, 2015). In this work, we consider heterogeneous architectures where they are built with CPUs, as main processors, and accelerated by GPU devices. Early graphics processors were special-purpose accelerators suitable only for applications related to graphics, image processing, and video

coding. Current GPUs are general-purpose (also known as GPGPU) and programmable, massively parallel processors (LINDHOLM et al., 2008).

In this sense, an approach called task-based parallelism is a data-oriented programming model used at high-level on heterogeneous architectures, the main idea is to build a task dependence graph, to create a queue of tasks with data directionality and to schedule the tasks into all available processors. Task parallelism allows the creation of multiple threads of control (processes or tasks) that can synchronize and communicate in arbitrary ways (HASSEN; BAL; JACOBS, 1998). Several runtime systems have been designed for programming and running applications on heterogeneous platforms, frameworks such as *StarPU* (AUGONNET et al., 2011), *G-Charm* (VASUDEVAN; VADHIYAR; KALÉ, 2013) or *PaRSEC* (BOSILCA et al., 2013a) have a growing impact in the scientific community. Nevertheless, the performance gains expected from the use of such runtime systems come at a price. The challenge is to decouple as much as possible the algorithms and the knowledge of the underlying architecture (STOJANOVIC et al., 2012), this situation is rather challenging for heterogeneous platforms and one of the main problems is to deal with the costly memory transfers (KRAKIWSKY; TURNER; OKONIEWSKI, 2004).

1.1.4 Research context

This research is conducted in the context of joint collaborations between the *Institute of Informatics of the Federal University of Rio Grande do Sul (INF-UFRGS)* and the *French Geological Survey (BRGM), Carnot Institute*, under the *High Performance Computing for Geophysics Applications project (HPC-GA)* funded by the *FP7-PEOPLE*, grant agreement number 295217. Research has also received funding from the *EU H2020 Programme* and from *MCTI/RNP-Brazil* under the *High Performance Computing for Energy Project (HPC4E)*, grant agreement 689772, and the *Iberian-American Network for High Performance Computing (RICAP)*, partially funded by the *Ibero-American Program of Science and Technology for Development (CYTED)*, Ref. 517RT0529. This research was also accomplished in the context of the *International Laboratory in High Performance and Ambient Informatics (LICIA)*.

At UFRGS, the research has been developed in the *Parallel and Distributed Processing Group (GPPD)*. This work has been granted by *Coordination for the Improvement of Higher Education Personnel (CAPES)*, *National Council for Scientific and Technological Development (CNPq)*, *Fundação de Amparo à Pesquisa do Estado do Rio Grande do*

Sul (FAPERGS), and *Petrobras* company.

It was also supported by *Intel Corporation* under the *Modern Code Project*. For computer time, this research partly used the resources of *Colfax Research Cluster*. Some experiments presented in this thesis were carried out using the *GridUIS-2* experimental testbed, being developed under the *Universidad Industrial de Santander High Performance and Scientific Computing Centre* (SC3UIS), development action with support from *Vicerrectoría de Investigación y Extensión (VIE-UIS)* and several UIS research groups as well as other funding bodies (<http://www.sc3.uis.edu.co>).

1.2 Objectives and contributions

The main objective of this research is to increase the performance of stencil computations from geophysics models. Many work has been guided in two alternatives: improvement of architectural features or improvement of algorithms and implementations in specific programming models; the first alternative is out of the scope of this research and second alternative limits the performance for each implementation on the underlying architecture. Thus, we research into a third alternative that has been recently used: how to tune the application by finding an optimal input set from a configuration set of runtime parameters. Then, our hypothesis is: **finding the optimal parameters from a input configuration set improve the performance of stencil applications**. Considering our objective and hypothesis, the steps are:

- To define which parameters from a configuration set, at runtime level, affect the performance of stencil computations;
- On multicore architectures, to optimize the performance of stencil computations by finding the optimal input configuration set based on an ML approach; and
- On heterogeneous architectures, to exploit the computing power by searching for an optimal runtime configuration set that uses all available processing units.

First, and according to our objectives, we contribute to the analysis and characterization of stencil computations performance on both multicore and heterogeneous architectures, they are composed by CPUs and Graphics Processing Units (GPU). On heterogeneous architectures, several high-level parameters such as data size, memory capability,

scheduling algorithms and the number of processing units have been considered. This contribution was part of *HPC-GA project*, the preliminary results were published in:

- Víctor Martínez, David Michéa, Olivier Aumage, Fabrice Dupros, and Philippe Navaux. "Hybrid CPU-GPU Computing for a Finite-Difference Numerical Seismic Kernel: First Results with StarPU". In: *High Performance Computing for Geophysics Applications, Workshop (HPC-GA)*. Oral presentation. October, 2014. Grenoble, France.

On multicore architectures, we studied the influence of several input and runtime configurations with respect to classical algorithm implementations. This contribution was part of *HPC4E project*, the results have been presented in the Latin American High Performance Computing Conference (CARLA 2016) and in the Workshop de Processamento Paralelo e Distribuído (WSPPD 2016):

- Víctor Martinez, Philippe Navaux, Fabrice Dupros, Hideo Aochi, and Márcio Castro. "Stencil-based applications tuning for multi-core architectures". In: *Latin American High Performance Computing Conference (CARLA 2016)*. Oral presentation. August, 2016. Ciudad de México, México. Available in: <https://hpc4e.eu/sites/default/files/files/presentations/Paper_Victor.pdf>.

- Víctor Martinez, Philippe Navaux, Fabrice Dupros, Hideo Aochi, and Márcio Castro. "Tuning space optimization for stencil-based applications on multi-core". In: *XIV Workshop de Processamento Paralelo e Distribuído (WSPPD)*. Oral presentation. September, 2016. Porto Alegre, Brazil. Available in: <http://www.inf.ufrgs.br/gppd/wsppd/2016/>.

Second, we introduce a Machine Learning (ML) model to predict and to optimize the performance of stencil computations on multicore architectures. The key idea is to provide adaptability of the input parameters, a training execution set is used in a learning process until the model is available to reach the best performance. The final model could be integrated into auto-tuning frameworks to find the best configuration for a given stencil application. This contribution was part of *HPC4E project*, the *Modern Code Project* and the *Petrobras 2016/00133-9 project*, the results are available in the proceedings of International Conference on Computational Science (ICCS 2017) published in *Procedia Computer Science*, proceedings of the Latin American High Performance Computing Conference (CARLA 2017) published in *Communications in Computer and*

Information Science, proceedings of 18° Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS), and has been published at HPC4E: High Performance Computing for Energy Workshop along with The Eighth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY 2018):

- Víctor Martínez, Fabrice Dupros, Márcio Castro and Philippe Navaux. "Performance Improvement of Stencil Computations for Multi-core Architectures based on Machine Learning". In: *International Conference on Computational Science (ICCS)* Zürich, Switzerland: Procedia Computer Science, 2017, V. 108, pp. 305–314. DOI:10.1016/j.procs.2017.05.164. Available in: <https://www.sciencedirect.com/science/article/pii/S1877050917307408>. Qualis: A1.

- Víctor Martínez, Matheus Serpa, Fabrice Dupros, Edson L. Padoin, and Philippe Navaux. "Performance Prediction of Acoustic Wave Numerical Kernel on Intel Xeon Phi Processor". In: *In: Mocskos E., Nesmachnow S. (eds) High Performance Computing. (CARLA 2017). Communications in Computer and Information Science*, vol 796, pp. 101–110. Springer, Cham. Buenos Aires, Argentine. DOI:10.1007/978-3-319-73353-1_7. Available in: <https://link.springer.com/chapter/10.1007/978-3-319-73353-1_7>. Qualis: B4.

- Víctor Martínez, and Philippe Navaux. "Performance Prediction of Stencil Applications on Accelerator Architectures". In: *18° Escola Regional de Alto Desempenho do Estado do Rio Grande do Sul (ERAD/RS), 2018*. Available in: <http://www.inf.ufrgs.br/erad2018/anais/FPG/179903.pdf>.

- Víctor Martínez, Matheus Serpa, Philippe Navaux, Edson L. Padoin, and Jairo Panetta. "Performance Prediction of Geophysics Numerical Kernels on Accelerator Architectures". Contribution presented at *HPC4E: High Performance Computing for Energy Workshop* along with The Eighth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY 2018). Qualis: B5.

Finally, the third contribution of this work is the introduction of a task-based implementation of the elastodynamics equation for heterogeneous architectures. This contribution on heterogeneous architectures use the maximum number of available processing units and have been demonstrated a better performance than standard implementations.

This contribution was part of *HPC-GA project*, the results were published in the proceedings of 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2015):

- Víctor Martínez, David Michéa, Fabrice Dupros, Olivier Aumage, Samuel Thibault, Hideo Aochi, and Philippe Navaux. "Towards Seismic Wave Modeling on Heterogeneous Many-Core Architectures Using Task-Based Runtime System". In: *27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2015).* Florianópolis, Brazil: IEEE Computer Society, 2015, pp. 1–8. DOI: 10.1109/SBAC-PAD.2015.33. Available in: <https://ieeexplore.ieee.org/document/7379827/>. Qualis: B1.

This contribution for heterogeneous architectures has been extended to the analysis of the energy consumption. A low-power many-core heterogeneous architecture was used as an alternative to solve the seismic model with a better energy efficiency. Our solution makes a better usage of the available resources (CPU and GPU cores) with a significant reduction of the energy consumption and communication cost. This contribution was a joined work with *SC3UIS*, the results were presented in the Latin American High Performance Computing Conference (CARLA 2015) and in the Workshop de Processamento Paralelo e Distribuído (WSPPD 2015):

- Víctor Martínez, John García, Carlos Barrios, Fabrice Dupros, Hideo Aochi, and Philippe Navaux. "Task-based Programming on Low-power Nvidia Jetson TK1 Manycore Architecture: Application to Earthquake Modelling". In: *Latin American High Performance Computing Conference (CARLA 2015).* Oral presentation. August, 2015. Petrópolis, Brazil.

- John García, Víctor Martínez, Philippe Navaux, and Carlos Barrios. eGPU for Monitoring Performance and Power Consumption on Multi-GPUs. In: *XIII Workshop de Processamento Paralelo e Distribuído (WSPPD).* Oral presentation. August, 2015. Porto Alegre, Brazil. Available in: <http://inf.ufrgs.br/gppd/wsppd/2015/papers/footer/WSPPD_2015_paper_13.pdf>

1.3 Outline

We divided this document into two parts. The First Part focuses on performance improvement of stencil applications on multicore architectures, and is organized as fol-

lows:

- Chapter 2 reviews the fundamental concepts involved in this work. We present the architectural features of multicore systems and the common parallel programming models;
- Chapter 3 describes the numerical background for the geophysics numerical kernels, and we characterize the performance of classical implementations on multicore architectures;
- Chapter 4 focuses on the use of ML to predict and to optimize the performance of geophysics numerical kernels on multicore and many-core architectures.

The Second Part is focused on performance improvement of a geophysics numerical stencil on heterogeneous architectures, and corresponds with following chapters:

- Chapter 5 reviews the basic features of heterogeneous architectures and common programming models;
- Chapter 6 presents the standard implementation of a seismic wave propagation stencil on heterogeneous architectures;
- Chapter 7 details the task-based implementation and compare the performance obtained with the classical CPU or GPU only versions.

Finally, we present the related works and the conclusion of this research in following chapters:

- Chapter 8 presents the related works;
- Chapter 9 concludes this document, and presents the perspectives.

Part 1: Performance Optimization on Multicore Architectures

2 MULTICORE ARCHITECTURES AND PROGRAMMING MODELS

We start the first part of this document with this chapter. We present a background of the HPC platforms and the programming models. Traditional methods to improve the performance of computing architectures were to increase the clock frequency, add high-speed, on-chip cache. These methods worked until physical issues limited the processor manufacturing. Nowadays, companies have turned to offer parallel machines, these architectures includes on processors several processing units to provide multiple executions of instructions in the same cycle. The performance improvement is achieved by replicating the processing units, adding additional processing instructions, improving communication between the cores, and the calculations are solved simultaneously, in parallel. In this context, Moore's law (BLAKE; DRESLINSKI; MUDGE, 2009) has also driven a constant increase in parallelism and performance.

2.1 Parallelism

Parallelism is considered for several levels, in (BUCHTY et al., 2012) the authors present six levels: (1) *Instruction Level Parallelism* (ILP) provides techniques to parallel processing at runtime, (2) *Data Parallelism* (DP) exploited mainly by Single Instruction Multiple Data (SIMD) processing, (3) *Hardware-Supported Multithreading* performs a thread level parallelism (TLP) with simultaneous multithreading (SMT) or HyperThreading (HT), when the inactive resources could be used to execute instructions from another thread, (4) *Core-Level Parallelism* builds a homogeneous multicore processor with strong coupling of cores, (5) *Socket-Level Parallelism* uses various processing devices (CPUs, GPUs, FPGAs, etc.), and finally (6) *Node Level Parallelism* is provided with a set of nodes connected by specific network topologies.

Performance optimization of parallel architectures is mainly related to adding more and more processing units, and applications need to be adapted to a wide range of platforms. Most common parallel architectures are the general-purpose multicore employing a low number of heavy-weight and highly complex cores, and multilevel cache hierarchies (i.e., private L1 and L2 caches and a shared L3 cache), as the memory access time depends on the memory location relative to the processor, each core is addressed to access its own private local memory, these architectures evolved to Non-Uniform Memory Access (NUMA) machines.

2.2 Multicore architectures

Architectural features on parallel architectures can be improved in two ways: First, multiple pipelines can be added to fetch and issue more instructions in parallel, creating a superscalar processing element; second, by increasing the number of stages, thus reducing the logic per stage. These improvements are successful for in-order processing elements. On the other hand, out-of-order architecture gains as much single thread performance as possible by dynamic scheduling to keep the pipelines full. This kind of optimization is implemented on multicore machines.

Multicore architectures can be classified according to their attributes: the application domain, the power/performance ratio, the class of processing elements, the memory system, and accelerators/integrated peripherals (SHUKLA; MURTHY; CHANDE, 2015). The application domain has two classes of processing: data processing dominated and control dominated; digital signal processors (DSPs) are the example for data processing-dominated applications and control-dominated applications include devices for file compression, decompression, and network processing. Power/Performance relation is also an important goal for multicore processors, power consumption has become a concern for computers.

At the architectural level, the memory system was a rather simple component, consisting of a few levels to feed the single processor with a data and instructions private cache. In multicores, the caches are just one part of the memory system, the other components include the consistency model, cache coherence support, and the intrachip interconnect. A consistency model defines how the memory operations may be reordered when the code is executing.

Caches have increased importance in multicore processors. They give a fast local memory to work with processing elements. The amount of cache required depends on the application. Bigger caches are better for performance but show diminishing returns as caches sizes grow. The number of cache levels has been increasing as processing elements get faster and become more numerous. The L1 cache is accessed on every instruction cycle as part of the instruction pipeline and is broken into separate instruction and data caches, and it is usually rather small, fast, and private to each processing element. The L2 cache can be private for each core or shared between cores. L3 is a shared cache for all processing elements and reduced delays in multi-threaded environments.

2.2.1 Manycore architectures

In this work, we consider the Xeon Phi processor as a special case of multicore architectures. The Knights Landing (KNL) is the code name for the second-generation Intel Xeon Phi family. It is a many-core architecture that delivers massive thread parallelism, data parallelism, and memory bandwidth in a CPU form factor for high throughput workloads. It is a standard, standalone processor that can boot an off-the-shelf operating system.

The KNL brings two types of memory: multichannel DRAM (MCDRAM) and double data rate (DDR) memory. MCDRAM is a high bandwidth and low capacity (up to 16GB) memory comprising eight devices (2 GBytes each) integrated on-package and connected to the KNL die via a proprietary on-package I/O. All eight MCDRAM devices together provide an aggregate Stream triad benchmark bandwidth of more than 450 GBytes per second. KNL has six DDR4 channels running up to 2,400 MHz, with three channels on each of two memory controllers, providing an aggregate bandwidth of more than 90 GBps. Each channel can support at most one memory DIMM. The total DDR memory capacity supported is up to 384 GBytes.

KNL has three memory modes and can be configured as *Cache* mode to work as a third level cache; in *Flat* mode, both the MCDRAM memory and the DDR memory act as regular memory and are mapped into the same system address space as a distinct NUMA node (allocatable memory); and the *Hybrid* mode, the MCDRAM is partitioned such that either a half or a quarter of the MCDRAM is used as cache, and the rest is used as flat memory (SODANI et al., 2016).

2.3 Programming models on HPC architectures

According to the evolution of HPC architectures, the programming models also have been developed to exploit the processing capability. They mainly involved the following aspects: how to send data in a network of processors and how to share and process data from the main memory between several cores. Parallel programming models are usually based on three fields: message passing, shared memory, and data-parallel. The first provides a high level of controlling architecture mapping and forces the programmer to detailed partitioning and orchestration. The second has standardized programming environments such as OpenMP for targeting compiler-exploitable TLP. The third, well-known

examples are the Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL), the first standard for kernel invocation on multiple heterogeneous systems (NVIDIA, 2016; INC., 2018). CUDA and heterogeneous architectures will be discussed in Chapter 5

2.3.1 Message Passing Interface

The Message Passing Interface (MPI) is the most common programming model for parallel machines with distributed memory and has been used widely in parallel and distributed computing systems. The basic content of MPI is point-to-point communication between processes and collective communications.

The point-to-point message-passing routines form the core of the MPI standard, the basic operations being *send* and *receive*. They allow messages to be sent between pairs of processes, with message selectivity based explicitly on message tag and source process, and implicitly on communication context. MPI defines a group as an ordered set of process identifiers, each of which is assigned a numerical rank, between zero and the size of the group. The identifier is used to send/receive data between processors.

Collective communications are provided where all processes in a group are involved in a collective operation. A collective function is called, in a group synchronization, when it is necessary; although this is not mandated and some implementations may not synchronize. These collective communications allow processing activities as broadcasting or data reducing. (CLARKE; GLENDINNING; HEMPEL, 1994)

2.3.2 Shared-Memory programming

In a shared-memory system, every processor has direct access to the memory of every other processor, meaning each one can load or store any shared address. The programmer also can declare data variables as privates to the processor. With shared-memory, the processes can exchange data more quickly than by MPI by using a designated area of memory. The data can be made directly accessible to all processes without having to use the communications (DAGUM; MENON, 1998).

Because of the ability to directly access memory throughout the system (with minimum latency and no explicit address mapping), combined with fast shared-memory locks,

OpenMP is used to implement the shared-memory applications. At its most elemental level, OpenMP is a set of compiler directives and runtime library routines that extend Fortran, C and C++ languages to express shared-memory parallelism. Program execution begins as a single process. This initial process executes serially and can set up the problem in a standard sequential manner until encounter a parallel construct defined by the directive `#pragma omp parallel`. The runtime forms a team of one or more threads and creates the data environment for each team member.

Parallel looping

The shared-memory model in OpenMP makes possible to parallelize at loop level without decomposing the data structures. The construct is `#pragma omp for`. The inner iterations from loops are executed by several threads. A parallel construct by itself creates a Single Program Multiple Data (SPMD) program, each thread redundantly executes the same code on different areas from shared data. Usually, there are many more iterations in a loop than the number of threads. Thus, a scheduling policy is necessary to assign the loop iterations to the threads.

OpenMP scheduling can be defined in runtime by the `OMP_SCHEDULE` environment variable, this variable is a string formatted by two parameters: scheduling policy and chunk size. Four different loop scheduling policies can be provided to OpenMP: *Static* divide the loop into equal-sized chunks; *Dynamic* uses the internal work queue to give a chunk-sized block of loop iterations to each thread; *Guided* is similar to dynamic, but the chunk size starts off large and decreases to better handle load imbalance between iterations; and *Auto*, when the decision regarding scheduling is delegated to the compiler. The optional parameter (chunk), when specified, must be a positive integer and defines how many loop iterations will be assigned to each thread at a time (INTEL, 2014).

Parallel tasking

Tasks in OpenMP are code blocks that the compiler wraps up and makes available to be executed in parallel threads. The construct is `#pragma omp task`. Teams of threads are created and tasks can be executed in arbitrary order, one per thread. They are synchronized by the master thread using a barrier, to check when all tasks are completed.

When a thread encounters the task construct, it may execute the task immediately or delay its execution. If delayed, the task is located in a pool of tasks associated with

the current parallel region. All threads in a team will take tasks out of the pool and execute them until the pool is empty. A thread that executes a task might be different from the thread that originally encountered it. At some point, if the list of delayed tasks is too long, the runtime may stop the task generating, and switches all threads to execute already generated tasks

2.3.3 Impact of compilers

Compilers are very important in performance optimization. To handle the multicore architectures and to parallelize a program, the compiler must perform three tasks: first, it analyzes the program to determine the dependencies between instructions; second, it performs ILP optimizations which remove dependencies between instructions; third, it reorders the instructions, a process known as code scheduling. For memory accesses, it is also useful to know if two or more instructions read the same memory location (HWU et al., 1995).

Optimization in compiling is a process where the control and data flows are analyzed and may transform the order of instructions to satisfy performance requirements. There are two common compilers for multicore architectures: the GNU Compiler Collection (GCC) is an integrated distribution of compilers for many programming languages (C, C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go) (STALLMAN; COMMUNITY, 2017), and the Intel C++ Compiler (ICC) (INTEL, 2018) to compile and generate applications that can run on the Intel® 64 and IA-32 architectures (available to Linux, Windows an MacOS).

The compilers support several optimization levels to control compiling time, memory usage, speed and code scaling at runtime. But finding optimal performance is quite difficult because to expand compiler options creates a large set. For example, to evaluate the compiler optimizations (-O1, -O2, and -O3 flags) of six parallel codes (implemented in sequential, Pthreads, C++11, OpenMP, Cilk Plus, and TBB) on two different multicore architectures (Intel Xeon and AMD Opteron) used a total of 240 performance combinations (MACHADO et al., 2017).

Additionally to OpenMP, there are some approaches to exploit the multicore architectures based on compiler directives. First, *OpenCL* is an open standard that provides a common language, programming interfaces, and hardware abstraction for developing applications in HPC environments. The environment consists of a host CPU and any at-

tached accelerator device. OpenCL offers classic compilation and linking features, and it also supports runtime compilation that allows the execution of kernels on devices unavailable when applications were developed. Runtime compilation in OpenCL allows an application to be independent of instruction sets of devices.

Second, the *OpenACC* approach is also represented by a set of compiler directives. OpenACC is a nonprofit corporation founded by four companies: CAPS Enterprise, CRAY Inc., the Portland Group Inc., and NVIDIA. Their objective was to create a cross-platform API that would easily allow acceleration of applications on manycore and multicore processors using directives. OpenACC API-enabled compilers and runtimes hide concerns about the initialization of accelerators, or data and program transfer between the host and the accelerator, inside the programming model. This allows the programmer to provide additional information to the compilers, including locality of data to an accelerator and mapping of loops onto an accelerator (CANTIELLO; MARTINO; MOSCATO, 2014).

Source-to-source transformation

Recent alternatives to support compilers and model programming for multicore architectures are the source-to-source (S2S) transformers. They convert a program source written in a given language to a new version in the same language or in a different one, and are conceived mainly with one or more of the following objectives: to transform a sequential version of code into a parallel version for a target architecture, to transform a parallel source code written with a particular paradigm (i.e., OpenMP) to a different language (i.e., OpenCL), to apply source code optimization on regions of code (i.e., loop-nests) in order to take advantage of hardware features or to improve data locality.

The process of transformation of code generally is not a priori fixed, but it can be driven in several ways: by users who can annotate code regions they want to transform, by analyzing dependences on data inside loop nests, or by the size of the data involved. There are systems that produce multiple versions of the translated code with software probes that can select among them at runtime, depending on the architecture performance, these are known as auto-tuning systems. One example of this is the BOAST framework (CRONSIOE; VIDEAU; MARANGOZOVA-MARTIN, 2013)

2.4 Performance evaluation

To exploit the parallelism, HPC applications must be programmed to considerably reduce the overhead between the processors. Because of the high cost involved in hardware implementation or software simulation of HPC architectures, a performance evaluation needs to be carried out through analytic techniques. A mathematical model to analyze the performance makes it possible to study the efficiency in terms of various design parameters used as inputs to a performance model. And it is necessary to take a general approach, independent of the application (BHUYAN; YANG; AGRAWAL, 1989).

In computer systems, performance evaluation is a fundamental phase. The most direct method of performance evaluation is by the executing applications, collecting the output, and observing the system performance by analyzing the output. Another method of performance evaluation is when the architecture is not available, this method helps to predict the requirements at previous stages of application execution (KUMAR; BALAMURUGAN, 2017). Furthermore, characterizing and predicting the performance of applications at runtime on multicore architectures help to design best performing configurations and to optimize the execution of applications on new systems. To build linear models to predict the performance in different chips is useful to characterize the performance of common HPC applications (ROSALES et al., 2017).

2.4.1 Hardware performance counters

Knowing the status on multicore architectures, when they are running the applications, allows to improve the performance. In this sense, the hardware performance counters exist as a small set of registers that counts the occurrences of specific signals related to the processor functions (cache misses, cycles, branch instructions, FLOPS, etc.). Monitoring these events facilitates correlation between the structure of code and the efficiency of that code to the underlying architecture.

In this work, we use a common library to access the registers from hardware counters called the *Performance Application Programming Interface (PAPI)*. It provides two interfaces to the underlying counter hardware: a high-level interface for the acquisition of measurements, it provides the ability to start, stop and read the counters for a specified list of events, and a low-level interface that deals with hardware events in groups called EventSets. An EventSet consists of countable events that the user can count as

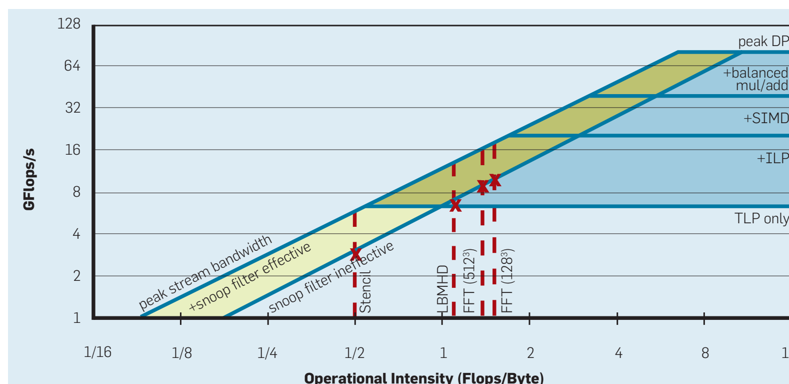
part of a group, it can reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events (MUCCI et al., 1999).

2.4.2 Roofline model

The Roofline model (WILLIAMS; WATERMAN; PATTERSON, 2009) is a performance model used for kernel computations and combines together the floating-point performance, the operational intensity, and memory performance in a 2D graph. The peak floating-point performance can be found through hardware specifications or microbenchmarks. The operational intensity means the operations per byte of DRAM traffic, defining total bytes accessed as those bytes that go to the main memory after they have been filtered by the cache hierarchy.

The graph is on a log-log scale. The y-axis is the floating-point performance. The x-axis is the operational intensity. The Roofline model provides several upper bound to performance that gives this model its name, the horizontal lines represent the parallel optimizations (TLP, ILP, SIMD, and floating-point balance) and diagonal lines represent the peak memory bandwidth and related optimizations (restructure loops for unit stride accesses, memory affinity, and software prefetching). An example of the Roofline model for an Intel Xeon multicore architecture is illustrated in Figure 2.1. The red vertical dashed lines indicate the operational intensity for different kernels (7-point stencil, Lattice-Boltzmann MagnetoHydro-Dynamics, and 3D the Fast Fourier Transform), the X marks the performance achieved for each kernel. As we can see, the different kind of applications and the parallelism level achieve different performance.

Figure 2.1: Example of the Roofline model for an Intel Xeon (Clovertown) multicore architecture.



Source: (WILLIAMS; WATERMAN; PATTERSON, 2009)

2.5 Target machines

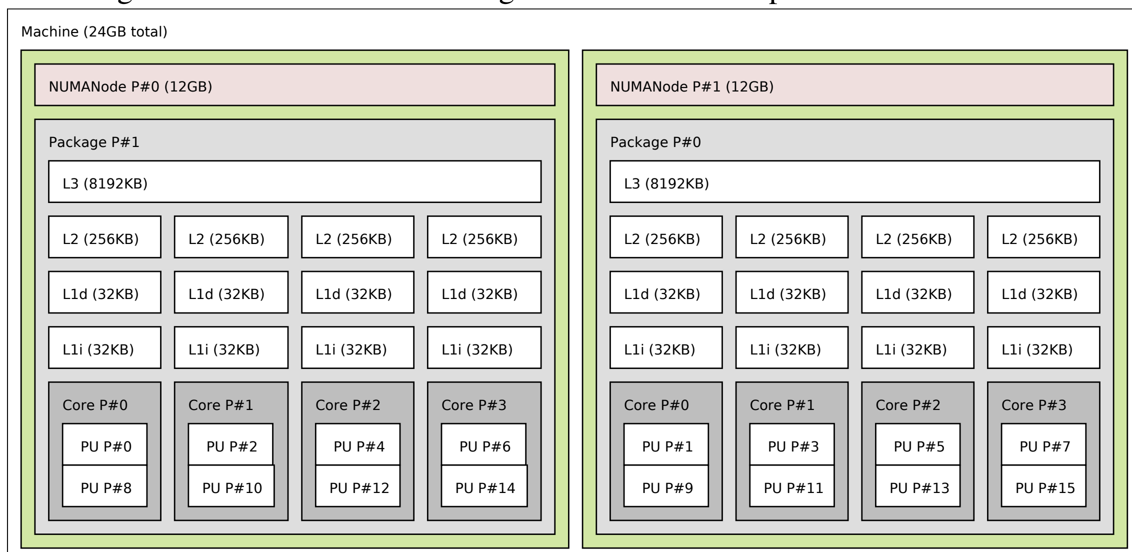
Characteristics of multicore architectures used in this research are described in Table 2.1. Although these machines are quite different in terms of the number of cores and memory size, the processor architecture is similar: a multicore chip with a memory hierarchy of several levels of cache. They are located at BRGM (France) and at Informatics Institute of UFRGS (Brazil).

Table 2.1: Configurations of multicore machines.

Hostname	***Viking (UFRGS)***	***Cryo (BRGM)***	***Server 185 (BRGM)***	***Turing (UFRGS)***	***KNL (UFRGS)***
Processor	Xeon E5530	Xeon E5-2650	Xeon E5-4650	Xeon X7550	Xeon Phi 7520
Clock (GHz)	2.40	2.00	2.70	2.00	1.40
Physical cores	4	8	8	8	68
Sockets	2	2	4	4	1
Threads	16	16	32	64	272
Cache (MB)	L3 (8)	L3 (20)	L3 (20)	L3 (18)	L2 (34)
Compiler	gcc 4.6.4	gcc 5.4.0	gcc 5.4.0	gcc 4.6.4	icc 18.0.1

Topology of *Viking* node is presented in Figure 2.2 and is a typical multicore machine. Each core runs two threads simultaneously (TLP) by the HyperThreading technology, and the memory hierarchy is easily exposed: the data and instruction L1 private caches for each core, it continues with an L2 private cache, also for each core, and the last level cache (L3) is a shared memory for all cores in the same socket.

Figure 2.2: Architeture of Viking multicore machine presented in Table 2.1.



Source: The author

2.6 Concluding remarks

In this chapter we presented the basis of common multicore architectures and how they are evolved in complex systems with many components, and how there is a constant increase in parallelism. In this context, programming models have to take into account the particular characteristics of different architectures to improve the performance. Thus, programming of hardware devices and scalability of software need to be optimized to reach the theoretically available performance.

With the rising of multicore architectures for several application domains, it is important to understand the common characteristics among all platforms (processing elements, memory hierarchy, and power/performance rates). However, as long as there is no convergence towards unified programming approaches applicable to a variety of different architectures, efforts into general solutions are a challenge. The solution for performance improvement is frequently one particular application for one detailed architecture. Moreover, although current compilers can deal with parallelism, freeing the programmer from the task of parallelizing and orchestrating is quite difficult to obtain the optimal processing, much of the responsibility is still put on the programmer.

Trending of current efforts may be focused on multicore aware algorithms, multicore enabled libraries, and multicore capable tools. The goal is not to design isolated solutions for particular configurations but to develop methodologies and concepts that apply to a wide range of problem classes and architectures. It is important for multicore systems to be able to satisfy the different computing requirements of a large fraction of users with multicore resources. To predict and to improve the performance on these systems depend on the underling architecture.

The research is necessary for exploring new models of performance optimization that are well adapted to the prerequisites of hardware and programming models. The approach of hardware aware computing is trying to find a balance between programmer capabilities from best performing implementations and the compiler based optimizations. Currently, restructuring of algorithms is needed to optimize the performance and to minimize the execution time.

3 NUMERICAL BACKGROUND: GEOPHYSICAL KERNELS ON MULTICORE PLATFORMS

In this chapter, we present the geophysics numerical models. Because of its simplicity, the Finite-Difference Method (FDM) is widely used to design the geophysics models, when discretizing Partial Differential Equations (PDE). From the numerical analysis point of view, the FDM computational procedure consists in using the neighboring points in horizontal, vertical or diagonal directions to calculate the current point.

3.1 Stencil applications

In the case of a 3D Cartesian grid, the computational procedure consists in using the neighboring points in the north-south, east-west and forward-backward directions to evaluate the current grid point. The stencil sweep can be expressed as iterative time domain (represented by the first loop controlled by n_times variable), and a triply nested parallel loop presented in Algorithm 1. The algorithm then moves to the next point applying the same stencil computation until the entire spatial grid have been traversed, and the time domain is completed. The number of points used in each direction depends on the order of the approximation and is of great importance for the overall performance.

Algorithm 1: Pseudocode for stencil algorithms

```
1:  for t = 1 to n_times do
2:    compute in parallel
3:    for i = 1 to SIZE_X_direction do
4:      for j = 1 to SIZE_Y_direction do
5:        for k = 1 to SIZE_Z_direction do
6:          compute stencil(3D tile)
7:        end for
8:      end for
9:    end for
10: end for
```

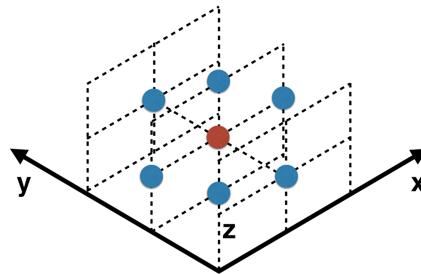
3.1.1 7-point Jacobi stencil

The 7-point Jacobi stencil is a reference example of numerical kernel used in various context in order to evaluate the impact of advanced reformulation or the impact of the underlying architecture. This numerical kernel can be described as a proxy of complex stencils like those corresponding to geophysical applications. A review can be found in (DATTA et al., 2010). This stencil model also corresponds to the standard discretization of the elliptic 3D Heat equation 3.1.

$$B_{i,j,k} = \alpha A_{i,j,k} + \beta(A_{i-1,j,k} + A_{i,j-1,k} + A_{i,j,k-1} + A_{i+1,j,k} + A_{i,j+1,k} + A_{i,j,k+1}) \quad (3.1)$$

Representation of stencil size is presented in Figure 3.1. Calculation of this numerical equation needs 7 values, one from current point plus 6 from neighbor points (one previous and one next on 3D axes).

Figure 3.1: Size of 7-point Jacobi stencil and its neighbor points.



Source: (NGUYEN et al., 2010)

A standard metric available to characterize a stencil kernel is the Arithmetic Intensity (AI) that can be defined as the ratio between the floating point operations and the memory transfers. In the case of the Seven-point Jacobi kernel, the lower-bound of the arithmetic intensity is 0.18. A synthetic pseudo-code of this kernel could be found in Algorithm 2.

Algorithm 2: Pseudo-code for the Seven-point Jacobi stencil.

for $i = 1$ to N_x **do**
  for $j = 1$ to N_y **do**
    for $k = 1$ to N_z **do**

$$X^{n+1}(i,j,k) = X^n(i,j,k) + X^n(i,j,k+1) + X^n(i,j,k-1) + X^n(i,j+1,k) + X^n(i,j-1,k) + X^n(i+1,j,k) + X^n(i-1,j,k)$$

    end for
  end for
end for

3.1.2 Seismic wave propagation stencil

Evaluation of damages occurred during strong ground motion is critical for urban planning. The seismic wave equation waves radiated from an earthquake are often simulated under the assumption of an elastic medium although the waves attenuate due to some anelasticity. If it considers a 3D isotropic elastic medium, the seismic wave equation is given by equation 3.2:

$$\rho \frac{\partial v_i}{\partial t} = \frac{\partial \sigma_{ij}}{\partial j} + F_i \tag{3.2}$$

The discretization of previous equation using a finite-difference method gives the following system of equations:

$$\begin{cases} \rho \frac{\partial}{\partial t} v_x &= \frac{\partial}{\partial x}\sigma_{xx} + \frac{\partial}{\partial y}\sigma_{xy} + \frac{\partial}{\partial z}\sigma_{xz} + f_x \\ \rho \frac{\partial}{\partial t} v_y &= \frac{\partial}{\partial x}\sigma_{yx} + \frac{\partial}{\partial y}\sigma_{yy} + \frac{\partial}{\partial z}\sigma_{yz} + f_y \\ \rho \frac{\partial}{\partial t} v_z &= \frac{\partial}{\partial x}\sigma_{zx} + \frac{\partial}{\partial y}\sigma_{zy} + \frac{\partial}{\partial z}\sigma_{zz} + f_z \end{cases} \tag{3.3}$$

Additionally, the constitutive relation in the case of an isotropic medium is presented in equation 3.4.

$$\frac{\partial \sigma_{ij}}{\partial t} = \lambda \delta_{ij} \left(\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \right) + \mu \left(\frac{\partial v_i}{\partial j} + \frac{\partial v_j}{\partial i} \right) \tag{3.4}$$

$$\begin{cases} \frac{\partial}{\partial t}\sigma_{xx} & = \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial x}v_x \\ \frac{\partial}{\partial t}\sigma_{yy} & = \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial y}v_y \\ \frac{\partial}{\partial t}\sigma_{zz} & = \lambda\left(\frac{\partial}{\partial x}v_x + \frac{\partial}{\partial y}v_y + \frac{\partial}{\partial z}v_z\right) + 2\mu\frac{\partial}{\partial z}v_z \\ \frac{\partial}{\partial t}\sigma_{xy} & = \mu\left(\frac{\partial}{\partial y}v_x + \frac{\partial}{\partial x}v_y\right) \\ \frac{\partial}{\partial t}\sigma_{xz} & = \mu\left(\frac{\partial}{\partial z}v_x + \frac{\partial}{\partial x}v_z\right) \\ \frac{\partial}{\partial t}\sigma_{yz} & = \mu\left(\frac{\partial}{\partial z}v_y + \frac{\partial}{\partial y}v_z\right) \end{cases} \tag{3.5}$$

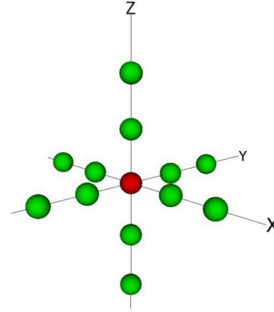
In the previous equations 3.3 and 3.5, v and σ represent the velocity and the stress field respectively and f denotes a known external source force. The medium is characterized by the elastic (Lamé) parameters λ and μ and ρ is the density. A time derivative is denoted by $\frac{\partial}{\partial t}$ and a spatial derivative with respect to the i-th direction is represented by $\frac{\partial}{\partial i}$. The Kronecker symbol δ_{ij} is equal to 1 if $i = j$ and zero otherwise. Exponents i, j, k indicate the spatial direction with $\sigma^{ijk} = \sigma(i\Delta s, j\Delta s, k\Delta s)$, Δs corresponds to the space step and Δt to the time step.

Due to its simplicity, the finite-difference method is widely used to compute the propagation of seismic waves. The numerical kernel under study relies on the classical 4-th order in space and second-order in time approximation (VIRIEUX, 1986; MOCZO; ROBERTSSON; EISNER, 2007). Considering the classical 4-th order in space and second-order in time approximation, the stencil applied for the computation of the velocity component in the x-direction is given by equation 3.6. The same numerical scheme is used to compute the stress components.

$$\begin{aligned} v_x^{(l+\frac{1}{2})jk}\left(l+\frac{1}{2}\right) &= v_x^{\left(i+\frac{1}{2}\right)jk}\left(i-\frac{1}{2}\right) + a_1 F_x^{\left(i+\frac{1}{2}\right)jk} \\ &+ a_2\left[\frac{\sigma_{xx}^{(i+1)jk}-\sigma_{xx}^{ijk}}{\Delta x} + \frac{\sigma_{xy}^{\left(i+\frac{1}{2}\right)\left(j+\frac{1}{2}\right)k}-\sigma_{xy}^{\left(i+\frac{1}{2}\right)\left(j-\frac{1}{2}\right)k}}{\Delta y} + \frac{\sigma_{xz}^{\left(i+\frac{1}{2}\right)j\left(k+\frac{1}{2}\right)}-\sigma_{xz}^{\left(i+\frac{1}{2}\right)j\left(k-\frac{1}{2}\right)}}{\Delta z}\right] \\ &- a_3\left[\frac{\sigma_{xx}^{(i+2)jk}-\sigma_{xx}^{(i-1)jk}}{\Delta x} + \frac{\sigma_{xy}^{\left(i+\frac{1}{2}\right)\left(j+\frac{3}{2}\right)k}-\sigma_{xy}^{\left(i+\frac{1}{2}\right)\left(j-\frac{3}{2}\right)k}}{\Delta y} + \frac{\sigma_{xz}^{\left(i+\frac{1}{2}\right)j\left(k+\frac{3}{2}\right)}-\sigma_{xz}^{\left(i+\frac{1}{2}\right)j\left(k-\frac{3}{2}\right)}}{\Delta z}\right] \end{aligned} \tag{3.6}$$

Figure 3.2 illustrates the size of the seismic stencil. In this case, the stencil needs 13 values, one from current point plus 12 from neighbor points (2 previous and 2 next on 3D axes).

Figure 3.2: Size of seismic stencil to calculate velocity and stress components.



Source: (MICHÉA; KOMATITSCH, 2010)

The governing equations associated with the three-dimensional modeling of seismic wave propagation in elastic media are implemented by finite-difference discretization for x86 cores and for GPU platforms corresponding to *Ondes3D* application developed by the *French Geological Survey (BRGM)*. Algorithm 3 provides an overview of the computational flowchart. The stress and velocity components are evaluated following an odd-even dependency (*i.e.*the computation of the stress field reuses the results of the update of the velocity field).

Algorithm 3: Pseudo-code of the stress component (σ_{xx}) in the seismic wave kernel.

for $i = 1$ to N_x **do**

  for $j = 1$ to N_y **do**

    for $k = 1$ to N_z **do**

$$\begin{aligned}\sigma_{xx}^{n+1}(i,j,k) = \sigma_{xx}^{n}(i,j,k) \\ + A_1[a_1(V_x^n(i+\tfrac{1}{2},j,k) - V_x^n(i-\tfrac{1}{2},j,k)) \\ + a_2(V_y^n(i,j+\tfrac{1}{2},k) - V_y^n(i,j-\tfrac{1}{2},k)) \\ + a_3(V_z^n(i,j,k+\tfrac{1}{2}) - V_z^n(i,j,k-\tfrac{1}{2}))] \\ + B_1[a_1(V_x^n(i+\tfrac{3}{2},j,k) - V_x^n(i-\tfrac{3}{2},j,k)) \\ + a_2(V_y^n(i,j+\tfrac{3}{2},k) - V_y^n(i,j-\tfrac{3}{2},k)) \\ + a_3(V_z^n(i,j,k+\tfrac{3}{2}) - V_z^n(i,j,k-\tfrac{3}{2}))]\end{aligned}$$

    end for

  end for

end for

3.1.3 Acoustic wave propagation stencil

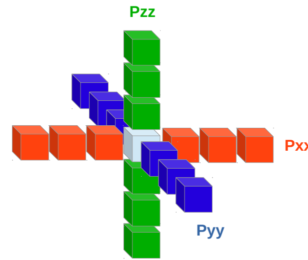
Acoustic wave propagation approximation is the current backbone for seismic imaging tools. It has been extensively applied for imaging potential oil and gas reservoirs beneath salt domes. The problem to simulate the propagation of a single wavelet over time is solved by the isotropic acoustic wave propagation (Equation 3.7), and the isotropic acoustic wave propagation with variable density (Equation 3.8) under Dirichlet boundary conditions over a finite three-dimensional rectangular domain, prescribing $p = 0$ to all boundaries, where $p(x, y, z, t)$ is the acoustic pressure, $V(x, y, z)$ is the propagation speed and $\rho(x, y, z)$ is the media density.

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p \tag{3.7}$$

$$\frac{1}{V^2}.\frac{\partial^2 p}{\partial t^2} = \nabla^2 p - \frac{\nabla \rho}{\rho}.\nabla p \tag{3.8}$$

The Laplace Operator is discretized by a 12th order finite differences approximation on each spatial dimension. The derivatives are approximated by a 2nd finite differences operator. Propagation speed depends on variable density, the acoustic pressure, and the media density. Numerical solution is detailed in (VILELA, 2017). Figure 3.3 illustrates the size of the acoustic wave propagation stencil, it needs 19 values, one from current point plus 18 from neighbor points (3 previous and 3 next on 3D axes)

Figure 3.3: Size of acoustic wave propagation stencil and its neighbor points.



Source: (VILELA, 2017)

The numerical method is solved by Algorithm 4; and *Petrobras*, the leading Brazilian oil company, provides a standalone mini-app of the numerical method. The code was written in standard C and leverage from OpenMP directives for shared-memory parallelism. But Indeed, the parallelization strategy relies on the decomposition of the three-dimensional domain based on OpenMP loop features.

Algorithm 4: Pseudo-code of the acousitc wave propagation kernel.

for $i = 1$ to N_x **do**
  for $j = 1$ to N_y **do**
    for $k = 1$ to N_z **do**

$$C_{i,j,k} = a_0 C_{i,j,k}$$
$$+ a_1(C_{i-1,j,k} + C_{i+1,j,k} + C_{i,j-1,k} + C_{i,j+1,k} + C_{i,j,k-1} + C_{i,j,k+1})$$
$$+ a_2(C_{i-2,j,k} + C_{i+2,j,k} + C_{i,j-2,k} + C_{i,j+2,k,} + C_{i,j,k-2} + C_{i,j,k+2})$$
$$+ a_3(C_{i-3,j,k} + C_{i+3,j,k} + C_{i,j-3,k} + C_{i,j+3,k} + C_{i,j,k-3} + C_{i,j,k+3})$$

    end for
  end for
end for

3.2 Standard implementations of numerical stencil

In this section, we present the implementation of stencil algorithms for parallel platforms, and we will analyze the performance on common HPC architectures. Classical implementations on multicore architecture are related to how the threads solve each point on the space and time domain.

3.2.1 Naïve

On shared-memory architectures, a popular way to extract the parallelism for such applications is to exploit the triple nested loops coming from the spatial dimensions of the problem under study. This strategy allows straightforward benefits of OpenMP directives. Additional optimizations should be considered in order to limit the impact of NUMA architectures (DUPROS et al., 2008).

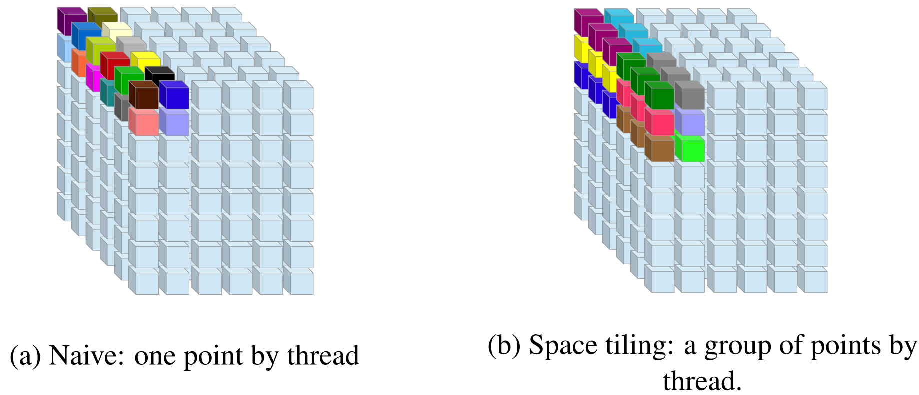
3.2.2 Space Tiling

The second algorithm uses the cache blocking technique (DUPROS; DO; AOCHI, 2013). The standard simulation algorithm typically scans an array spanning several times the size of the cache using each retrieved grid point only for a few operations. There-

fore, the cost to bring the needed data from the main memory to the fast local cache memories accounts for an important share of the total simulation time, especially for a three-dimensional problem. In this case, the main idea is to exploit the inherent data reuse available in the triple nested loop of the kernel by ensuring that data remains in cache across multiple uses, this is achieved by creating blocks to improve data locality. Dependency between blocks is exploited to implement a space-time decomposition.

The advantage of space tiling algorithm is to improve the computational intensity by keeping a relatively small amount of data in cache memory and by performing many more floating-point operations on them. Figure 3.4 shows the solution space of each algorithm. Each color represents a thread when solving one point or a set of points.

Figure 3.4: Representation of solution by thread, for naive and space tiling algorithms.



(a) Naive: one point by thread

(b) Space tiling: a group of points by thread.

Source: The Author

3.3 Performance charaterization of numerical stencils

In order to understand the performance of numerical kernels, we used two multi-core architectures to run a set of experiments for the 7-point Jacobi stencil by varying the available runtime parameters in OpenMP. The machines are NUMA platforms. Configurations of testbed (nodes Viking and Turing) have been listed in Table 2.1 from Chapter 2. A configuration runtime input vector was created and its corresponding performance output vector was obtained, for each stencil experiment. Parameters for input vector are listed in Table 3.1. Common measures to performance characterization are executing time and speedup, defined as a ratio of the time of a given program on a single core processor over the performance obtained in a multicore architecture (KRISHNAN; VEERAVALLI, 2014).

Table 3.1: Measures for the parameters of input vector.

	Total configurations	
	Viking	*Turing*
Thread counting	3	6
Problem size	3	3
Parallel looping	2	2
Scheduling	3	3
Chunk size	4	4
Algorithm	2	2
Total	**432**	**864**

The input parameters are related to code optimization and execution runtime. The first parameter is the thread counting, and it is determined by the `OMP_NUM_THREADS` variable. Second, the problem size determines the memory size and the total of computations, we used three sizes labeled as small, medium, and large. Next, related to code optimization, we used one parameter for parallel looping, we implemented the `#pragma omp parallel for` directive to parallelize the triple nested loops with `collapse` option, and the `#pragma omp task` directive, in the second `for` without `collapse`. Later, one parameter for scheduling policy used by OpenMP (Static, Guided and Dynamic) and one parameter for the chunk size configured by the `OMP_SCHEDULE` variable. Finally, we used the two standard implementations (Naive and Space Tiling) described in section 3.2. The performance vector is determined with following values:

- **Time**, which corresponds to the total execution time to solve the stencil.
- **Speedup**, it was calculated as the ratio between the execution time on the multicore machine and the single core execution time on Turing machine as the reference value.
- **GFLOPS**, it was calculated from the execution time, the 3D domain size and the stencil size.
- **Total cache misses L3**, it was obtained through `PAPI_L3_TCM` event.
- **Total cache access L3**, it was obtained through `PAPI_L3_TCA` event.

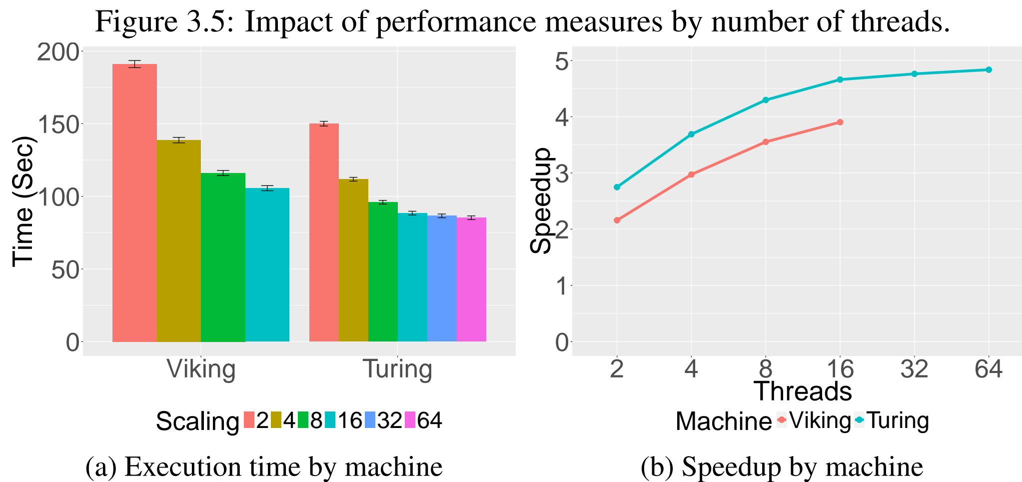
Each one of experiments was executed 15 times to compute the average, and the

Shapiro-Wilk test to time measurement was performed to confirm normality, and we obtained that all data were normally distributed.

3.3.1 Scalability

Related to the scalability of stencil applications, we measured the execution time for different values of threads on each machine. The results are presented in Figure 3.5 and show an expected behavior: when the number of threads (on the same machine) is increased the performance also increases. But, the performance improvement is not optimal. The speedup is under-linear and not correspondent with the number of threads.

Figure 3.5: Impact of performance measures by number of threads.



(a) Execution time by machine (b) Speedup by machine

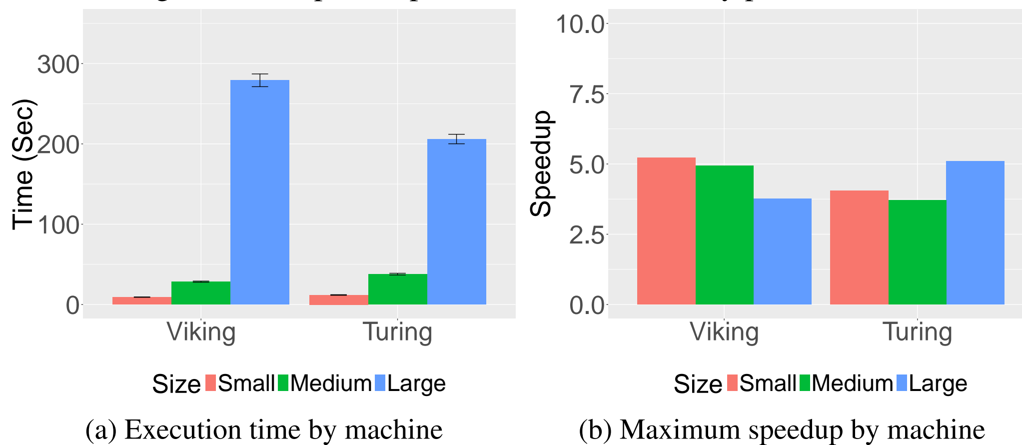
Source: The Author

Problem size

To confirm the assumption that runtime parameters affect the performance, we also analyzed three different values for the 3D problem size (it represents small, medium and large memory usage), because this parameter is associated with memory consumption. The maximum number of threads was considered on each node. Figure 3.6 illustrates the performance by 3D Cartesian grid size.

We noted when the small and the medium problems are executed on the simplest machine the speedup is better. On the contrary when the large problem is executed on the more complex machine (Turing) speedup is better for both the problem size and the multicore architecture, as it has be shown in Figure 3.6b.

Figure 3.6: Impact of performance measures by problem size.



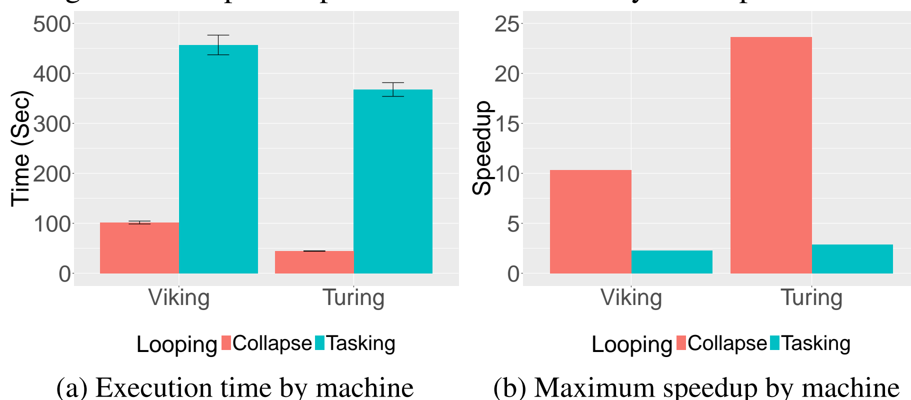
(a) Execution time by machine (b) Maximum speedup by machine

Source: The Author

Parallel loop vs tasking

The 3D stencil is calculated by three `for` loops, as explained in section 3.1. Then we compare two possible parallel implementations relevant to code optimization: i) by collapsing all `for` loops and performed the parallelization with `#pragma omp for collapse(3)`; and (ii) by using `#pragma omp task` in the second `for` to create parallel tasks and a `taskwait` directive at the end of this `for`. Figure 3.7 presents the results for each node with the maximum number of available threads in each platform. As it can be observed, the `parallel for` implementation achieved better performance than tasking. The main reason is that it creates a lot of parallel tasks and each thread has to solve a set of task and to wait and to synchronize them with each other as a result of `taskwait` clause.

Figure 3.7: Impact of performance measures by code optimization.

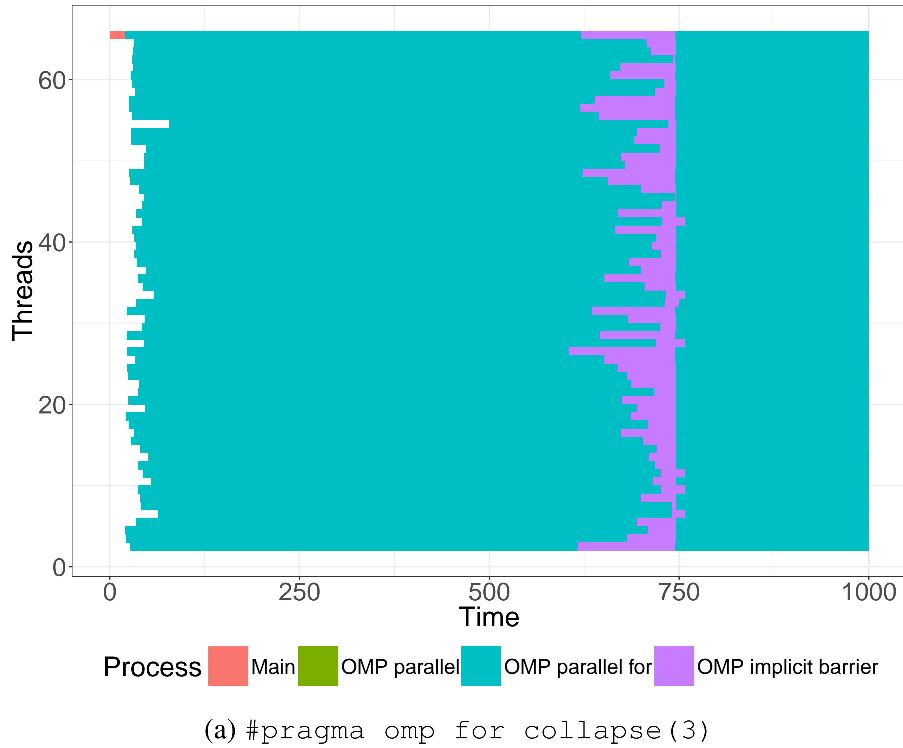


(a) Execution time by machine (b) Maximum speedup by machine

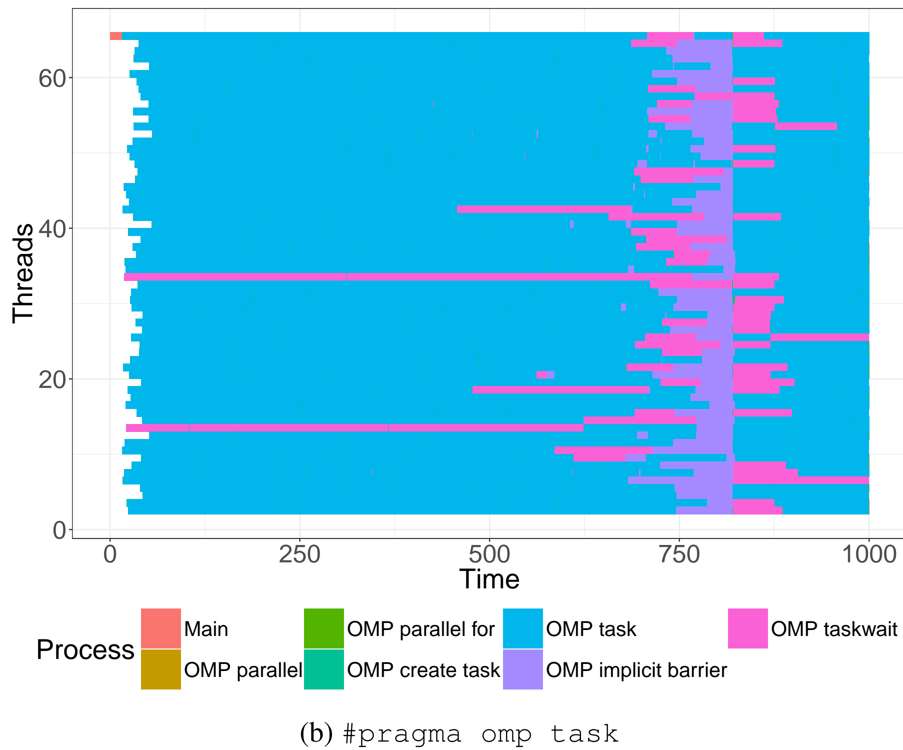
Source: The Author

To confirm this fact, the execution was traced with pajeNG[1] and it was found that `task` implementation takes more time to synchronize all the threads than `for collapse` implementation.

Figure 3.8: Traces of one-time iteration for 7-point stencil execution.



(a) `#pragma omp for collapse(3)`



(b) `#pragma omp task`

Source: The Author

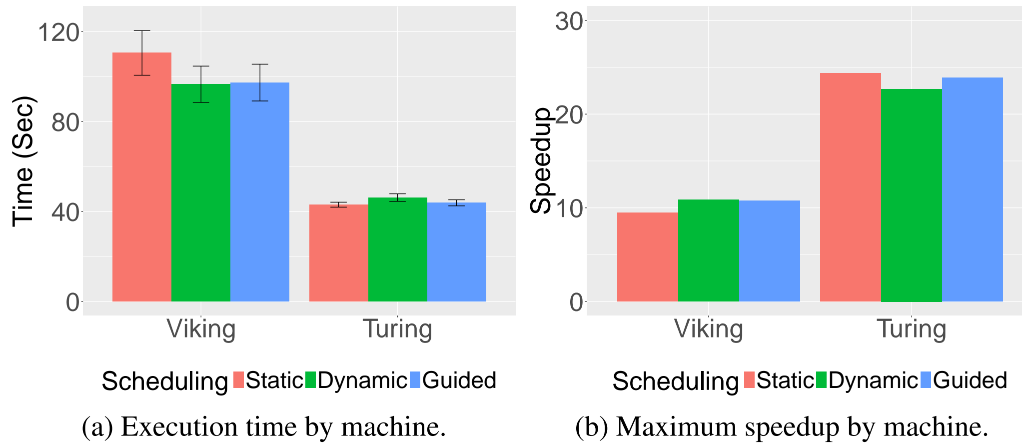
[1]https://github.com/schnorr/pajeng

Figure 3.8 shows the traces of threads (y axis) in the time domain (x axis), at first iteration represented by the n_times variable in Algorithm 1. The traces measured in a execution of `for` implementation show that `OMP parallel for` distribute all the computations between the available threads, and calculate the nested loops in a parallel way. To calculate the next iteration in the time domain, the threads have to wait in the `OMP implicit barrier` to complete the 3D space domain. In correspondent, the traces measured in `task` implementation show a different behavior. The threads solve the computations by segmented calculations assigned by `OMP task`. Some threads stay all time in a `OMP Task wait` process, or when threads complete the computations they enter in this waiting process. Later, when the time iteration finishes, the threads have to wait in the `OMP implicit barrier`. We can see that threads in `task` take more time in waiting than `for` implementation. We also noted that one-time iteration for all the 3D space is solved near to 750 seconds, in the `for` implementation; at this time, the task implementation is just starting the threads synchronization.

Scheduling policies

Loop scheduling on OpenMP is defined by two parameters of `OMP_SCHEDULE` variable: policy and chunk size (INTEL, 2016). Analyzing how these parameters influence the overall performance is important to reach optimal performance in complex architectures. As it was mentioned in section 2.3.2, the first parameter of loop scheduling is the policy. We used the available strategies in OpenMP: Dynamic, Guided and Static. Figure 3.9 presents the impact of scheduling policies on the Turing machine. In consequence, we found that policy does improve the performance of more complex platforms.

Figure 3.9: Impact of performance measures by scheduling policy.



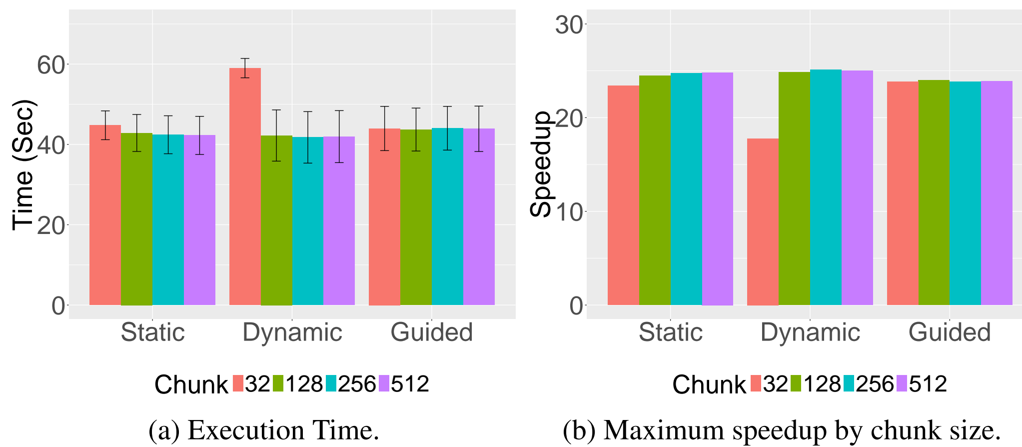
(a) Execution time by machine. (b) Maximum speedup by machine.

Source: The Author

Chunk size

The second parameter in the `OMP_SCHEDULE` variable is the chunk size, it defines the loop iterations to be assigned to each thread. Our minimum value correspond to the default value for Static scheduling, and the maximum value of chunk size correspond to the size of the 3D Cartesian grid. In Figure 3.10, we can observe two aspects: because the standard deviation of time in the experiments, we can say that chunk size is not significant. But, actually, a small chunk size has more overhead because there are more threads solving the stencil. In the same context, if we have a chunk size quite similar to problem size the stencil is solved by few threads. The performance would be affected by these conditions. In terms of scheduling policy, we can see that Dynamic is more affected if the chunk is undersized.

Figure 3.10: Impact of performance measures by scheduling and chunk size on Turing machine.



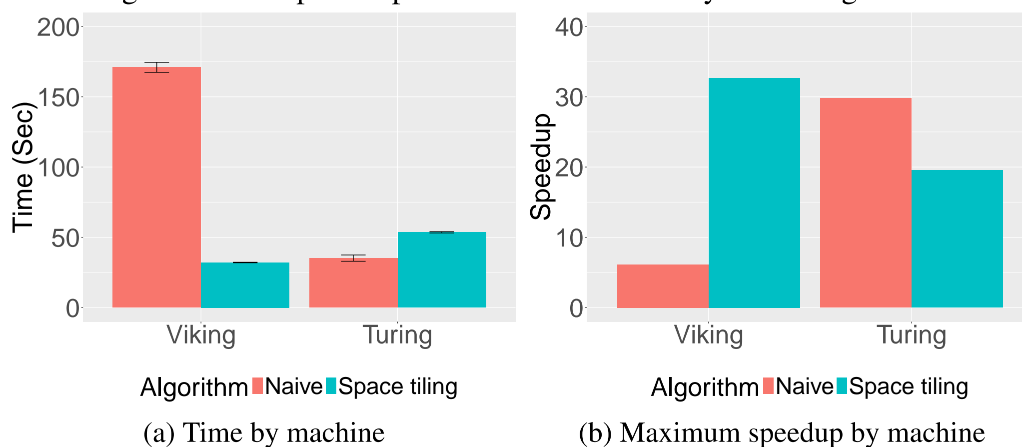
(a) Execution Time. (b) Maximum speedup by chunk size.

Source: The Author

Algorithms

Then, we analyzed the impact of algorithm implementations on the performance. Each algorithm presents a different performance for each machine. Results are presented in Figure 3.11, it shows that best performance on node Viking is achieved with Space tiling algorithm whereas on node Turing the naive algorithm achieves the best performance.

Figure 3.11: Impact of performance measures by stencil algorithm.



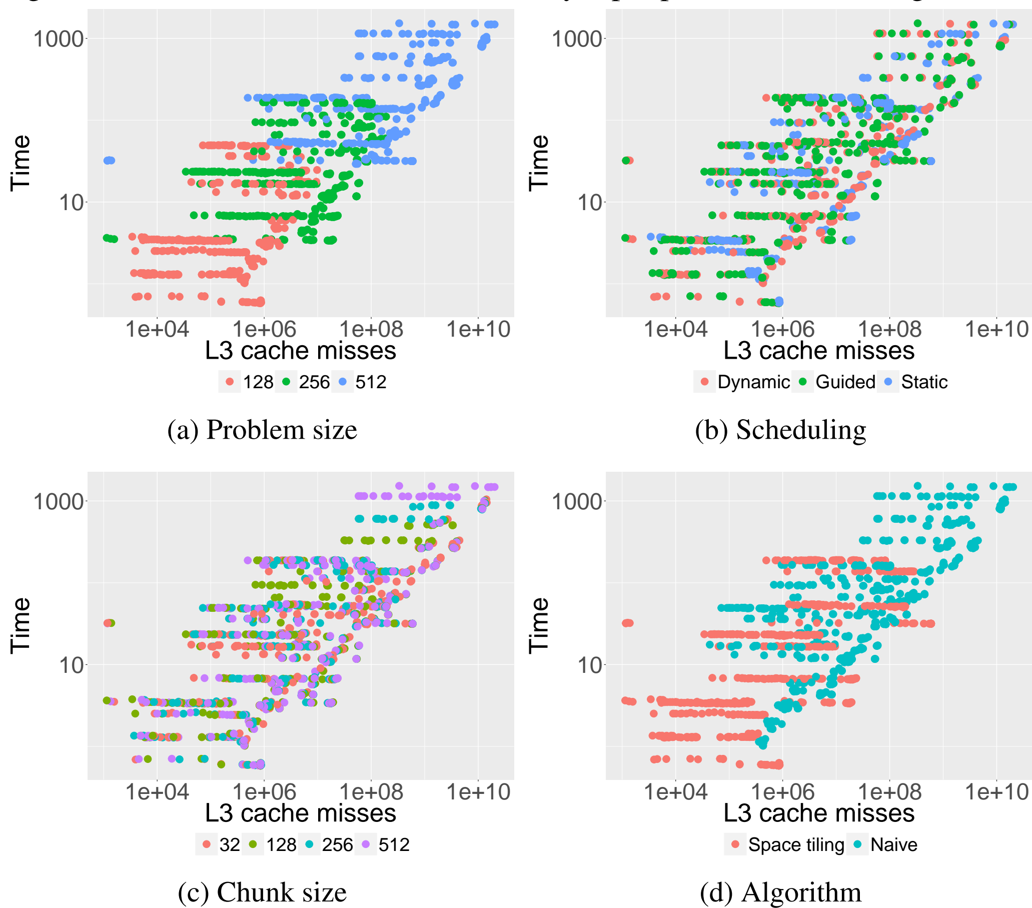
(a) Time by machine (b) Maximum speedup by machine

Source: The Author

Performance fitting and cache misses predictors

In this section, we present a previous statistical analysis and fitting of data. This would be useful to construct predictors for some parameters. Performance and cache misses are compared for all experiments. As is noticed in Figure 3.12, if we compare L3 cache misses with execution time, some input parameters tend to create separable groups in the graphical representation. Actually, this condition is clearly presented in Figures 3.12a, and 3.12d; instead, in Figures 3.12b and 3.12c the separation by their values is quite difficult.

Figure 3.12: Performance vs cache misses by input parameters in Turing machine



(a) Problem size

(b) Scheduling

(c) Chunk size

(d) Algorithm

Source: The Author

Linear fitting of L3 cache

Naive algorithm offers poor cache reuse. Thus, understanding and predicting cache behavior would help to optimize the algorithm performance. For each machine

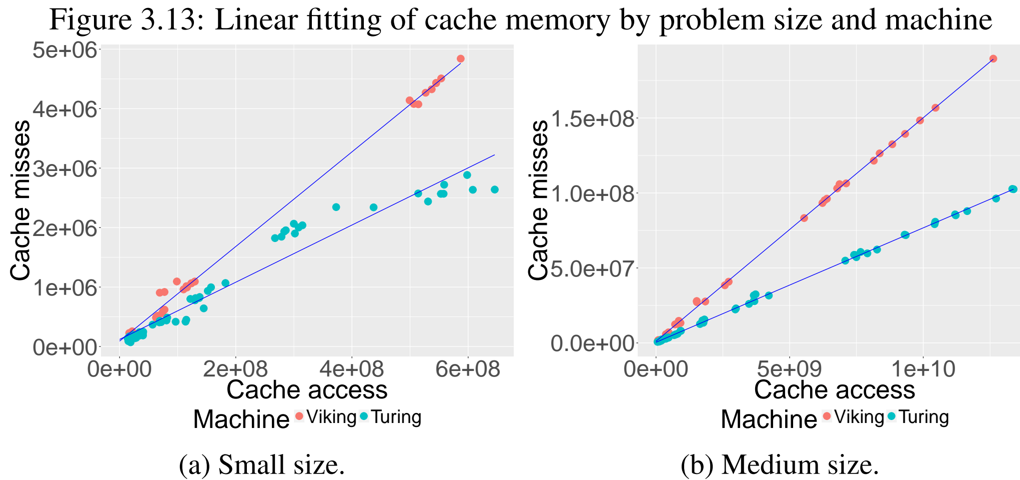
and according to the problem size we found a linear correlation between cache misses and cache access. The correlation coefficient (Table 3.2) was calculated and confirmed a linear dependency between the number of L3 cache misses and cache accesses.

Table 3.2: Correlation coefficient of cache access vs cache misses

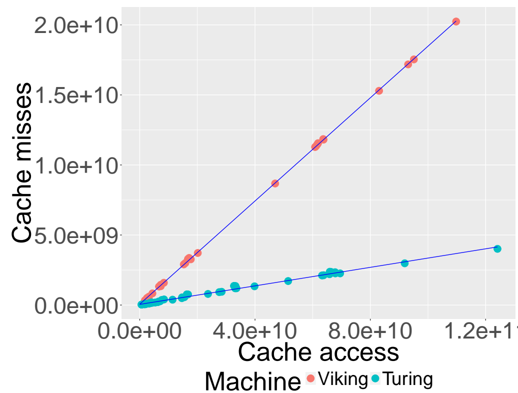
Problem size	*Small*	*Medium*	*Large*
Viking	0.9984507	0.9997246	0.9999551
Turing	0.9716369	0.9995467	0.9969517

Therefore a linear fitting by least squares was built to predict L3 cache behavior. Figure 3.13 shows this adjust.

Figure 3.13: Linear fitting of cache memory by problem size and machine



(a) Small size. (b) Medium size.



(c) Large size.

Source: The Author

We used the Root Mean Squared Error (RMSE), the Standard Deviation (SD), and the Coefficient of Determination (R-SQUARE) to describe the accuracy of our fitting model. Results are calculated in Table 3.3. If we compare RMSE and SD we find that error is quite lower than the standard deviation, thus it shows that good predictions can

be made by linear fitting. To confirm this fact, the R-SQUARE presents accuracy of prediction near to 99%; then our fitting model of cache memory helps to predict the cache access.

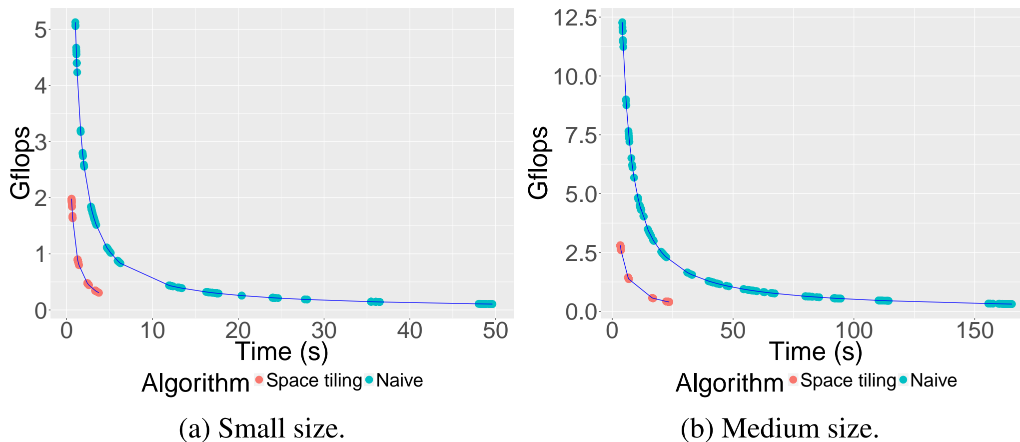
Table 3.3: RMSE, SD and R-SQUARE of cache fitting

	RMSE			*SD*			*R-SQUARE*		
Problem size	*Small*	*Medium*	*Large*	*Small*	*Medium*	*Large*	*Small*	*Medium*	*Large*
Viking	96207.09	1322882	58487888	1761918	57443329	6291905304	0.9968	0.9994	0.9999
Turing	222768	979262.6	73942621	950870.9	32830986	956618380	0.9407	0.9990	0.9938

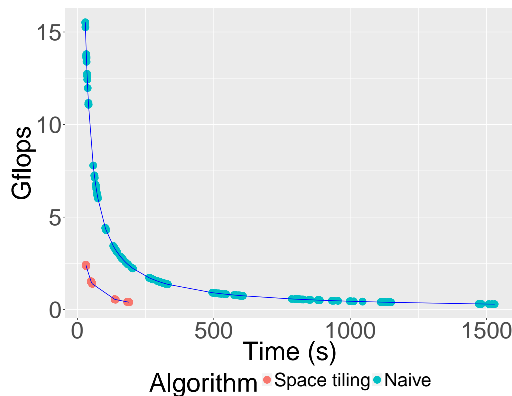
Exponential fitting of performance

As performance (Gflops) depends on the execution time, the idea in this section is to build a predictor for this parameter. We used the spline fitting because an exponential dependency was found. Figure 3.14 illustrates the fitting for the naïve and space tiling algorithm.

Figure 3.14: Exponential fitting for problem size



(a) Small size. (b) Medium size.



(c) Large size.

Source: The Author

Table 3.4 presents RMSE, SD and R-SQUARE for exponential fitting. In the same way that linear fitting, measures of statistical estimators show that a good performance predictor for Gflops and execution time was found.

Table 3.4: RMSE, SD and R-SQUARE of exponential fitting

	RMSE			*SD*			*R-SQUARE*		
Problem size	*Small*	*Medium*	*Large*	*Small*	*Medium*	*Large*	*Small*	*Medium*	*Large*
Naive	0.0004	0.0032	0.0014	1.1797	2.9209	3.5935	0.9999	0.9999	0.9999
Space tiling	0.0004	0.0002	0.0003	0.5369	0.8307	0.7338	0.9999	0.9999	0.9999

3.4 Concluding remarks

In this chapter, we presented the stencil applications, the basis of geophysics numerical kernels, and their common implementations. Shared-memory programming is the most common model for this kind of applications. We studied the influence of several input configurations for runtime (problem size, threads, looping, scheduling policy, chunk size and algorithm implementation) to the performance on multicore architectures. It was observed that two known algorithms (naive and space tiling) may present different performance in several scenarios.

For more complex architectures, input parameters like chunk size and scheduling algorithms play an important role and can contribute to achieving a peak of performance when threads do not perform intensive data communications (MARTINEZ et al., 2016). It was possible to tune the stencil execution until to reach a peak of performance, with an acceleration up to 23 and 34 times (for naive and space tiling algorithm respectively) compared to the sequential solution. Moreover, we also observed that tasks looping achieves better performance when the algorithm does not use cache intensively (space-time tiling) on architectures with few cores.

Finally, some of these parameters can be predicted by simple and common fitting. For example, the number of cache misses can be approximated by linear fitting depending on the number of cache access; whereas the performance in Gflops could be predicted by spline exponential fitting of execution time. The accuracy of statistical estimators shows a good precision and this prediction could help to make performance predictors more complex.

4 MACHINE LEARNING STRATEGY FOR PERFORMANCE IMPROVEMENT ON MULTICORE ARCHITECTURES

In this Chapter, a model to optimize the input configuration set at runtime to predict and to improve the performance of geophysics numerical kernels on multicore architectures is presented. First, we introduce how to improve the performance based on new approaches; second, we propose the general model for performance prediction; third, we present an implementation of the model applied on two multicore architectures and two different algorithm implementations; fourth, the characteristics of model for manycore architectures are described and implemented to three stencil applications. Finally, we discuss how this model could be used to improve the performance.

4.1 Performance improvement by Machine Learning models

The term Machine Learning (ML) was coined by (SAMUEL, 1959), the main idea was to program computers to learn from experience using the game of checkers through spanning the possible solutions and to select the better way by scoring the results. Applications of ML algorithms have been developed for various fields: pattern and object recognition, text categorization, time-series prediction, bioinformatics, etc.

The model of learning proposed by ML algorithms can be described by three components: 1) a generator of random vector x; 2) a supervisor that returns an output vector y for each input vector x, according to a conditional function $P(y|x)$; and 3) a learning machine capable of implementing a set of functions $f(x, \alpha)$. The problem of learning is that of choosing from a set of functions $f(x, \alpha)$ the one which predicts the supervisors' response in the best possible way (VAPNIK, 1999).

ML algorithms can be classified by the following two tasks: 1) supervised algorithms in which the input vector is identified to a predefined output, 2) unsupervised algorithms (e.g., clustering) in which the input vector is assigned to an unknown class or response. The pattern recognition problem is posed as a classification or categorization task, where the classes are either defined by the system designer (in supervised classification) or are learned based on the similarity of patterns (in unsupervised classification) (JAIN; DUIN; MAO, 2000; XU; WUNSCH, 2005).

Kernel-based algorithms are common ML methods. They are used for both classi-

fication and regression. The task of classification is to find a rule, which, based on external observations, assigns an object to one of several classes, it is made by a hyperplane created by one function. When the kernel function approximates another unknown function we obtain the solution for regression problems. The goal of learning is to find the kernel function that minimizes the error between observed values and values obtained by the ML algorithm (MULLER et al., 2001).

Recently, ML algorithms have been used on HPC systems in different situations. In (WANG; O'BOYLE, 2009) the authors presented ML-based predictors to map parallelism to multicores. They considered several different parameters such as the number of threads and scheduling policies in OpenMP programs. In (VLADUIC; CERNIVEC; SLIVNIK, 2009) the authors used ML algorithms to select the best job scheduling algorithm on heterogeneous platforms whereas in (CASTRO; GÓES; MÉHAUT, 2014) the authors proposed an ML-based approach to automatically infer a suitable thread mapping strategy for a given application. In (BOITO et al., 2016) the authors proposed an ML-based scheme to select the best I/O scheduling algorithm for different applications and input parameters.

4.2 General model for performance prediction

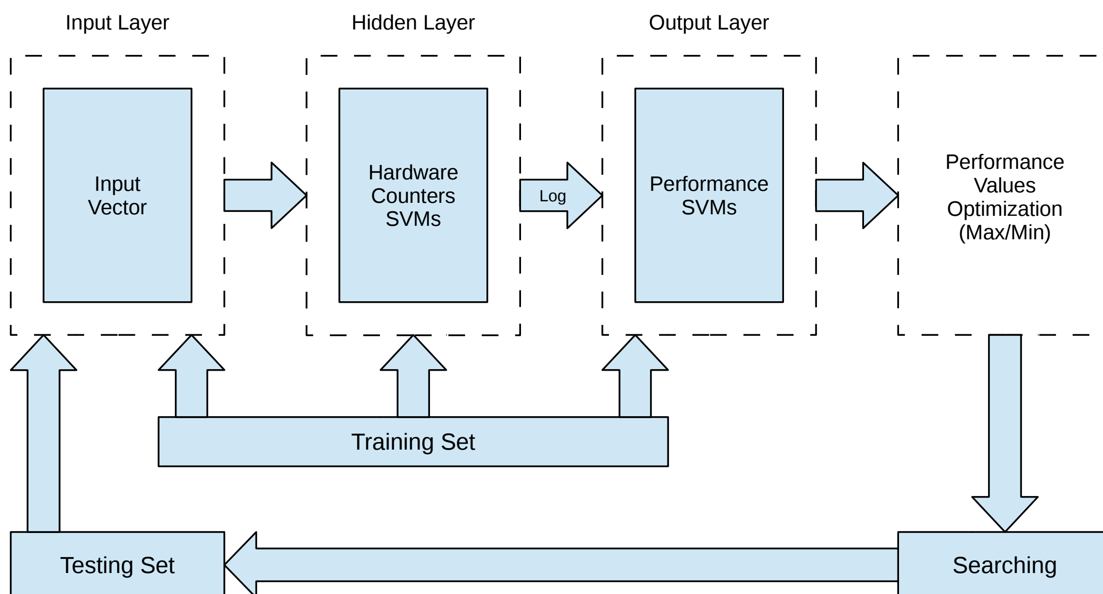
The proposed model is based on Support Vector Machine (SVM) method, which is a kernel-based and supervised approach proposed in (CORTES; VAPNIK, 1995) and it was extended to regression problems where support vectors are represented by kernel functions (DRUCKER et al., 1997). The main idea of SMV is to expand hyperplanes through the output vector. It has been employed to classify non-linear problems with non-separable training data by a decision surface, as we presented in Figure 3.12 from Section 3.3.1. Our model was implemented using *e1071* R package (MEYER et al., 2015).

4.2.1 Architecture of geophysics prediction model

The *General Model for Performance Prediction of Geophysics Stencils based on Machine Learning (Golem)* is built on top of three consecutive layers, where the output values of a layer are used as the input values of the next layer. Figure 4.1 shows the flowchart of this strategy.

The input layer contains the runtime configuration parameters from the input vector. The hidden layer contains a set of SVMs and takes the values from input vector to simulate the available hardware counters on the HPC architecture, measures for training stage were taken by PAPI library (MUCCI et al., 1999). Because hardware counters have very large values it was necessary to perform a dynamic range compression (log transformation) between the hidden and the output layers (GONZALEZ; WOODS, 2002). Finally, the output layer contains another set of SVMs and takes each value from hardware counters layer to obtain the corresponding predicted performance (GFLOPS, and execution time).

Figure 4.1: Flowchart of General Model for Performance Prediction of Geophysics Stencils based on Machine Learning (Golem).



Source: The Author

We use two separated experiment sets for training and testing stages. The training set is used for input, hidden and output layers. Since the model has been trained we use an input testing set to predict the performance and we compare the output performance between predicted and actual values to obtain the regression accuracy. Finally, we search into these predicted values the maximum of GFLOPS, the minimum of time, and its corresponding input configuration to optimize the execution time.

Statistical analysis

We conduct two statistical analysis. First, before training stage and in order to refine the results, we applied the Analysis of Variance (ANOVA) statistical model to analyze if variables for hardware counters and performance measures have different populations affected by the input values.

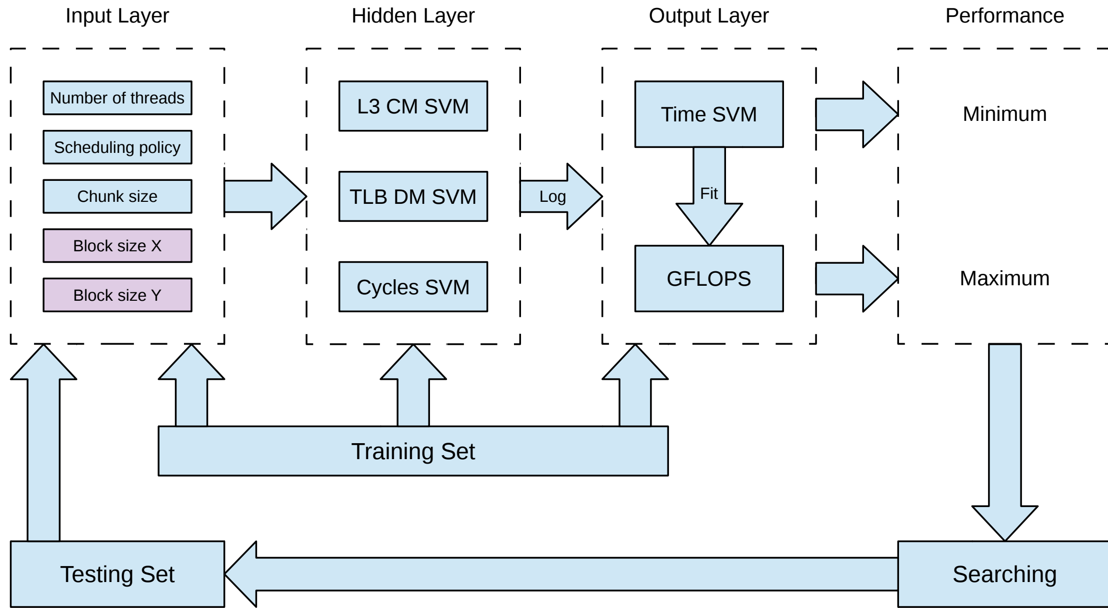
We assume the hypothesis that different populations for each variable have equal mean, it is called H_0. Thus, we compute the statistical significance (*p-value*) to determine whether the hypothesis H_0 must be rejected or not: if this value is lower than 0.05 then the hypothesis H_0 is rejected and populations have different means, that is hardware counters and performance measures are affected by input parameters with statistical significance. This analysis was divided into two classical ANOVA models: one-way ANOVA, when only one factor affects all populations; and two-way ANOVA, when two factors affect all populations.

The second statistical analysis is related to the accuracy of the regression model, we evaluate the trained model with two statistical estimators: the root mean square error (RMSE) and the coefficient of determination (R-square). The former represents the standard deviation of the differences between predicted values and actual values whereas the latter represents how close the regression approximates the actual data (R-square equal to 1 indicates a perfect fit of data regression).

4.2.2 Performance prediction on multicore architectures

Figure 4.2 presents the flowchart of *Golem* approach on multicore architectures, and we made the experiments with naive and space tiling implementations, explained in Section 3.2 from Chapter 3, for 7-point Jacobi and seismic wave propagation stencils. We note that quantity of available hardware counters depends on each HPC architecture.

Figure 4.2: Flowchart of Golem on multicore architectures.



Source: The Author

The parameters in each layer are described as:

- **Input Layer:** Values in the input vector depend on algorithm implementation, and they are defined by the runtime parameters listed in Table 4.1. In both algorithms, OpenMP runtime parameters are considered, such as the number of threads defined by the `OMP_NUM_THREADS` environment variable, that will perform the computation in parallel, loop scheduling policy (Static and Dynamic) and the chunk size defined by the `OMP_SCHEDULE` environment variable. For the space tiling algorithm, the block size in the X and Y domains are also considered.

Table 4.1: Parameters of input vector for each algorithm.

Naive	**Space tiling**
Number of threads Scheduling policy Chunk size	Number of threads Scheduling policy Chunk size Block size X Block size Y

- **Hardware Counters Layer:** The metrics considered in this layer depend on the number of available PAPI events. Because stencil computations are a memory-bound problem, we choose as the most relevant events the L3 total cache misses

(PAPI_L3_TCM), the data translation lookaside buffer misses (PAPI_TLB_DM) and the total of cycles (PAPI_TOT_CYC).

- **Output Layer:** The performance vector contains the measure of the billions of floating-point operations per second (GFLOPS) and the execution time to solve de geophysics stencil.

Testbed and configuration domain

Two multicore platforms were used to carry out the experiments, their hardware configurations are shown in Table 2.1 from Chapter 2 (Cryo and Server 185 Nodes). Based on these platforms, Table 4.2 details all the available configurations for the optimization categories. As it can be observed, a brute force approach would be unfeasible, requiring more than 3 millions in simulations for the space tiling algorithm.

Table 4.2: Optimizations set for multicore architectures

Optimization	**Parameters**	**Total configurations**	
		Cryo	**Server 185**
Number of threads	1	8	12
Scheduling policy	1	2	2
Chunk size	1	32	32
Block size X	1	64	64
Block size Y	1	64	64
Total for Naive	**3**	**512**	**768**
Total for Space Tiling	**5**	**2,097,152**	**3,145,728**

Analysis of variance and hardware counters behavior

In this analysis, GFLOPS, L3 cache misses, TLB data misses and the number of cycles were used as population variables and factors are defined by all parameters in vector input (the number of threads, the scheduling policy, and the chunk size). The results of *p-value* for the ANOVA of GFLOPS variable for naive algorithm are presented in Table 4.3. As it can be observed, all factors rejected the hypothesis for the 7-point Jacobi, since the *p-value* is lower than 0.05. In other words, the statistical significance indicates that variable GFLOPS have different populations.

Table 4.3: *p-value* of one-way ANOVA for the GFLOPS variable in the naive algorithm experiments.

	7-point Jacobi	*Seismic Wave*
Scheduling policy	2.58e-16	0.5284
Chunk size	1.37e-12	0.9985
Num. of threads	<2.2e-16	<2.2e-16
Num. of cores	<2.2e-16	<2.2e-16

For seismic stencil, the hypothesis cannot be rejected, for scheduling and chunk size variables. Then, we also used a two-way ANOVA to determine if combined variables affect populations. Table 4.4 shows the results that combine scheduling and chunk size, with executed and available threads, the *p-value* rejects the hypothesis, and variables have the statistical difference if two factors are combined. Then, analysis of variance introduces first assumption: the input factors (number of cores, number of threads, scheduling policy, and chunk size) produce statistical significance into selected GFLOPS variable in the naive algorithm experiments.

Table 4.4: *p-value* of two-way ANOVA for the seismic wave kernel.

	p-value
Scheduling policy:Num. of threads	<2.2e-16
Scheduling policy:Num. of cores	0.4664
Chunk:Num. of threads	<2.2e-16
Chunk:Num. of cores	<2.2e-16

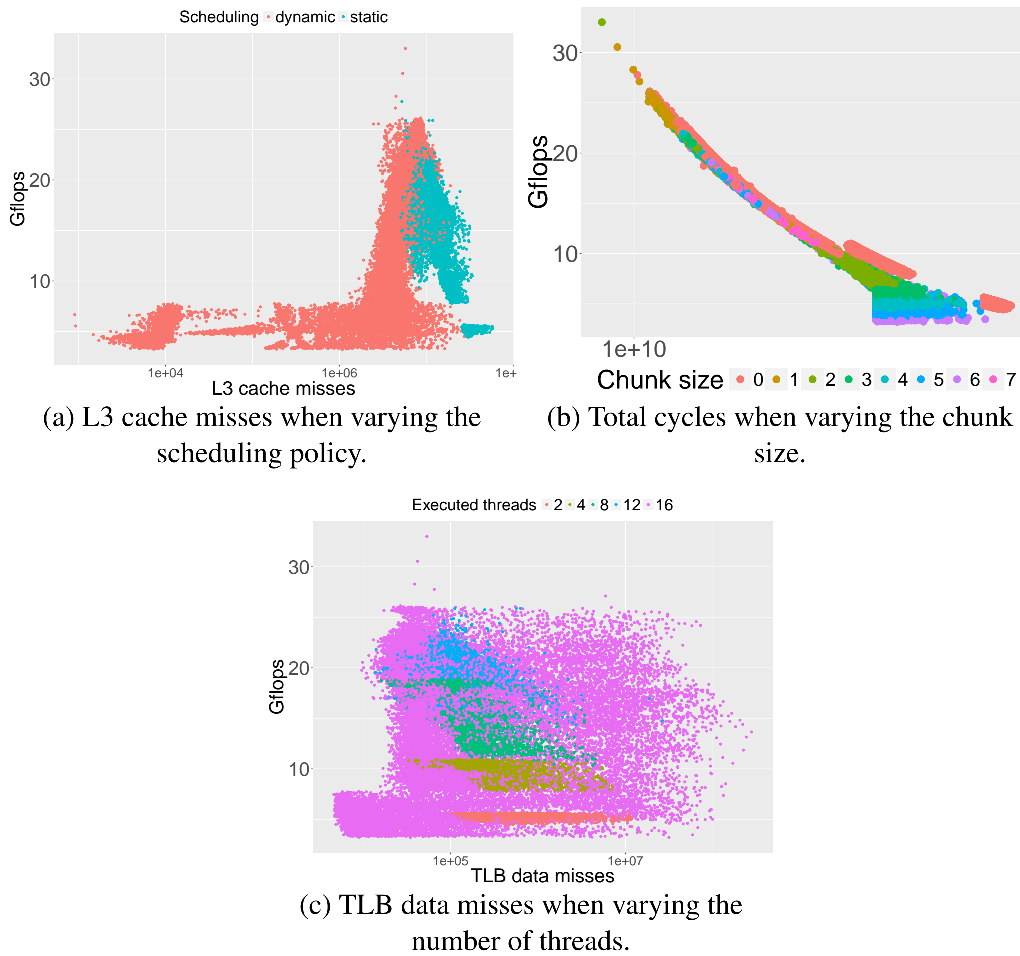
We also performed the one-way ANOVA to data from Space tiling algorithm executions, for the same variables, and a two-way ANOVA for *block size X* and *block size Y* variables. Results are presented in Table 4.5. For these analyses all the *p-value* were <0.05; then, the hypothesis is also rejected and the input factors (number of cores, number of threads, scheduling policy, chunk size, block size X and block size Y) affects the GFLOPS variable in the Space tiling algorithm experiments.

Table 4.5: *p-value* of one-way and two-way ANOVA for the GFLOPS variable in Space tiling algorithm experiments.

	7-point Jacobi	*Seismic Wave*
Scheduling policy	<2.2e-16	4.08e-05
Chunk size	<2.2e-16	<2.2e-16
Num. of threads	<2.2e-16	<2.2e-16
Num. of cores	<2.2e-16	<2.2e-16
Block size X:Block Size Y	<2.2e-16	<2.2e-16

About hardware counters behavior, Figure 4.3 illustrates how the performance of the 7-point Jacobi kernel is affected by the input variables and their relations with hardware counters, creating the non-separable training data surfaces.

Figure 4.3: Hardware counter behavior of 7-point Jacobi on Node Cryo.



(a) L3 cache misses when varying the scheduling policy.

(b) Total cycles when varying the chunk size.

(c) TLB data misses when varying the number of threads.

Source: (MARTINEZ et al., 2017)

Each point represents one experiment. For instance, Figure 4.3a shows the impact

of the scheduling policy that creates two separated areas when the GFLOPS values are analyzed with respect to the L3 cache misses. The same behavior is observed in Figure 4.3b for the chunk size when the GFLOPS values are observed with respect to the total number of cycles. The situation is rather different when the GFLOPS values are related to the amount of TLB data misses as we can see in 4.3c.

Training and validation sets

A random training set was created by selecting a subset from the experiments set with random input values (number of threads, chunk size, scheduling policy, block size X and block size Y). For each input configuration, the hardware counters (L3 cache misses, data translation lookaside buffer misses and total cycles) and performance values (GFLOPS and execution time) were measured.

A random testing set was used since all SVMs in both hidden and output layers are trained to calculate new GFLOPS and execution time values through simulation. After that, it was measured the accuracy of the model using statistical estimators. Finally, the maximum value of GFLOPS and the minimum value of the execution time are selected and matched with their input values. Simulated and real values are compared to determine if simulated best performance is the same as the real best performance. Table 4.6 presents the total number of experiments that were performed to obtain the training and validation sets.

Table 4.6: Total number of experiments.

		Naive		Space tiling	
Stencil	Set	Cryo	Server 185	Cryo	Server 185
7-point Jacobi	Total	55	48	44794	49152
	Training	44	38	2355	4054
	Testing	11	10	589	1014
Seismic Wave	Total	264	297	6849	1020
	Training	211	237	2176	371
	Testing	53	60	544	93

Accuracy of predictive modeling

As it can be observed in Table 4.7, the regression model is highly accurate. For the naive algorithm, the model presented an accuracy of up to 99.70% and 99.87% for

the GFLOPS and execution time metrics, respectively. For the space tiling algorithm, on the other hand, the model presented an accuracy of up to 98.22% and 99.71% for the GFLOPS and execution time metrics, respectively. These results are similar to the prediction of multicore architectures presented in (CRUZ; ARAYA-POLO, 2015).

Table 4.7: RMSE and R-square for predicted values of the 7-point Jacobi and the Seismic Wave kernels.

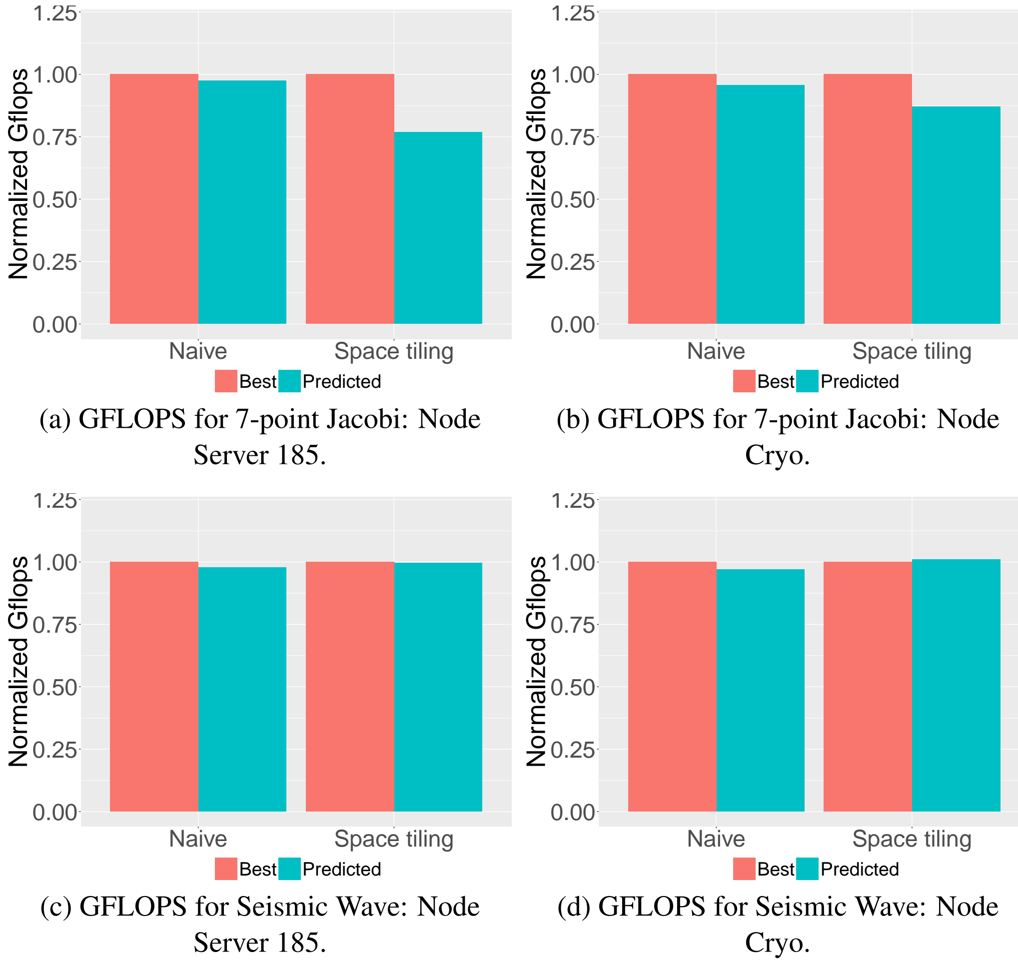
			Naive		Space tiling	
			Cryo	Server 185	Cryo	Server 185
7-point	RMSE	Gflops	0.7941	1.0179	1.4185	1.6065
		Time	0.6642	2.5089	2.2537	3.4211
	R-square	Gflops	0.9782	0.9313	0.9627	0.8540
		Time	0.9879	0.8689	0.8881	0.8049
Seismic	RMSE	Gflops	0.2273	0.6351	0.3158	0.4597
		Time	13.5391	212.282	15.6548	347.4940
	R-square	Gflops	0.9970	0.8334	0.9822	0.7313
		Time	0.9987	0.6263	0.9971	0.7494

Performance optimization

Since the goal is to obtain the best performance, the model was compared with measurements from all actual data. Figure 4.4 presents the results of this comparison. Blue bars represent the normalized output of the predicted best performance from the ML-based model (maximum GFLOPS) whereas red bars represent the normalized best performance from actual data. Perfect fit of best performance is obtained when the predicted values are the same (or very close) to actual best performance values. This means that the predicted best performance actually matches the best performance from all data. Best performance for space tiling algorithm is the human-optimized implementation described in (DUPROS et al., 2015).

For 7-point Jacobi stencil, Figures 4.4a and 4.4b compare the performance of predicted and actual data: the model achieves the best performance for the naive algorithm on these multicore architectures but predicted values for space tiling algorithm did not reach exactly the same best performance, although they are close. For the Seismic Wave kernel, Figures 4.4c and 4.4d show predicted performance is so close to the best performance, then our model achieves the best performance.

Figure 4.4: Normalized performance comparison between predicted results from the ML-algorithm and results from the best performance experimens on multicore architectures



(a) GFLOPS for 7-point Jacobi: Node Server 185.

(b) GFLOPS for 7-point Jacobi: Node Cryo.

(c) GFLOPS for Seismic Wave: Node Server 185.

(d) GFLOPS for Seismic Wave: Node Cryo.

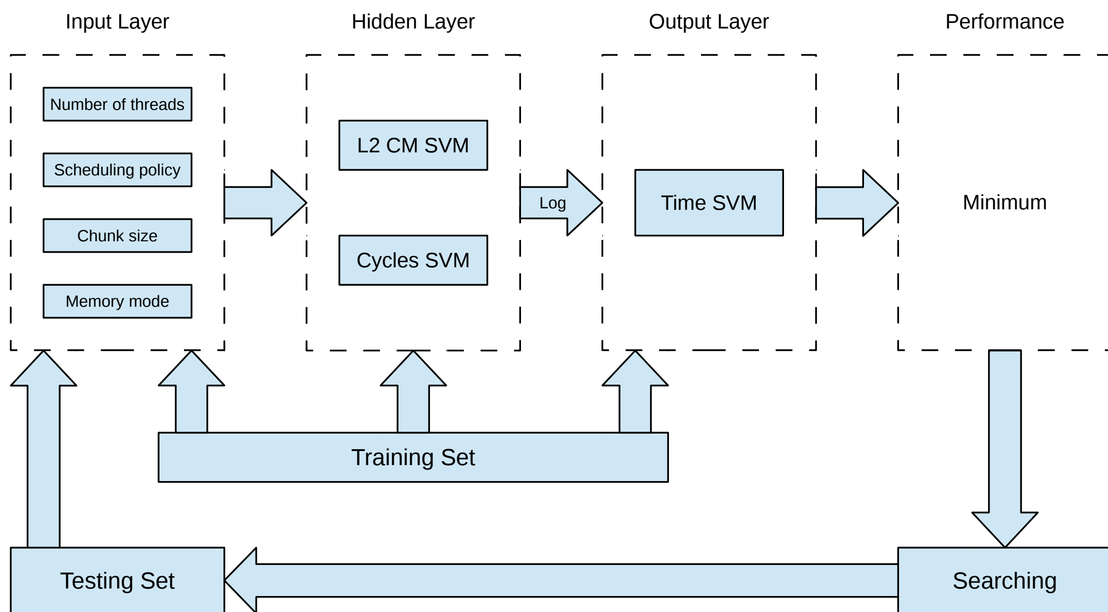
Source: (MARTINEZ et al., 2017)

Approximation of best performance for 7-point Jacobi was 97.46% and 95.61% for naive in node Server 185 and node Cryo respectively; for space tiling, it was 76.79% and 87.01% in node Server 185 and node Cryo respectively. For seismic, the approximation was 97.78% and 97.04% for naive in node Server 185 and node Cryo respectively, for space tiling in node Server 185 was 99.62% and it outperformed by 1.06% in node Cryo. Prediction for naive is easier than space tiling because the model has fewer parameters to build and it used all configurations described in Table 4.6. The speedup of the best performance for 7-point Jacobi stencil was $\times$5.57 and $\times$12.57, for naive and space tiling respectively, compared to the worst performance. For the seismic stencil, the speedup of best performance was $\times$9.33 and $\times$18.88, for naive and space tiling respectively.

4.2.3 Performance prediction on manycore architectures

In this section, we present the results of our prediction model on many-core architectures. We used the stencil naive for 7-point Jacobi, the seismic and the acoustic wave propagation with a three-dimensional grid of size 512x512x512, and 190 time iterations, as the benchmark for our experiments. Figure 4.5 presents the flowchart to this approach.

Figure 4.5: Flowchart of Golem on manycore architectures.



Source: The Author

Parameters in each layer are defined as:

- **Input Layer:**, As well as the model for multicore architectures, values for the input vector depend on OpenMP runtime parameters, as the number of threads defined by the `OMP_NUM_THREADS` environment variable, the loop scheduling policy (Static, Dynamic and Guided) and the chunk size defined by the `OMP_SCHEDULE` environment variable; additionally, for this architecture we also considered the memory mode explained in Section 2.2.1 (cache and flat).

- **Hardware Counters Layer:** We choose as the most relevant events the L2 total cache misses (`PAPI_L2_TCM`), and the total of cycles (`PAPI_TOT_CYC`).

- **Output Layer:** The performance vector contains the execution time to solve de geophysics stencil.

Testbed and configuration domain

We used one manycore platform o carry out the experiments, an Intel Xeon Phi processor shown in Table 2.1 from Chapter 2. Based on this platform, Table 4.8 details all the available configurations for the optimization categories. As it can be observed, a brute force selection would take many time on experiments.

Table 4.8: Available configurations for optimization procedure.

Optimization	Parameters	Total configurations
Number of threads	1	272
Chunk size	1	272
Scheduling policy	1	3
Memory mode	1	2
Total	**4**	**443,904**

Analysis of variance and hardware counters behavior

In this analysis, the execution time, the number of L2 cache misses, and the number of cycles were used as population variables and factors are defined by all values in vector input (the number of threads, the scheduling policy, the chunk size and memory mode). The results of *p-value* for the 7-point Jacobi, seismic and acoustic wave propagation stencils are presented in Table 4.9.

Table 4.9: *p-value* of one-way ANOVA for the manycore architecture.

	Execution time			*L2 cache misses*			*Cycles*		
	Jacobi	*Seismic*	*Acoustic*	Jacobi	Seismic	Acoustic	*Jacobi*	*Seismic*	*Acoustic*
Scheduling policy	1.26e-13	1.000	1.95e-08	<2e-16	0.992	<2e-16	<2e-16	1.000	<2e-16
Chunk size	<2e-16	0.949	<2e-16	<2e-16	0.927	<2e-16	<2e-16	0.956	<2e-16
Num. of threads	<2e-16	<2e-16	0.444	1.82e-10	<2e-16	0.000757	<2e-16	<2e-16	2.80e-07
Memory mode	0.77	0.949	<2e-16	0.132	0.919	0.424529	0.969	0.915	5.81e-05

General considerations can be observed with results from Table 4.9. First, the memory mode is a factor with no statistical significance in most of variables and stencils, only for the acoustic stencil, the memory mode produces statistical significance in execution time and cycles variables. It could be explained because the code optimizations made for this stencil (SERPA et al., 2017); the Jacobi and seismic stencils have been implemented with no code optimization for the Xeon Phi architecture. Second, the Jacobi stencil has statistical significance in all variables (execution time, L2 cache misses and cy-

cles) for the other factors. Third, variables for seismic stencil have statistical significance by threads counting. Fourth, the acoustic stencil has statistical significance for almost all variables, except for execution time by thread counting. Finally, we calculate a two-way ANOVA to determine if combined variables affect the seismic stencil performance. Table 4.10 shows the results of two-way ANOVA.

Table 4.10: *p-value* of two-way ANOVA for the seismic wave kernel.

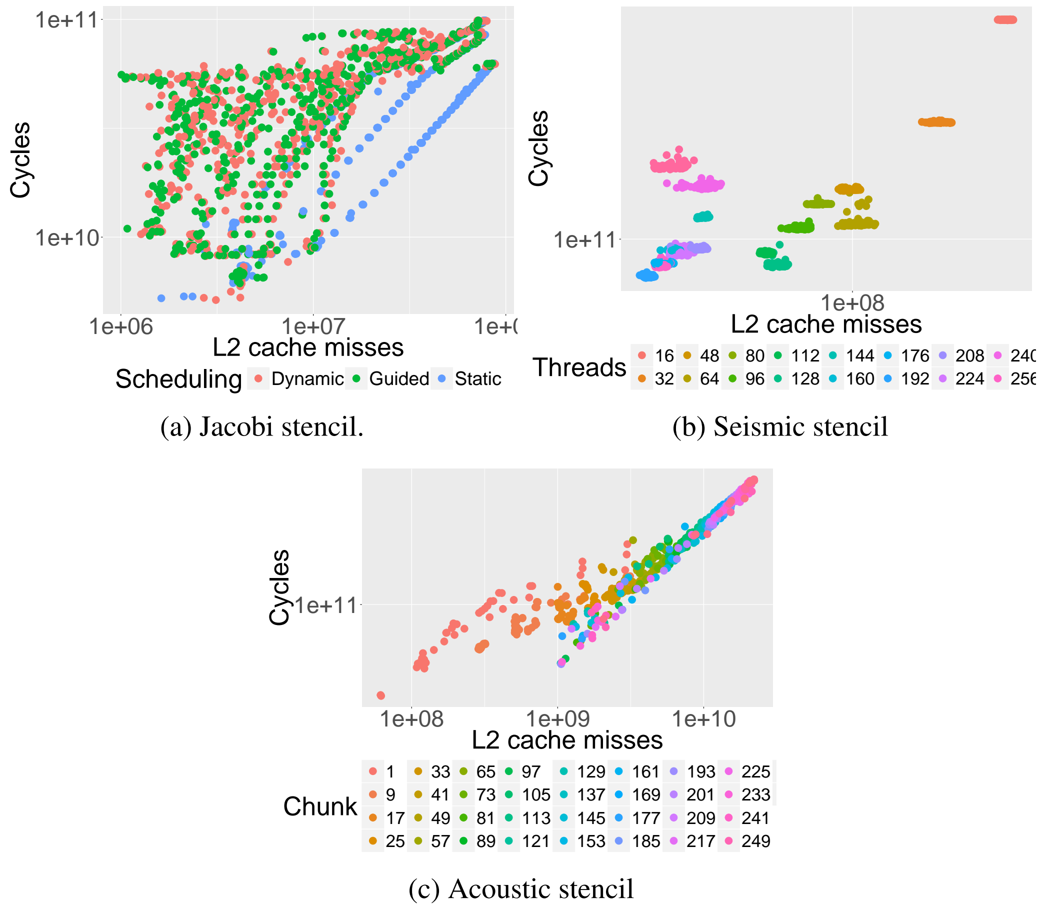
	Execution time	*L2 cache misses*	*Cycles*
Scheduling policy:Chunk	1.000	0.997	1.000
Scheduling policy:Num. of threads	<2e-16	<2e-16	<2e-16
Chunk:Num. of threads	0.992	0.956	0.988

Results in Table 4.10 show that combining scheduling and thread counting, in the seismic stencil, rejects the hypothesis, and the variables have the statistical difference if the two factors are combined. But, when we combine chunk with another factor, we still do not find statistical significance. Then, we can summarize that most of the input variables produce statistical significance into performance variables, except the chunk size.

On manycore architectures, Figure 4.6 illustrates how the hardware counters measurements are affected by the input variables. Each point represents one experiment when varying the input parameters described in Section 4.2.2, X domain represents the L2 cache misses, Y axis represents the total cycles, and the color represents the different values for the input parameter. For instance, Figure 4.6a represents the scheduling policy (green is dynamic, red is guided, and blue is static) for the Jacobi stencil, we can see how the static scheduling tends to be separated from other values. Figure 4.6b shows the number of threads used to solve the seismic stencil, we can see how each value create one easily separated area; this fact also confirms the ANOVA results of statistical significance by the number of threads. Figure 4.6c also presents how chunk size tends to create separated areas for the acoustic stencil.

We can resume this behavior as follows, changing values in input parameters affects the application performance and creates several separated areas in the graphic representation, as each color represents a different value for the input value these areas could be separated by hyperplanes from an SVM.

Figure 4.6: Hardware counters behavior on Manycore Node.



(a) Jacobi stencil.

(b) Seismic stencil

(c) Acoustic stencil

Source: The Author

Training and validation sets

We also created a training set by randomly selecting a subset from the configuration parameters presented in Table 4.8. Then, for each experiment, we measured the hardware counters (L2 cache misses, and total cycles) and performance (execution time). The random testing set was used since all SVMs in both the hidden and the output layers are trained to predict the execution time values.

After that, we measured the accuracy of the model using the statistical estimators explained in 4.2.1. Table 4.11 presents the total number of experiments that were performed to obtain the training and validation sets. It is remarkable that the total number of experiments used for testing and validation, for each stencil, is lower than 1% of total configurations described in Table 4.8.

Table 4.11: Number of experiments

	Training	Testing	**Total**
Jacobi	334	3007	**3341**
Seismic	346	3122	**3468**
Acoustic	335	3021	**3356**

Accuracy of predictive modeling

Table 4.12 presents the results for accuracy of the regression model. For the 7-point Jacobi implementation, the model presented an accuracy of up to 97.39%. For the seismic wave implementation, on the other hand, the model presented an accuracy of up to 85.62% and the acoustic wave propagation model has 93.2% of accuracy.

Table 4.12: Statistical estimators of our prediction model

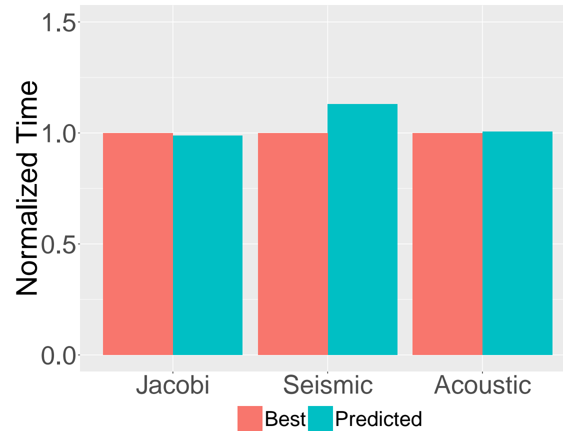
	RMSE	R-squared
Jacobi	2.9894	0.9739
Seismic	21.0183	0.8562
Acoustic	40.0748	0.9322

Performance optimization

On manycore architectures, we use the same strategy to obtain the best performance on multicore architecture. The model was compared with measurements from all actual data. Figure 4.7 present results of this comparison. Blue bars represent the normalized output of the predicted best performance from the ML-based model (minimum execution time) whereas red bars represent the normalized best performance from actual data.

Perfect fit of best performance is obtained when the predicted values are the same (or very close) to actual best performance values. For Jacobi stencil, the model achieves the best performance with accuracy of 98.88%. While for the Seismic and the Acoustic kernel, the best execution time was outperformed in 12.95% and 0.59% respectively. The speedup of best performance was $\times$49.76, $\times$35.53 and $\times$39.83 for 7-point Jacobi, seismic and acoustic stencils respectively, compared to worst performance.

Figure 4.7: Normalized performance comparison between predicted results from the ML-algorithm and results from the best performance experimens on manycore architectures.



Source: The Author

4.3 Concluding remarks

In this chapter, we proposed an ML-based model to predict the performance of stencil computations on multicore and manycore architectures when using a shared memory programming model. We showed that performance of three different stencil kernels (7-point Jacobi, seismic wave, and acoustic wave propagation) and two classical algorithm implementations (naive and space tiling) can be predicted with a high accuracy using the hardware counters measurements and the best configuration can be obtained with this methodology.

ML approaches appear as an efficient way to predict the performance of stencil computations. It allows finding the optimal input configuration to improve the performance. One major limitation on this model is related with the size of training and testing sets; for this situation, the challenge of performance prediction in real time may be obtained by research on unsupervised ML algorithms.

In this sense, complementary works would be researched towards to extend ML methodologies in order to capture complex behaviors on advanced architectures (compilers flags, vectorization, space-time blocking algorithms, or heterogeneous architectures); second, the proposed prediction model could be integrated into an auto-tuning framework to find the best performance configuration for a given stencil kernel; it may be ease by using automatic S2S transformations like a target (i.e, *BOAST* framework) (VIDEAU et al., 2018).

Part 2: Performance Improvement on Heterogeneous Architectures

5 HETEROGENEOUS ARCHITECTURES AND PROGRAMMING MODELS

The second part of this work starts with this chapter. Heterogeneous architectures are growing fast. Commodity NVIDIA's GPUs are the most popular accelerators, they are built with several stream processors with SIMD elements. Another common heterogeneous architecture is the Accelerated Processing Architecture (APU) from AMD, where CPU and GPU cores are integrated into one chip as System-on-a-Chip (SoC). The main problem on these parallel platforms is related to data movement from main memory to accelerator memory, a Dynamic RAM (DRAM). Hence, memory-aware programming is strongly recommended to achieve maximum performance.

CPUs and GPUs possess distinct architectural features. Modern multicore CPUs use up to a few tens of cores, they run at high frequency and use large-sized caches to minimize the latency of a single thread. Clearly, CPUs are suited for latency-critical applications. In contrast, GPUs use a much larger number of cores, which are in-order cores that share their control unit. GPU cores use a lower frequency and smaller-sized caches. In some cases, for applications where data transfers dominate execution time or branch divergence does not allow for the uninterrupted execution on all GPU cores, CPUs can provide better performance than GPUs. There are factors relating to architecture of Heterogeneous Computing Systems (HCS) to be taken into account: computation power of the Processing Units (PU), current load on PUs to achieve load balancing between them, memory bandwidth and CPU-GPU data transfer overhead, size of GPU and CPU memory, number of GPU threads and CPU cores, reduced performance of GPU for double-precision computations, etc.

At the application level, the performance of HCS is affected by nature of algorithms, amount of parallelism, the presence of branch divergence, subdividing the workload and selecting suitable work sizes to be allocated to all PUs and data dependencies. Workload division between CPU and GPU is used into two criteria: dynamic division, the decision about running the subtasks or program phases or code portions on a particular PU is taken at runtime; and static division, the subtasks that are executed on a particular PU are already decided before program execution; in other words, the mapping of subtasks to PUs is fixed. The decision about where a subtask should be mapped depends on which PU provides higher performance and which mapping helps in achieving load balancing. However, if subtasks differ, it may be more suitable to map a particular subtask to a particular PU; for example, highly parallel subtasks can be mapped to the GPU, while

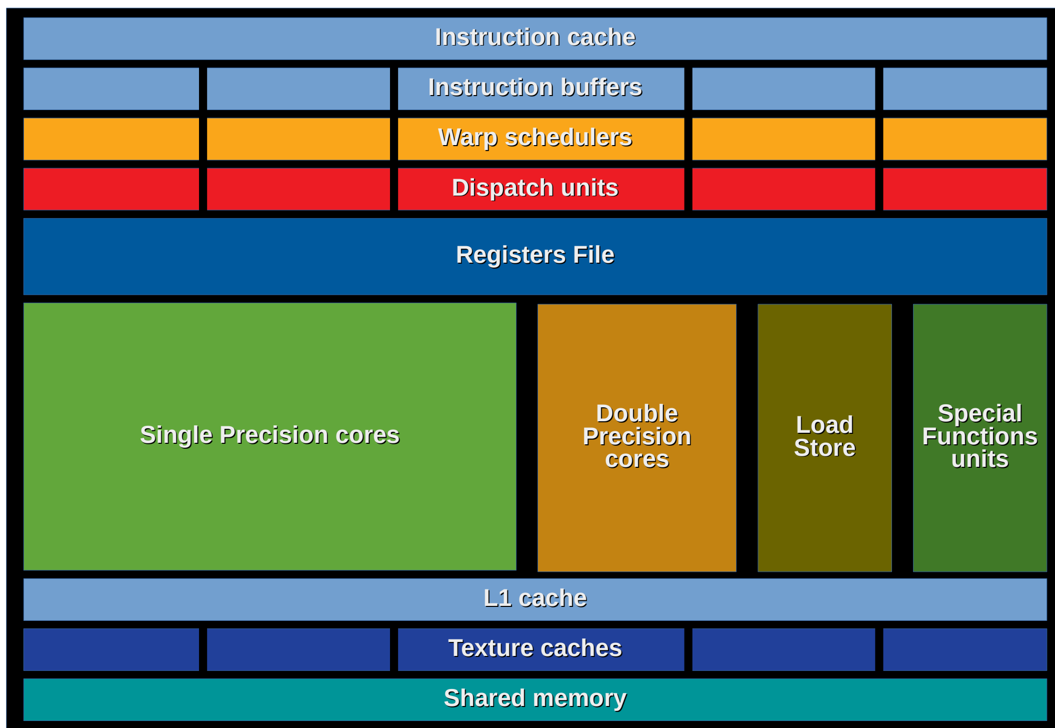
the sequential subtasks can be mapped to the CPU. Workload division techniques schedule tasks on devices based on several factors such as the contention of devices, historical performance data, number of cores, processor speed, problem size, device status (busy or free), and location of data.

5.1 Streaming Multiprocessors

The GPUs are addressed to problems that express data-parallel computations (the same program is executed on many data elements in parallel). Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches. Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. The most common example is the image processing algorithms, large sets of pixels are mapped to parallel threads; similarly, many algorithms outside this field are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology (NVIDIA, 2016). In this way, the main architectural component on a GPU is the Streaming Multiprocessor (SM).

The principal components of an SM are the cores and the several memories. There are also thousands of registers that can be partitioned among threads of execution and warp schedulers that can quickly switch contexts between threads and issue instructions to warps that are ready to execute. The cores of the most recent SM are specialized for integer and single-precision floating point operations, double-precision floating point, Load/Store Units, and Special Function Units (SFUs) for single-precision floating-point transcendental functions such as sin, cosine, reciprocal, and square root. Figure 5.1 represents the block diagram of the common SM from an NVIDIA GPU. The memory of an SM is composed by the shared memory for fast data interchange between threads, constant caches for a fast broadcast of reads from constant memory, texture caches to aggregate bandwidth from texture memory, and an L1 cache to reduce latency to local or global memory.

Figure 5.1: Block diagram of a NVIDIA GPU SM.



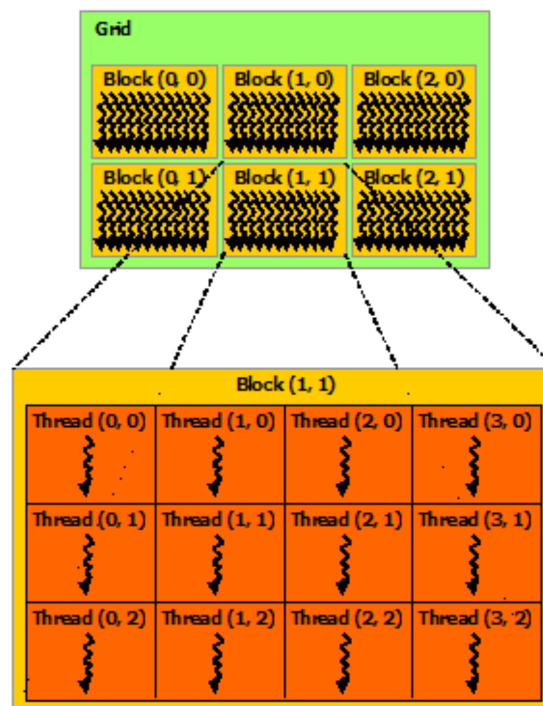
Source: The author

5.2 Programming models on heterogeneous architectures

The common programming model for NVIDIA GPUs is the CUDA programming. The CPU is known as the *host* and the GPU is known as the *device*, the data is transferred from the main RAM memory by the PCI express (PCIe) bus to the DRAM on GPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language, and at its core are three key abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. CUDA C extends C by defining C functions called kernels that are executed N times in parallel by N different CUDA threads. A kernel is defined using the `__global__` declaration and the number of CUDA threads that execute that kernel is specified using a `KernelName<<<...>>>` execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in `threadIdx` variable that is a 3-component vector. The threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a block of threads. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid as pre-

sented in Figure 5.2. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system. There is a limit to the number of threads per block since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads (NVIDIA, 2016).

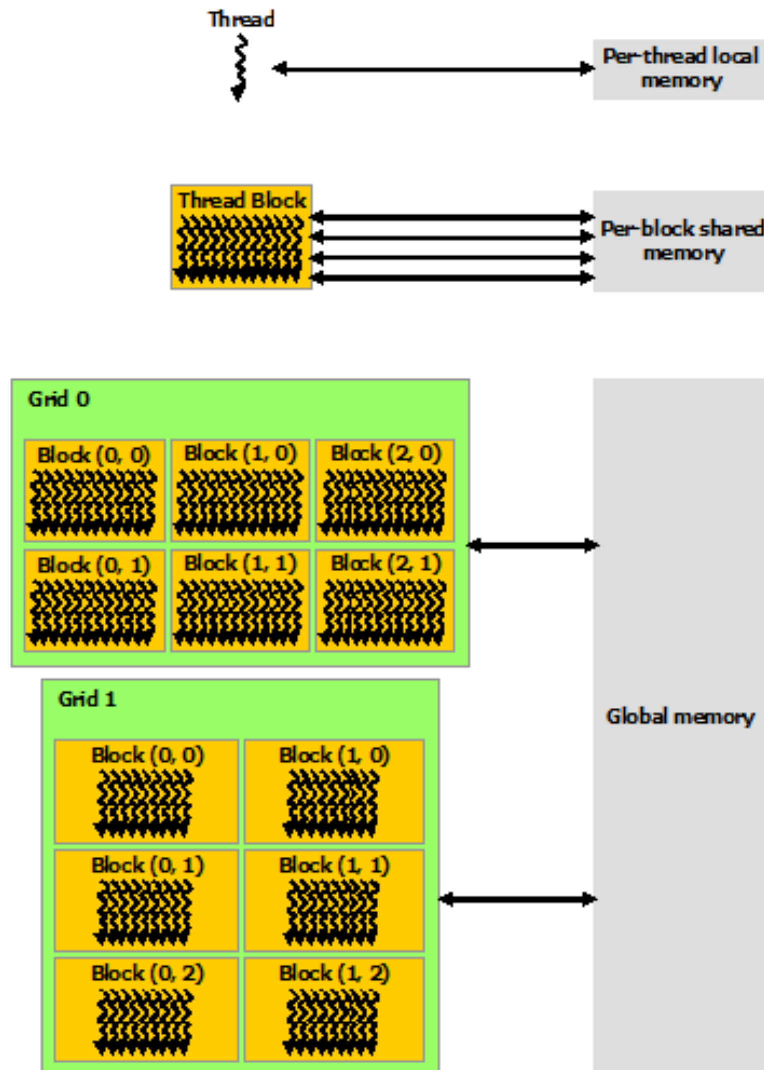
Figure 5.2: Representation of a grid with the thread blocks.



Source: (NVIDIA, 2016)

CUDA threads may access data from multiple memory spaces during their execution as presented in Figure 5.3. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages. Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application (NVIDIA, 2016).

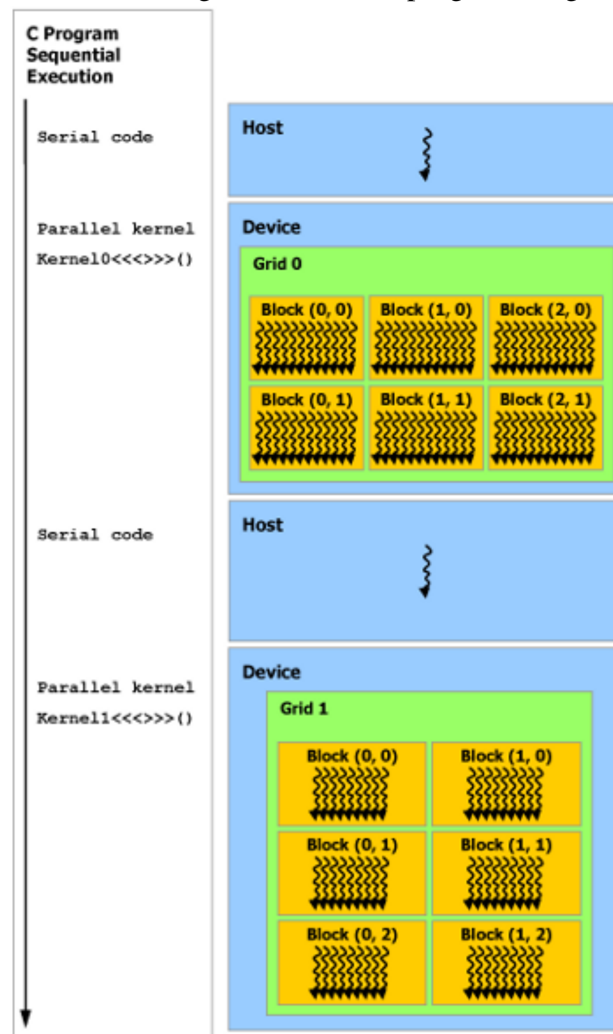
Figure 5.3: Representation of memory hierarchy for threads, blocks and grids.



Source: (NVIDIA, 2016)

The heterogeneous programming model proposed by CUDA is presented in Figure 5.4, the model assumes that the threads execute on the *device* that operates as a coprocessor to the *host* running the C program. The programming model also assumes that both the host and the device maintain their own memory spaces. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory (NVIDIA, 2016).

Figure 5.4: Heterogeneous CUDA programming model.



Source: (NVIDIA, 2016)

5.2.1 OpenCL programming model

The OpenCL is another programming model for heterogeneous architectures. It is a low-level API that runs on GPUs. Using the OpenCL API, developers can launch compute kernels written using the C programming language. OpenCL is a cross-vendor programming language used for massively parallel multi-core graphic processors, and OpenCL kernel functions define the operations carried out by each data-parallel hardware-thread (MAGNI; DUBACH; O'BOYLE, 2014). OpenCL is also a framework for writing programs for heterogeneous systems. The framework defines resources and a set of interfaces that programmers use to construct an application. The OpenCL specification defined a hierarchy of models (BALAJI, 2015):

- **Platform model:** It defines the heterogeneous system available for computations. An OpenCL platform consists of a single *host* resource, this is a general-purpose computer. Connected to the host are one or more OpenCL *devices*. An OpenCL device can be a CPU, a GPU, an FPGA, or a specialized processor. For OpenCL, the device is decomposed into one or more compute units each of which is composed of one or more *processing elements*.

- **Execution model:** It defines how a computation is launched and executed on the platform. The computation occurs on the device and this execution model is based on the *kernel parallelism* design pattern. A function is provided by the programmer to execute on the device. This is called a *kernel*. The kernel is submitted to a *command queue* for execution. A command is anything submitted by the host to the queue.

- **Memory model:** The OpenCL memory is organized into regions. *The Host memory* is available and managed by the host and provided by the native host platform. The *Global memory* is a memory region that is globally accessible to all work-items executing within a context; the *Constant memory* is defined as a region that is globally accessible on the device that can be read only during the execution of a kernel; the *Local memory* is a memory region associated with a computing unit and visible to the work-items within a work-group; and the *Private memory* region associated with a processing element and visible only within a work-item.

- **Programming model:** defines the fundamental abstractions used to map an algorithm onto source code. OpenCL supports two basic programming models. The *Data parallelism*, a single sequence of instructions is applied concurrently to each element of a data structure; and the *Task parallelism*, a task is a sequence of instructions and the data required by those instructions. In task parallelism, multiple tasks are run concurrently.

5.3 Evolution of NVIDIA GPU Architectures

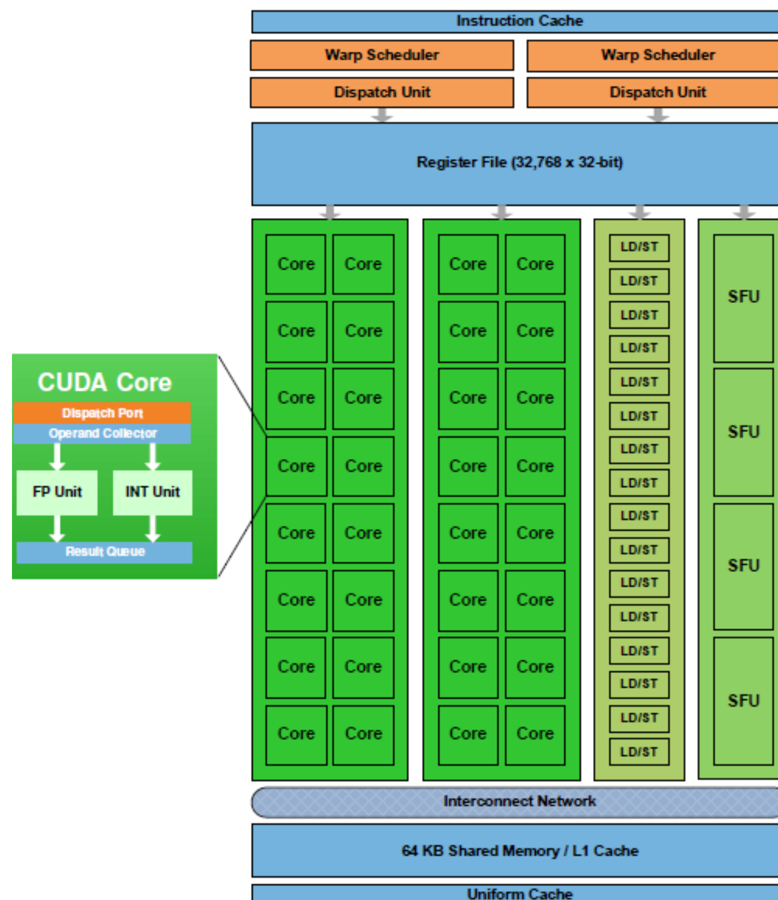
The NVIDIA's Tesla architecture, introduced in November 2006, was one the earliest improvements on GPUs, this architecture was based on a scalable processor array with 128 streaming-processor cores organized as 16 SMs in eight independent processing

units called texture/processor clusters (TPCs) (LINDHOLM et al., 2008). Consequently, NVIDIA GPU architectures have been evolved through the time, they have been released by family codenames and the principal features are the following:

Fermi

This architecture was the first with implementation of real floating point (single and double precision), the Error Correcting Codes (ECC) on main memory and caches, a single 64-bit address to unify the memory space, and atomic instructions to read from a shared location in memory, to check its value, and to write a new value back without any other processors being able to change the memory value during the execution of the atomic instruction (PATERSON, 2009; NVIDIA, 2009). Figure 5.5 illustrates the SM of Fermi architecture.

Figure 5.5: Architecture of the Fermi SM.

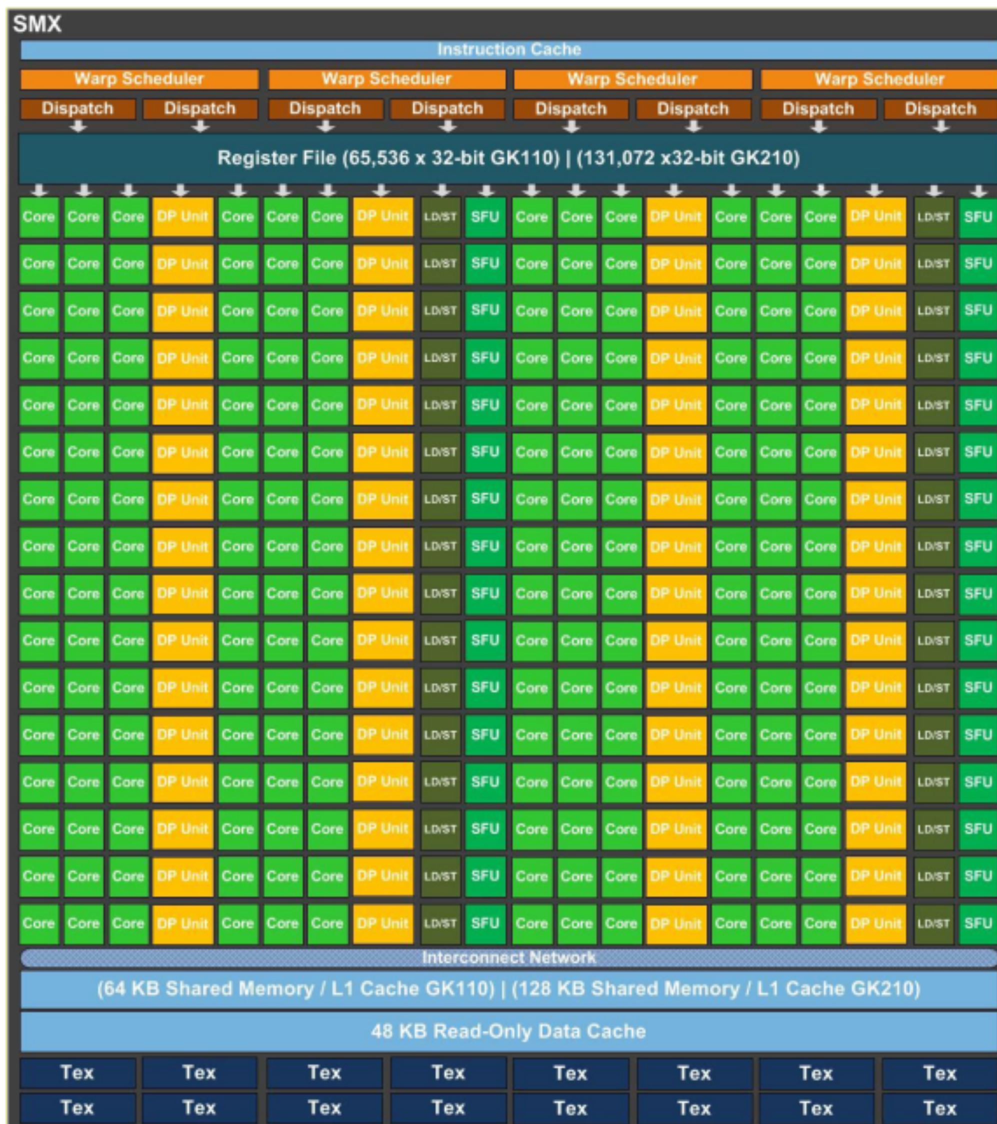


Source: (NVIDIA, 2009)

Kepler

The principal improvements on this architecture were the *Dynamic Parallelism* and the *Hyper-Q*. First, it allows the GPU device to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated paths, all without involving the CPU host. Second, it allows more connections from multiple CUDA streams, from multiple MPI processes, or from multiple threads by allowing 32 simultaneous, hardware-managed connections (NVIDIA, 2014b). Figure 5.6 illustrates the SM of Kepler architecture.

Figure 5.6: Architecture of the Kepler SM.

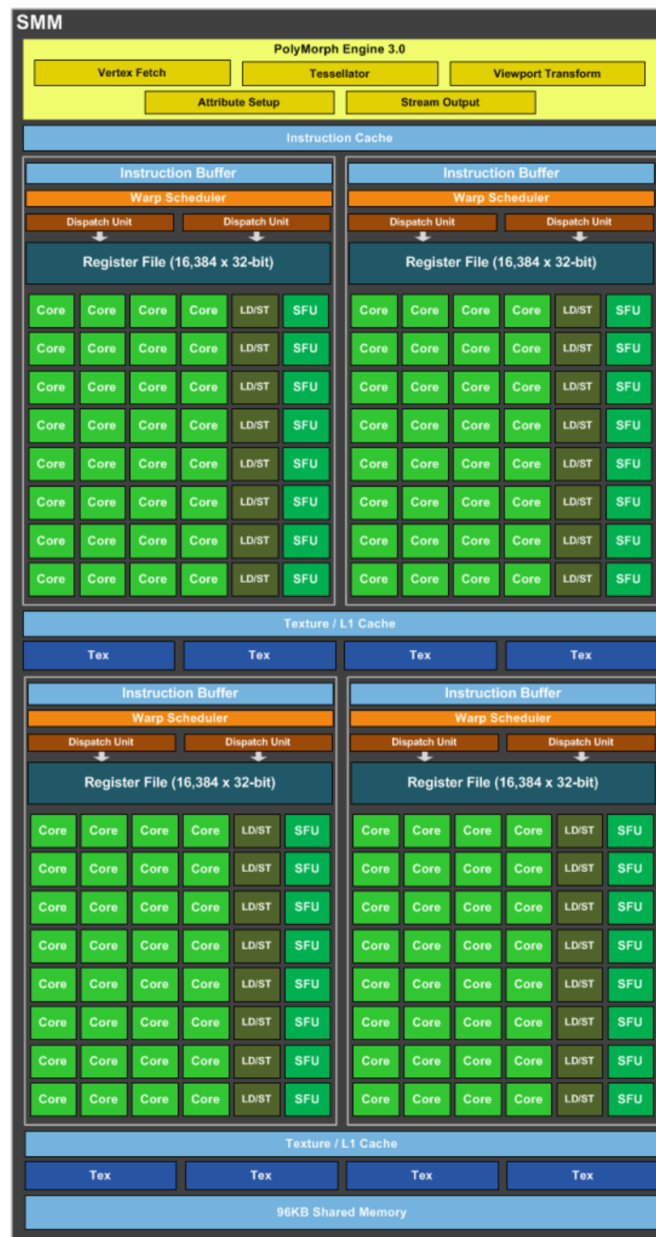


Source: (NVIDIA, 2014b)

Maxwell

It improves on Kepler by separating shared memory from the L1 cache, providing a dedicated 64KB shared memory in each SM. This architecture introduces native shared memory atomic operations for 32-bit integers and native shared memory 32-bit and 64-bit compare-and-swap (CAS) (HARRIS, 2014). Figure 5.7 illustrates the SM of Maxwell architecture.

Figure 5.7: Architecture of the Maxwell SM.



Source: (HARRIS, 2014)

Pascal

The main feature improvements on Pascal architecture are the *Unified Memory*, a significant advancement for NVIDIA GPU computing and software-based feature that provides a single, seamless unified virtual address space for CPU and GPU memory, it provides a memory accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space, and the *Compute Preemption*, this allows compute tasks running on the GPU to be interrupted at instruction-level granularity, and their context swapped to GPU DRAM. The Pascal architecture is also optimized to better performance of algorithm implementations for Artificial Intelligence (NVIDIA, 2017). Figure 5.5 illustrates the SM of Pascal architecture.

Figure 5.8: Architecture of the Pascal SM.

Source: (NVIDIA, 2017)

5.4 Assymetric low power architectures

The Asymmetric Manycore Processor (AMP) is a special case of HCS, which uses cores of different types in the same processor and thus embraces heterogeneity as

a first principle. Different cores in an AMP may be optimized for power/performance, different application domains or for exploiting different levels of parallelism (ILP, TLP, or Memory-Level Parallelism, MLP). They are used mainly in mobile systems because they are important to optimize energy for prolonging battery life during idle periods as it is to optimize performance for multimedia applications and data processing during active use.

Although big and little cores generally provide better performance and energy efficiency, respectively, in several scenarios no single winner may be found on the metric of energy-delay product (EDP). The big core may show better EDP in applications that compute intensive and have predictable branching and high data reuse. By contrast, the small core may be better for memory-intensive applications and applications with many atomic operations and little data reuse. Reconfigurable AMPs facilitate flexibly scaling up to exploit MLP and ILP in single-threaded applications and can scale down to exploit TLP in multithreaded (MT) applications.

Several challenges must be addressed to fully realize the potential of AMPs: The heterogeneous nature of AMPs demands complete re-engineering of the whole system. The cores of an AMP may have different supply voltages and frequencies, which presents manufacturing challenges; some techniques use offline analysis to perform static scheduling; however, they cannot account for different input sets and application phases; in static AMPs, thread migration may take millions of cycles, for example, in a big.LITTLE system with Cortex A15 and A7 processors the latency of moving tasks from the A15 to A7 and vice versa could be 3.75ms and 2.10ms; thread schedulers, threads running on an AMP may be unfairly slowed down, which leads to starvation and unpredictable per-task performance. (MITTAL, 2016)

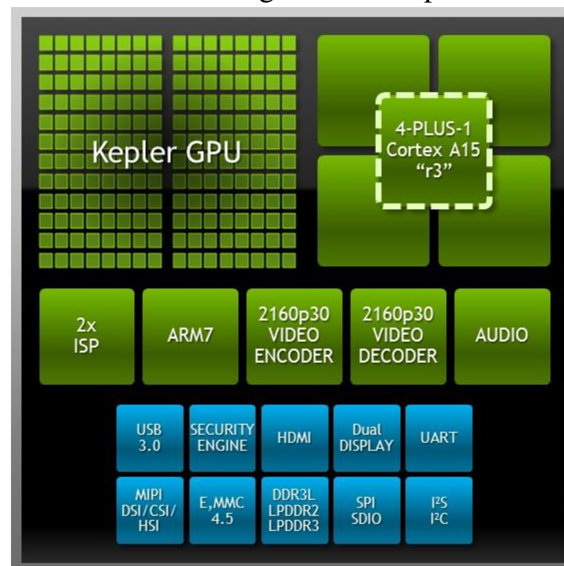
5.5 Target machines

Examples of HCS and AMP architectures used in this work are presented in Table 5.1 and Figure 5.9 presents the architecture of asymmetric node called Tegra K1. These machines are located at Industrial University of Santander (Colombia), BRGM (France) and Informatics Institute of UFRGS (Brazil).

Table 5.1: Heterogeneous architecture configurations.

	BRGM	*Salmão node*	*Guane-1*	*Tegra K1*
Processor	i7-3720QM	i7-930	Xeon E5645	ARM Cortex-A15
Clock (GHz)	2.60	2.80	2.40	2.3
CPU Cores	4	4	6	4
CPU Sockets	1	1	2	1
Accelerator	GeForce GTX 670M	Tesla K20c	Tesla M2075	NVIDIA Kepler GPU
Architecture	Fermi	Kepler	Fermi	Kepler
GPU Cores	336	2496	448	192
GPU Sockets	1	1	8	1
GPU RAM (GB)	1.3	5	4.9	2

Figure 5.9: Architecture of Tegra machine presented in Table 5.1.



Source:(NVIDIA, 2014a)

5.6 Concluding remarks

In this chapter, we introduced the heterogeneous architectures. We focused on platforms composed by the CPU host and the GPU devices. Improvements on GPUs are evolved to efficient float point precision work, and the limitations on these architectures are related to data movement between main memory and accelerator memory, and the idle time of free CPUs cores while GPU is computing. The GPU memory is a limitation that could be a bottleneck for the processing of large-scale problems. Consequently, for the CPU/GPU systems, out-of-core data management techniques are required to tackle the memory overflow problem in the GPU, when the size of a data exceeds the capacity of the DRAM. Although, the strategy of data partitioning between CPU memory and GPU memory could also affect the quality and efficiency of data processing when these out-of-

core techniques are applied due to the several data transferences between main memory and DRAM.

The most common programming model for NVIDIA GPUs is CUDA, it is very easy to use as programming procedure. CUDA is a conventional C adding a few of GPU platform specific keywords and syntax to define the parallel kernels. Older implementations of CUDA applications had to consider an explicit data movement from the main memory to DRAM, with unified memory lets programmers focus on developing parallel code without getting bogged down in the details of allocating and copying device memory.

Taking into account architectural improvements and limitations, the GPUs offer a platform for efficiency and speed-up optimization to applications with the possibility of data parallelism. In this sense, the geophysics numerical stencils are the kind of applications that can be implemented into these architectures with a well-suited performance improvement.

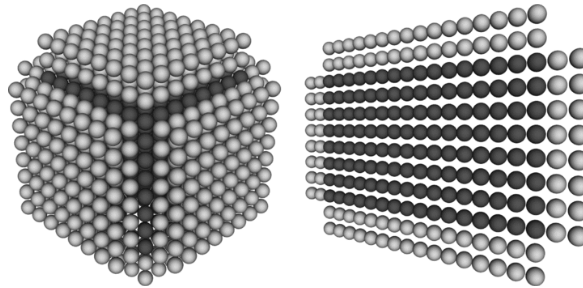
6 NUMERICAL IMPLEMENTATION OF GEOPHYSICS STENCILS ON HETEROGENEOUS PLATFORMS

The main idea is to divide the 3D grid into 2D slices. In this manner, each core of stream multiprocessors exploits the multi-threading parallelism to solve each point on this slice. In (MICIKEVICIUS, 2009), the authors use GPUs to solve a 3D stencil from the finite difference discretization of the wave equation. They create a 2D tile and their halos that are loaded into shared memory, then each thread block computes the 2D stencil. They also extended the solution for multi GPUs: first, they divided each slice into 2 GPUs and a computation of order k in space, data is partitioned by assigning each GPU half the data set plus (k/2) slices of ghost nodes. Each GPU updates its half of the output, receiving the updated ghost nodes from the neighbor; and second, in order to maximize scaling, they overlap the exchange of ghost nodes with each kernel execution.

In (ABDELKHALEK, 2007; ABDELKHALEK et al., 2009), the authors solve the acoustic wave equation and the Reverse Time Migration (RTM), a technique for creating seismic images in areas of complex wave propagation, and they used two approaches: first, they limited the stencil domain to use global memory on each GPU and to dedicate a CUDA thread for each grid point in the space domain; and second, instead of dedicating a thread to each grid point, they used a sliding window algorithm, creating slices in the z direction.

We used a standard implementation of seismic wave propagation model described in (MICHÉA; KOMATITSCH, 2010), the authors also subdivided the 3D space in 2D tiles, in the X and Y directions, and each tile corresponding to a block of threads for the GPU, to iterate along the third direction, in this case, the Z direction. The data of the 2D mesh tile and its halos are loaded in shared memory from global memory. Data sharing between GPUs is performed by Message Passing Interface. Figure 6.1 represents the data domain and the tiling division.

Figure 6.1: Tiling division of 3D data domain, each slice is computed by the GPU.



Source: (MICHÉA; KOMATITSCH, 2010)

6.1 Setup and Performance Measurement

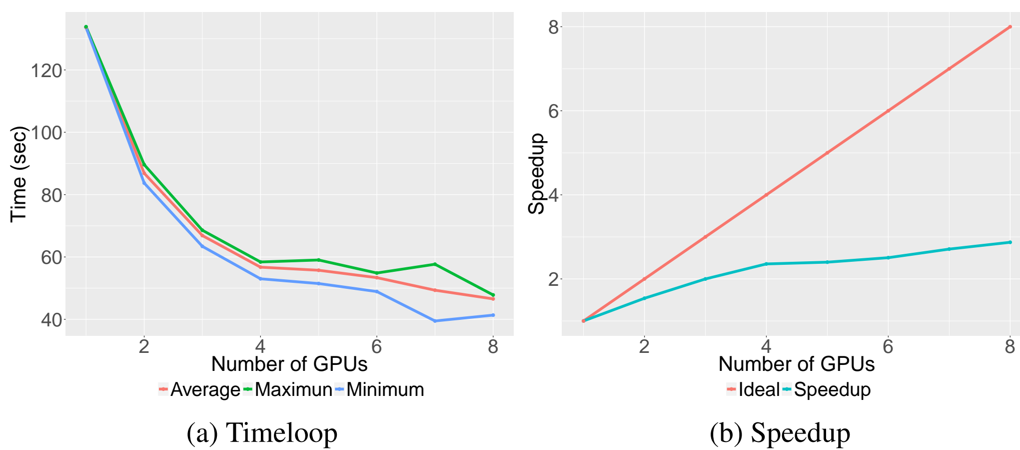
In order to analyze the workload on a cluster of GPUs, the seismic model (Ondes 3D) was executed on the HCS called Guane-1 (SANTANDER, 2015). Experiments were considered as follows: first, execution of seismic model in one GPU as the baseline to compare the application performance; second, to increment the number of GPUs to get an unbalanced problem, when the GPUs show an uneven utilization; third, to measure the time for execution and communication, the load and the memory usage of GPU; finally, to compare measures to find a relation between unbalanced load and the measures involved in this problem. The objective of this section is to find correlations between time (execution and communications), GPU load and memory usage. For each experiment was measured following data:

- **Timeloop:** average time to solve the model on each GPU (Kernel 1 plus Kernel 2)
- **Kernel 1:** time to calculate the stress kernel in the model.
- **Comm 1:** time of kernel 1 used to wait communications.
- **Kernel 2:** time to calculate the velocity kernel in the model.
- **Comm 2:** time of kernel 2 used to wait communications.
- **GPU load:** percent of GPUs utilization.
- **GPU memory:** percent of available memory usage.

6.2 Elapsed Time

Performance on HCS can be improved by increasing the number of GPUs, Figure 6.2 shows the minimum time, the average time and the maximum time of stencil execution on a multi-GPU node, we found a typical behavior for parallel applications, if the number of GPUs is incremented the execution time will be reduced, but the speedup of GPU implementation is not optimal, average execution with 8 GPUs is almost a $\times 3$ acceleration.

Figure 6.2: Timeloop and speedup measures of Ondes 3D application when increasing the number of GPUs.

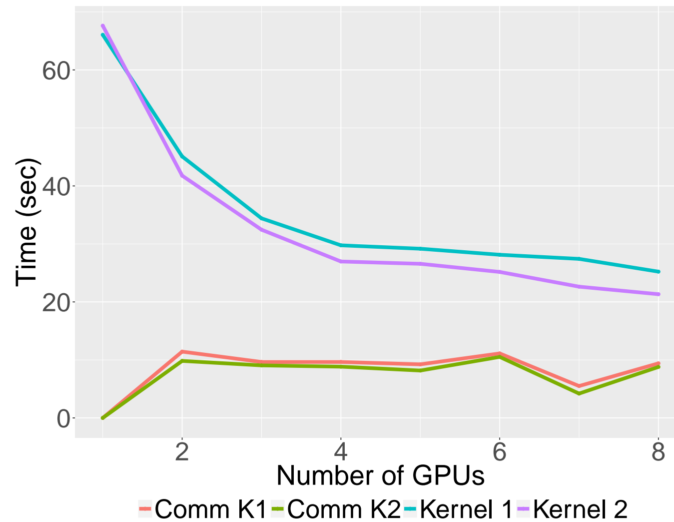


(a) Timeloop (b) Speedup

Source: The Author

Another measure to analyze the performance of HCS is the communication time between the host and the device. The seismic model executes two kernels on each GPU (Stress and velocity calculation). Figure 6.3 shows average time for kernel 1 and kernel 2. It shows that time for each one is quite similar. Another thing remarkable about this figure is that communication is increasing when the number of GPUs is increasing, the average time is almost the same for all executions.

Figure 6.3: Kernel execution and communication time of Ondes 3D application when increasing the number of GPUs.



Source: The Author

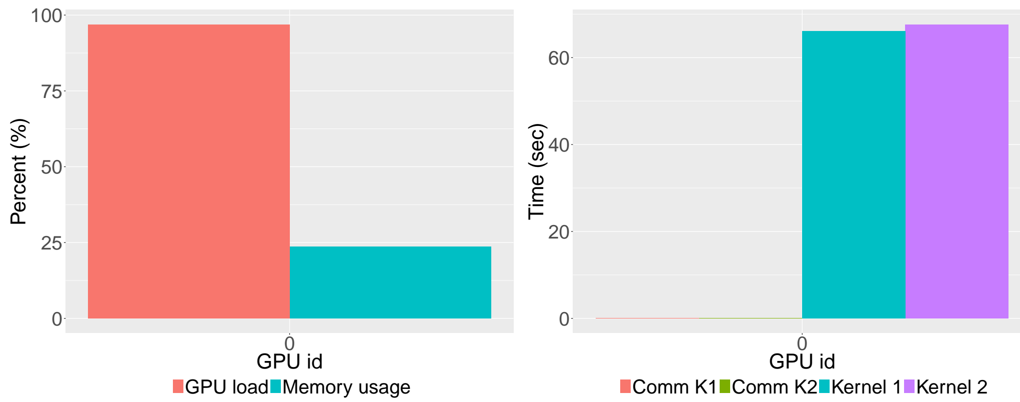
Measures presented in Figure 6.4 in this experiment are as expected, for one GPU there are no communication on kernels because there is only one GPU executing the simulation and GPU load is almost 100%. We measured for complete simulation since the time that CPU is processing to make the stencil and to send data to GPU memory. At the start of the simulation, the GPU is not working and waits until getting all data. It is remarkable that not all memory is used, almost 1/4 of memory is used in the simulation. This was one of the limitations related to (MICHÉA; KOMATITSCH, 2010).

If the number of GPUs is increased, the experiments start to reveal something that appears with many more GPUs. There is two kind of GPUs, the first solve the domain boundaries of simulation and for these, the load is greater than the other GPUs; the second, GPUs with lower load is solving the inner domain in grid space. Another thing, GPUs in the inner domain is using less memory than the others. Then, we can say that there are two factors affecting the load balancing: the first, data is not located in the GPU when is requiered; and second, getting new data causes waste of time for communications. Another factor that could be affecting the load balancing is the geometry in the simulation. GPUs that are solving the grid points near in the Y-axis have lower load than the GPUs that are solving the grid on the greatest values.

The experiment with the worst load balancing was with 8 GPUs, then we can say that uneven balance is caused by GPUs with high rates of communication has lower utilization, GPUs with greater load are solving the inner domain and values near to lowest values on Y-axis, for the grid space, GPUs with lower load is using less memory than the

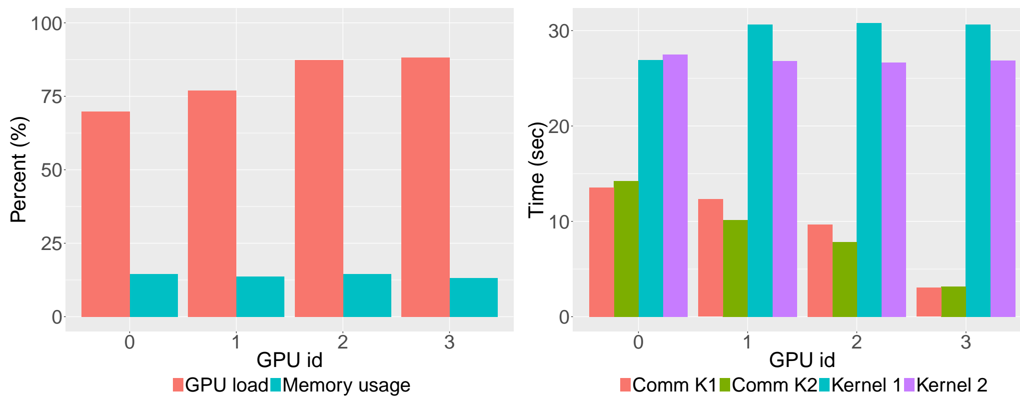
others with higher load.

Figure 6.4: Measures (time, load and memory consumption) of Ondes 3D on a cluster of GPUs.



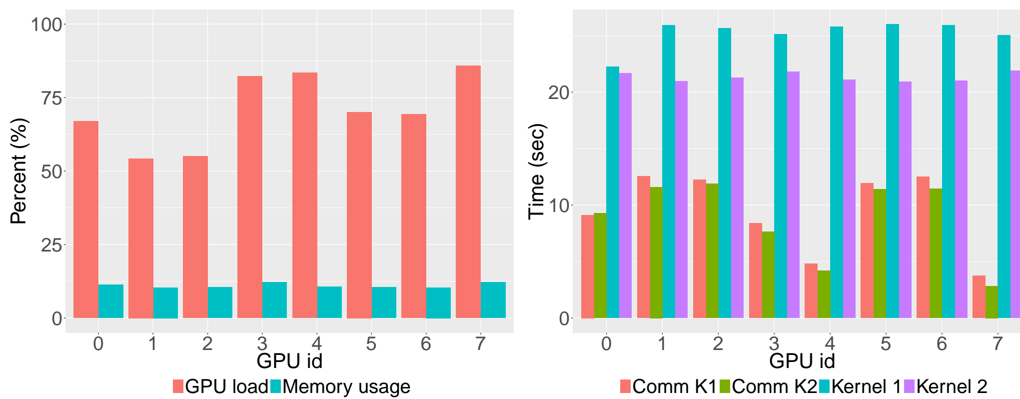
(a) GPU and memory usage for 1 GPU

(b) Time measures for 1 GPU



(c) GPU and memory usage for 4 GPUs

(d) Time measures for 4 GPUs



(e) GPU and memory usage for 8 GPUs

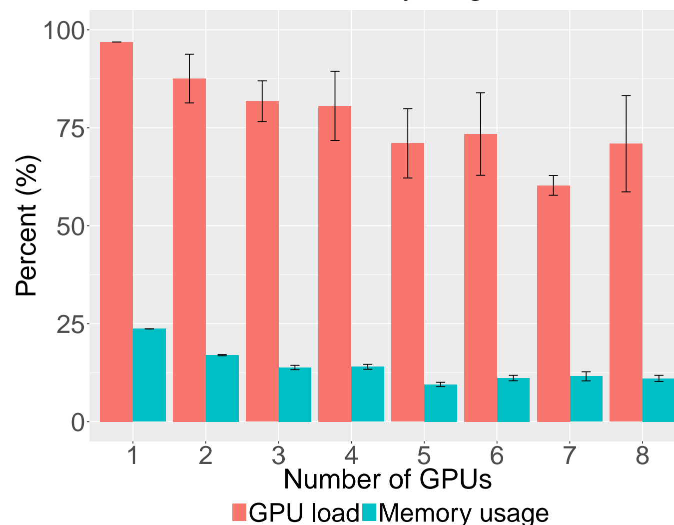
(f) Time measures for 8 GPUs

Source: The Author

6.3 GPU Load and Memory Usage

We found there are four factors that are affecting the load balancing for the seismic model, then we evaluate if these factors actually have influence for this problem, to prove this situation we measured GPUs load and memory usage and communication and we present these measures in Figure 6.5.

Figure 6.5: Statistical estimators (average and standard deviation) of Ondes 3D for GPU load and memory usage.



Source: The Author

Figure 6.5 presents average values for GPU load (red bars) and memory usage (blue bars) when we increased the number of GPUs, it shows clearly how the average load on GPUs is affected when more GPUs are used. Another situation in this figure is that memory usage has a similar behavior than the GPU load. Then, we estimate the correlation coefficient between these measures, and founded that this value is *0.88*; because of this correlation between memory and load we can prove that these two factors are highly correlated, that is if GPUs will have a lower load on execution they will have lower utilization of memory. Instead, if we estimate the value of the correlation coefficient between the GPU load and the kernel communications the result is *-0.30*, this value is not enough to prove any correlation between them.

Now, we will analyze the standard deviation (error bars) of these measures. We can see that multi-GPU experiment with 7 GPUs has the better balance and the experiment with 8 GPUs has the worst. The increment of standard deviation indicates a major difference between the values of this measure. Another thing we found on this figure is that a better-balanced experiment has a major value on the standard deviation of memory

usage. Then we estimate the correlation between the standard deviation of load an memory usage, this value is *-0.71*. Again, this new value is proving the correlation between load and memory usage, in this case, indicates if an experiment is unbalanced there is a high correlation to get lower utilization of memory. On the other hand, we estimate the correlation between the standard deviation of load and communications, the value is *0.80*. In this case, the value is shown that there is a high correlation between unbalanced experiments and time of communication, proving that these measures are also correlated.

6.4 Concluding Remarks

In this chapter, we presented another classical alternative to solve the geophysics numerical kernels. The standard implementation (based on Message Passing and CUDA) on heterogeneous architectures shows a performance improvement in terms of speedup and execution time.

Although there is a performance improvement, the available architecture is underused most of the time, and the unbalanced performance is increased for multi-GPU settings. In this sense, the performance for GPU-only systems remains not optimal. Furthermore, after allocating the workload to the GPU (i.e., starting the kernel) the CPU usually stays idle and waiting for the GPU to finish. The GPU memory bandwidth would act as a bottleneck (i.e., kernel communications), and the computational resources of GPUs could remain underutilized. This situation can be solved by data prefetch mechanisms on most recent architectures. Another challenge with standard implementation on HC is that the input configuration at runtime depends only on the number of GPUs and number of threads used by block, and there are no scheduling, or load balancing, strategies to improve the computing use of available heterogeneous cores.

One alternative to solve the unbalance problem, when increasing the number of available GPUs, could be to research on non-classical implementations to involve the free CPU cores into computations, and to avoid unnecessary memory transfers; in this sense, one challenge on HC is to exploit its computing capability, because common programming models on HC uses the CPU only as manager of the GPU device.

7 TASK-BASED APPROACH FOR PERFORMANCE IMPROVEMENT ON HETEROGENEOUS PLATFORMS

Task-based parallelism is a data-oriented programming model at the high level for heterogeneous architectures. The main idea is to build a task dependence graph, the runtime creates a queue of tasks with data directionality and schedules them into available processors. This programming model is exploited and developed mainly at Barcelona Supercomputing Center (BSC). Tasking works well when computations can be divided into blocks. The programmer specifies the distribution of data structures, and the compiler takes care of low-level details, such as the generation of messages and synchronization (HASSEN; BAL; JACOBS, 1998).

7.1 Runtime systems for task-based programming

One of these runtime systems is *SMP superscalar* (SMPSs), a programming environment focused for shared memory systems, as multicore and SMP platforms. A program in SMPSs follows the model of `#pragma` sentences that identify atomic parts and functions in the code that are candidates to be run in parallel in the different cores. Data directionality of parameters is explicit by `input` (Read only), `output` (Write only) and `inout` (Read/Write) clauses. At execution time, the runtime takes the memory address, size and directionality of each parameter at each task invocation and uses them to analyze the dependencies between them. Whenever a task is called, a node in a task graph is added for each task instance and a series of edges indicating their dependencies. At the same time, the runtime schedules the tasks to the different processors when their input dependencies are satisfied. Threads consume ready tasks from their own list in LIFO order, they get tasks from the main list in FIFO order, and they can steal from other threads in FIFO order. (PEREZ; BADIA; LABARTA, 2008; PEREZ; BADIA; LABARTA, 2010)

Recently works at BSC are oriented on a family of task-based runtime systems called StarSs. In this context, they are developed OmpSs, based on OpenMP for a task-based programming model with the indication of data directionality (DURAN et al., 2011). OmpSs specification includes a `target device` clause, for example, CUDA or OpenCL, the programmer needs to provide the code of the kernel for each one, and this code can be part of a task. Another support of OmpSs is the possibility of providing more

than one version of a given task (`implements` clause). The versions can target one or more devices. At runtime, the scheduler will decide which version should be scheduled taking into account some parameters such as execution time or locality of the data (FERNÁNDEZ et al., 2014). They have also integrated OmpSs with OpenCL framework to exploit the underlying hardware platform with greater ease in programming and to gain significant performance using data parallelism (ELANGOVAN; BADIA; PARRA, 2013).

Another framework is PaRSEC, it employs the dataflow programming and execution model to provide a dynamic platform that can handle heterogeneous hardware resources. The runtime combines the source program and the data flow information with supplementary information provided by the user and orchestrates task execution on the available processors. When a task is completed, the runtime reacts by examining the data flow to find what tasks can be executed and handles the data exchange between nodes. When no tasks are triggered because the hardware is busy executing application code, the runtime gets out of the way, allowing all hardware resources to be devoted to the application execution (BOSILCA et al., 2013b).

XKaapi is a task-based framework with locality-aware work-stealing algorithm to manage data locality and scheduling, fully asynchronous task execution strategy on GPUs, a light representation of tasks that allows to generate high degree of parallelism at low cost, and lazy computation of dependencies with an optimization that enables to move the overhead on the critical path rather than on the work (GAUTIER et al., 2013). StarPU is also a task-based runtime and it will be discussed in the next section.

7.2 StarPU runtime system

The StarPU runtime system performs well on task graphs with less regular patterns (matrix factorization, n-body problems using the fast multipole method, sparse linear algebra, h-matrix linear algebra). The main characteristics that may results in good or bad performance results are the amount of parallelism (if too few tasks are available for scheduling due to dependences, this may result in a bottleneck) and the granularity of tasks which must be heavy enough to limit the scheduling overhead, but high enough to allow for a good load balancing (AUGONNET et al., 2011).

Communication costs are evaluated by sampling the bus transfer performance between every pair of computing units (main CPU and accelerators). If several CPU sockets are available on the system, the transfer capability between each socket and each acceler-

ator is measured to account for non-uniform memory access effects (if an accelerator is closer from a given socket than from another socket). The results about bus performance sampling are stored in a file, for each computer node, and it is reused for subsequent runs unless a re-calibration is requested.

The StarPU runtime system developed by STORM Team provides a framework for task scheduling on heterogeneous platforms. It is able to calibrate the relative performance of multiple, heterogeneous implementations of computing stencils, as well as the cost of data transfers between the main memory space and accelerator memory spaces, such as to optimize work mapping dynamically among heterogeneous computing units, during the execution of the application. This scheduling framework jointly works with a distributed shared-memory manager in order to optimize data transfers, to perform replication and consistency management for avoiding redundant transfers, and to overlap communications with computations. A complete description of the available schedulers could be found in (BORDEAUX; CNRS; INRIA, 2014). StarPU considers the following scheduling algorithms:

eager: A central queue from which all workers pick tasks concurrently.

prio: A set of central queues, each associated with a priority level, from which all worker pick tasks concurrently.

ws: A work-stealing scheduler, where idle workers may steal work from busy workers.

random: A scheduler that maps work randomly among per-worker queues, according to the assumed worker performance.

dm: The Deque Model (DM) scheduler maps tasks onto workers using an history-based stencil performance model.

dmda: An variant of the DM scheduler also taking transfer costs into account.

dmdas: A variant of the DMDA scheduler such that per-worker task queues are sorted according to the priority of tasks.

dmdar: A variant of the DMDA scheduler such that per-worker task queues are sorted according to the number of already available dependent pieces of data.

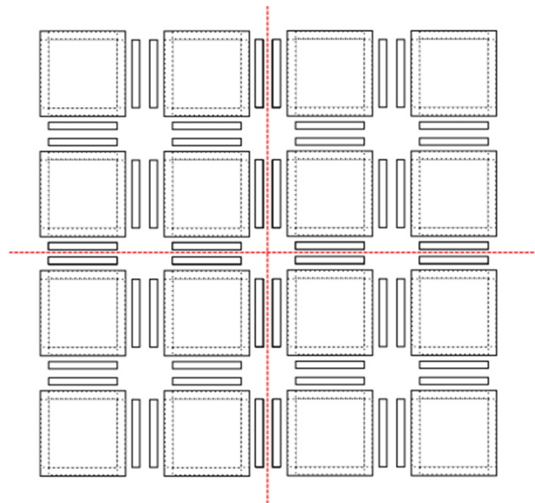
peager: A variant of the eager scheduler with the ability to schedule parallel tasks on multiple CPU cores.

pheft: A variant of the DMDA scheduler with the ability to schedule parallel tasks on multiple CPU cores.

7.3 Elastodynamics over runtime system

Finite-difference methods naturally express data parallelism whereas the model works at tasks level. In order to express task-based parallelism with such a numerical approach, one needs to split the model into a fine-grained grid of blocks where each part of the code (mainly the update of the velocity and the stress components) is executed on a single block as a computing task. Each block includes inner grid-points corresponding to the physical domain and outer ghosts zones for the grid points exchanged between neighboring subdomains (Figure 7.1). The three-dimensional domain is cut along the horizontal directions as models in seismology often describe a thin and wide crust plate in order to evaluate surface effects. A naive implementation would require expensive copies between blocks because the boundary data are not contiguous in memory (several calls of *memcpy*, or *cudaMemCpy*, for small size data). Therefore the GPU RAM buffer which is filled using a CUDA kernel is created and then copied only once.

Figure 7.1: Grid of blocks including inner grid-points corresponding to the physical domain and outer ghosts zones

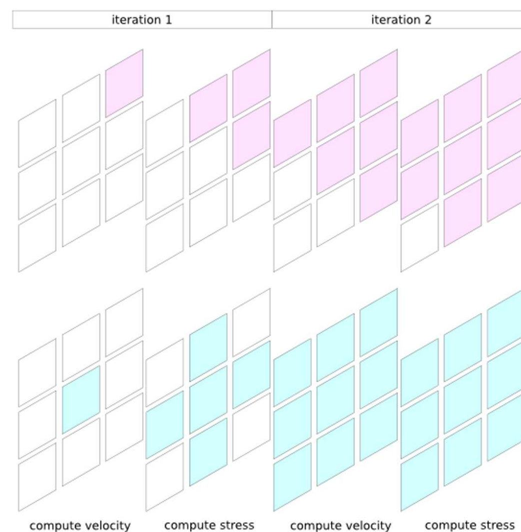


Source: (MARTINEZ et al., 2015b)

The task-based parallelism model leads to the creation of a large number of tasks, ideally loosely coupled. These tasks could be arranged as a Directed Acyclic Graph (DAG) according to the data dependencies. Typically, it can identify the tasks devoted to data transfers (used to save data from block boundaries to buffers and vice versa) and

the two computing kernels (computation of the six stress and the three velocity components). For each time step, the same pattern of task creation is repeated. At this stage, it has simplified the original implementation by discarding the absorbing boundary conditions. As these numerical conditions introduce load imbalance at the boundaries of the computational domain, it makes much more complex the analysis of the results on heterogeneous platforms. The time spent in data management kernels is very small compared to the elapsed time for the computation kernels. Nevertheless, these tasks are crucial as they express the dependencies between blocks that are adjacent in horizontal directions.

Figure 7.2: Tasks dependency on a grid of 3×3 blocks



Source: (MARTINEZ et al., 2015b)

Figure 7.2 illustrates this situation considering a grid of 3×3 blocks. For instance, if a single task is scheduled on slow computing resources and the remaining tasks are executed on faster resources, the colored tasks cannot be scheduled before finishing the first one. The main programming effort has been the creation of the relevant CPU and the GPU kernels corresponding to the numerical scheme.

7.4 Experiments

In this section, the architectures and configuration used for experiments are described. Experiments were executed on BRGM (Desktop) and GUANE-1 (HPC) nodes, and configuration of heterogeneous testbed are listed in Table 5.1 from Chapter 5.

Several scenarios for each node have been created, based on the memory consumption in the GPU (in-core, the data domain fits into GPU RAM, and out-of-core, the

data domain do not fit on GPU RAM) and the number of parallel tasks. The first example is based on a Cartesian mesh of an average of 2 million of grid points in order to fit in the memory available on the GPU. For the out-of-core example, requiring several data transfers between global and GPU RAM, it was selected two different sizes of the problem, for the commodity node was used a three-dimensional grid of approximately 32 million of points and 80 million of points for the HPC node. The other parameters (number of tasks for instance) strongly depends on the block size variable. Table 7.1 presents memory consumption and the number of tasks to be scheduled.

Table 7.1: Memory consumption, number of blocks and number of parallel task for the simulated scenarios

	In-core	Out-of-core	
		BRGM node	GUANE-1 node
Memory (GB)	0.18	2.71	6.57
Number of blocks	1	9	16
Computation Tasks	40	200	600
Communication Tasks	0	424	1800

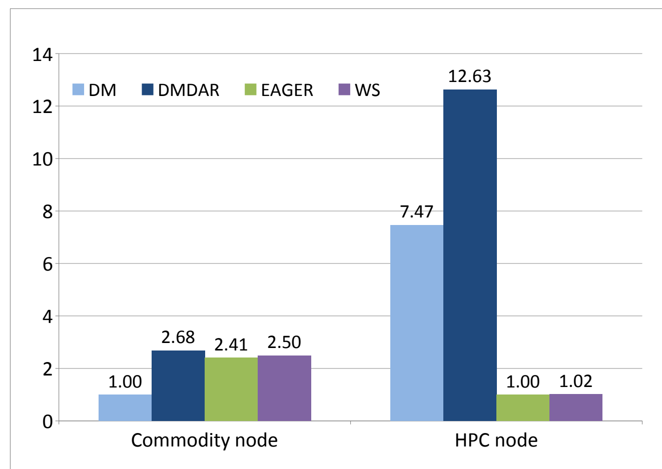
Experiments were performed using several configurations in terms of processing cores usage. For pure GPU experiments, the model is simulated only using the GPU cores available on the target architecture. Symmetrically, the pure CPU executions only target x86 cores. The hybrid experiments tackle all the computing cores available (CPU and GPU cores). It is because StarPU runtime system has two flags for heterogeneous cores utilization (workers): `STARPU_NCPU` to define the number of CPU cores and `STARPU_NCUDA` to define the number of GPU cards to be used on the computation of stencils. According to each node, the number of cores for CPU computation corresponds to the number of available physical cores (each GPU need a CPU core to management). The `STARPU_CUDA_ASYNC` flag enables concurrent stencil execution on graphics cards that support this feature (BORDEAUX; CNRS; INRIA, 2014). In this case, this flag is available on both machines.

7.4.1 Scheduling strategies

One of the key contributions of StarPU runtime system is to provide several scheduling algorithms adapted to various computing load and hardware heterogeneity. In this section, several different schedulers are compared. Figure 7.3 shows the speedup over the worst result on each platform. On the commodity-based platform, it considers one

GPU and three CPU cores. In this case, the best results are provided by the DMDAR algorithm that takes into account the cost of data transfers. Eager and Work Stealing algorithm appear like alternatives to DMDAR. These two algorithms do not use information from the memory bus. The rather good results underline the limited contention at the memory bus level for this configuration. The situation is rather different on the HPC node. The best results are also obtained with the DMDAR algorithm but the performance with the other schedulers is very poor. In this case, it uses eight GPU and four CPU cores and data transfers optimization is critical, especially for multi-accelerator platforms with major bottlenecks. The scheduling strategy can be selected by the `STARPU_SCHED` environment variable.

Figure 7.3: Impact of the scheduling algorithms for experiments on heterogeneous platforms. Relative speedup over the worst situation on each platform.



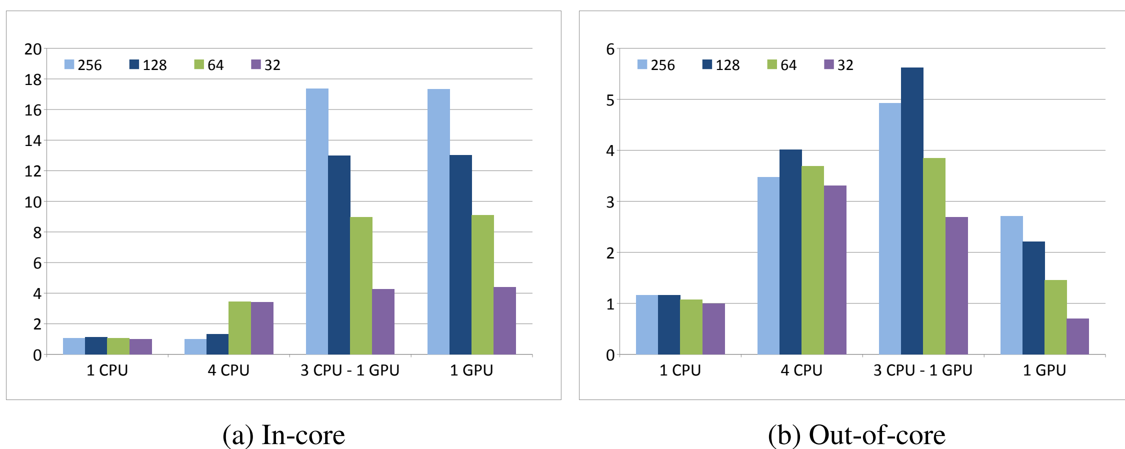
Source: (MARTINEZ et al., 2015b)

7.4.2 Size of the block

Selecting the best granularity is crucial to maximizing the performance. Considering heterogeneous platforms, this parameter is of great importance. Figure 7.4 shows the speedups over one CPU core for in-core and out-of-core data set. The results have been obtained on the commodity node. It can be observed that depending on the hardware configuration, the most efficient granularity may vary. In both cases, the efficiency with one GPU is optimal with a block size of 256. When it decreases the size of the block the GPU efficiency is significantly reduced (from $\times 17.3$ to $\times 4.3$ for in-core problems). This is because GPU architecture could deliver an optimal level of performance when all stream processors are used. Then, it means that larger block performs better on such architecture.

The situation is rather different on CPU platforms as tiny blocks could improve locality and cache effects. In this case, using a large number of blocks is also mandatory to extract enough concurrency from the task-based algorithm. Indeed, the wavefront decomposition reaches a good level of efficiency with blocks of the size equal to 64 or less for the in-core problem. For the out-of-core problem, it generates enough tasks to benefit from the four CPU cores in all cases. The bigger problem size corresponding to more computing tasks explains this result.

Figure 7.4: Impact of the granularity on the efficiency of seismic wave modeling on GPUs+CPUs.



(a) In-core (b) Out-of-core

Source: (MARTINEZ et al., 2015b)

Indeed, the choice of the appropriate granularity relies on a tradeoff between the performance level of the computing tasks depending on their sizes and the suitable number of tasks necessary to provide enough parallelism. Obviously, the optimal configuration is not always possible depending on the overall size of the domain and speedup ratio between the processing units.

7.4.3 In-core dataset

In this section, the overall performance is analyzed considering a problem size that fit in the GPU memory. Obviously, this situation is not the most suitable to benefit from the heterogeneous implementation as the price to pay for data transfers between the CPU and the GPU may overcome expected gains from the additional CPU cores.

Table 7.2 shows the results obtained on the commodity computing node (the elapsed time for pure CPU cores is the baseline). Firstly, it can be noticed that the speedup measured with one GPU over four CPU cores ($\times$5.03) is in the same order of magnitude of

Table 7.2: Speedup on the commodity-based hardware configuration (in-core dataset) over multicore execution

	pure GPU	hybrid (dmdar)	hybrid (ws)
Speedup	$\times$5.03	$\times$5.02	$\times$0.42

the results reported in (ABDELKHALEK et al., 2012; MARTINS et al., 2014; MICIKEVICIUS, 2009). In this case, the overhead coming from StarPU runtime system is limited as the number of blocks is very small and the computing stencils are solely scheduled on the GPU.

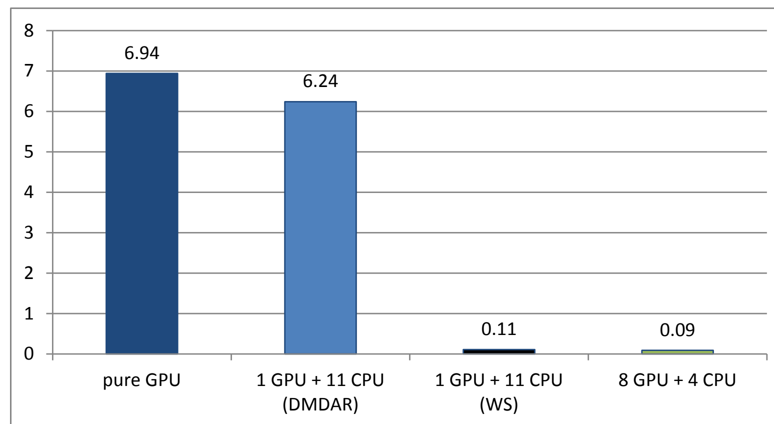
Hybrid simulations are supposed to exploit both GPU and CPU cores with the DMDAR scheduler that takes into account the cost of data transfer. Indeed, this strategy schedules all the computing tasks on the GPU because of the high cost of data transfers. As a result, the similar level of performance is observed if compare to the pure GPU implementation. In order to force the usage of both type of cores, the scheduling policy was changed using the work-stealing algorithm. This strategy simply implements a situation where idle workers steal work from busy workers. The results are very poor as the original and elapsed time is almost multiplied by a factor of $\times$2.3.

Figure 7.5 shows the results for the HPC computing node (the twelve CPU cores results are the baseline). Similarly to the previous results, the speedup measured with one GPU appears consistent with the scientific literature ($\times$6.94). The impact of the Tesla card available on the HPC node is significant, a ratio of 4.27 is observed between the pure GPU results on these two architectures. Previous remarks on the DMDAR and the work-stealing algorithm remain also valid. In this case, a stronger degradation of the speedup is noticed when using all the CPU cores compared to previous experiments. This probably could be explained by the higher level of performance of the GPU on this machine. The situation is even worst when eight GPU and four CPU cores are used. The elapsed time is increased by more than a factor due to the data transfers and the poor usage of the available resources. The granularity of the problem also plays an important role in this degradation.

7.4.4 Out-of-core dataset

This section discusses the results on heterogeneous architectures when the size of the data exceeds the memory available on the GPU. In this case, the accelerators are fully used as additional computing resources to the computing power delivered by the CPU

Figure 7.5: Speedup on the HPC node (in-core dataset) over multicore execution



Source: (MARTINEZ et al., 2015b)

cores. Data transfers are of great importance and should be carefully controlled by the scheduling strategies.

Performance obtained on the commodity node is detailed in Table 7.3. Four CPU cores results are used as a baseline. With one GPU, the simulation is slowed down in comparison with the baseline configuration. It is because of the price of data movements between the CPU and the GPU main memory. Indeed, the idle time ratio for the GPU cores reaches a maximum of 80.16% preventing any acceleration over the pure CPU version.

Table 7.3: Speedup on the commodity-based configuration (out-of-core dataset) over multicore execution

	pure GPU	hybrid (4 CPU cores and 1 GPU)
Speedup	$\times 0.92$	$\times 1.32$

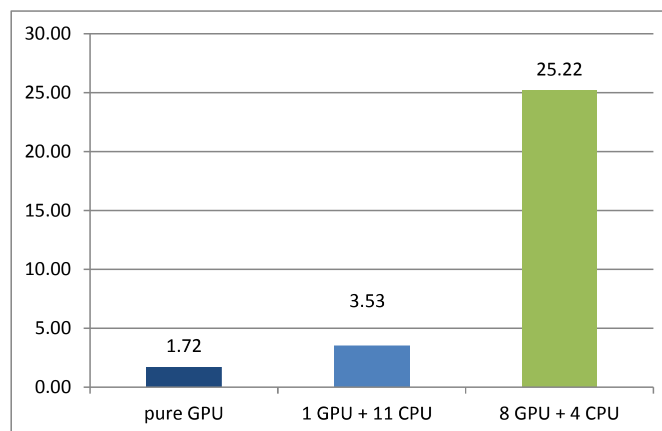
The heterogeneous results are slightly better with a speedup of $\times 1.32$. The best results are obtained with the DMDAR scheduler that takes into account the cost of data transfers. Contrary to the results obtained with the in-core dataset, the fine-grained of the decomposition of the large problem (at least 624 tasks) allows to schedule the computing load on all the computing resources available. Nevertheless, the usage of the cores remains low. The idle time for the three CPU cores varies between 41% and 79%, with maximum value for the GPU.

Results on the HPC node are summarized in Figure 7.6. All configurations show a speedup over the baseline results. For the pure GPU experiments, the acceleration reported ($\times 1.72$) demonstrates the benefits from task-based programming even when only the GPU card is used. In comparison with the commodity-based architecture, it benefits from larger bandwidth at the I/O bus level. This value is still far from the average

acceleration reported when the problem fits in the GPU memory ($\times 6.94$).

Combining one GPU and the eleven remaining CPU cores leads to an increase of the speedup compared with the GPU only configuration. In this case, our implementation is able to smoothly benefits from this additional computing power by almost doubling the performance level obtained with one GPU. Using both GPU and CPU cores could provide more flexibility for the scheduling of the computing tasks. As a side effect, it also observed a better usage of the GPU resources. The multi-GPU results confirm this trend with a maximum speedup of $\times 25.22$. Considering the various parameters that need to be taken into account, any tentative of exhaustive scalability analysis would be false. Indeed, this result should be compared to the corresponding speedup for the in-core dataset with one GPU ($\times$6.94) to understand the remaining effort to fully optimized our implementation on multi-GPU.

Figure 7.6: Speedup for out-of-core dataset when running on the HPC node over multicore execution.



Source: (MARTINEZ et al., 2015b)

7.5 Summary of runtime parameters

At this point, we found that proposed implementation for heterogeneous architectures is also influenced by an input configuration set. The parameters are determined by environment variables defined by StarPU as `STARPU_NCPU`, `STARPU_NCUDA` and `STARPU_SCHED`, analogously to the environment variables in OpenMP, and changing these parameters allows to improve the performance of the seismic stencil kernel. In this sense, the proposed implementation reach a well-performed execution when compared to multicore architectures. The input parameters are listed in Table 7.4.

Table 7.4: List of runtime parameters that affect the performance of task-based implementation on heterogeneous architectures

Parameter	***Description***
Number of GPUs	StarPU allows to use a multi-GPU cluster, this value is determined by `STARPU_NCUDA` environment variable.
Number of CPU cores	It depends on number of free CPU cores, because each GPU needs one CPU core for its management, and describes the available CPU cores to compute tasks, this value is determined by `STARPU_NCPU` environment variable
Task size	This parameter determines the number of points to be solved by each task.
Number of tasks	It depends on size of problem to be solved and can be defined by a rate between the data domain, the whole 3D grid, and the task size.
Scheduling algorithm	The scheduling strategy distribute the tasks into CPUs and GPUs cores, it can be selected by `STARPU_SCHED` environment variable.
Memory consumption	Two situations are considered: in-core data, when the data domain from complete 3D grid fits on DRAM; out-of-core data, when the data domain requires several data transferences.

7.6 Task-based implementation for energy efficiency

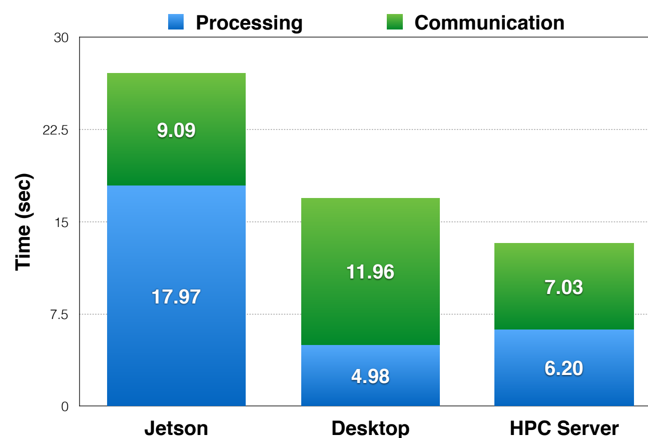
As a supplementary result, one of the advantages of task-based implementation is that can be executed on recent heterogeneous architectures oriented to energy saving. The objective of this section is to demonstrate that Nvidia Jetson manycore architecture represents an alternative for energy efficient seismic wave modeling. We used a simulation scenario for the three-dimensional model that includes a Cartesian mesh of 7.2 million grid points (300$\times$300$\times$80). The memory consumption for this problem is 625 MB of memory. Based on StarPU task-based implementation, the two computing stencils (velocity and stress components) must be scheduled to use 36 parallel computing tasks and 144 parallel communication tasks (data sharing between blocks). DMDAR scheduler maps tasks onto workers using an history-based stencil performance mode such that per-worker task queues are sorted according to the number of already available dependent pieces of data. Experiments were executed on *BRGM*, *Guane-1* and *Tegra K1* nodes listed in Table 5.1, from Chapter 5. In this case, we used three metrics to discuss the performance:

- Time-to-solution (execution time for the simulation)
- Energy consumption (using platform sensors by NVPROF).
- Energy efficiency (FLOPS/Watt)

7.6.1 Computing time

The results described in Figure 7.7 confirm the first assumption, Nvidia Jetson board exhibits poor performance considering the time-to-solution metrics. The Guane-1 node is ×2.05 faster than the Nvidia Jetson board. For the commodity-based architecture, the ratio is ×1.59x. This is mainly coming from the different levels of performance of the GPU available on each platform.

Figure 7.7: Comparison of Time-to-solution metrics for Tegra K1 (Jetson), Guane-1 (HPC server) and BRGM node (Desktop platform).



Source: (MARTINEZ et al., 2015a)

If we considered the ratio between computation time (tasks to solve velocity and stress stencils) and communication time (tasks to transfer data between global RAM and GPU RAM, and thread synchronization) Jetson board is processing 66.40% of the time, while desktop and server are processing only 29.39% and 53.11% of time respectively. This is because the Jetson has a shared memory between CPU cores and GPU. As a result, it only pays the costs of threads synchronization of GPU stencils and CPU functions. Due to the size of the test-case (625 MB) and the speedup ratio between the GPU and the CPU cores, StarPU uses both CPU and GPU cores to schedule the computing tasks on the Nvidia Jetson board. For the desktop machine and the server node, the runtime system

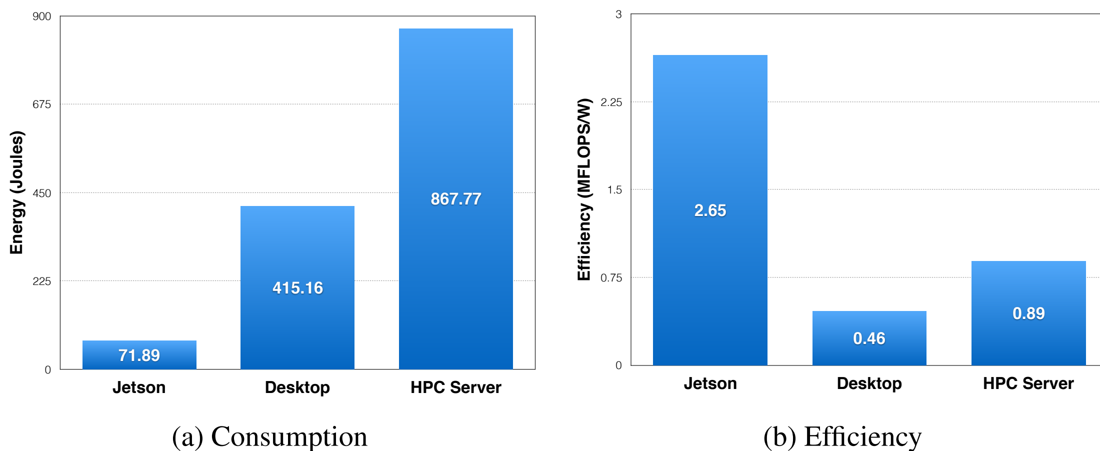
only schedules the computation tasks on the GPU and the remaining communication tasks on the CPU cores.

7.6.2 Energy efficiency

We also analyzed the energy consumption. For Jetson board we assumed that power consumption corresponds to 4W because we can not measure it, this value represents an upper-bound when CPU and GPU cores are working and interface ports (USB, HDMI) are disabled (NVIDIA, 2014a); for desktop and server machines power measures are measured from Nvidia Profiler. For the computation phase, power consumption is 83.40W and 123.49W for the desktop machine and the server node respectively.

If we compared energy consumption presented on Figure 7.8 (left), Jetson board reduces the energy consumption by a factor $\times 12.21$ in comparison with the server node and by a factor $\times 2.08$ with respect to the desktop machine. Obviously, the desktop machine is tuned for energy-efficiency contrary to high-end HPC node.

Figure 7.8: Energy measures for the earthquake modeling on three heterogeneous architectures.



(a) Consumption (b) Efficiency

Source: (MARTINEZ et al., 2015a)

Finally, the flops/watt ratio for each platform (Figure 7.8, right) was measured. Jetson board is $\times 5.82$ more efficient than desktop machine and $\times 2.99$ than a standard HPC node. One important remark should be underlined at this stage. The problem under study is tailored to fit in the memory available on the Nvidia Jetson Card (2 GB). This parameter is of great importance to discuss the overall performance as the size of the problem (and the number of blocks) could significantly influence the speedup on GPU architectures (MICHÉA; KOMATITSCH, 2010; MARTINEZ et al., 2015b).

7.7 Concluding remarks

We introduced a task-based implementation and analyzed its performance, with this approach we obtained a significant performance improvement when compared to multicore architectures. We demonstrated that changes on input configuration set at runtime, as scheduling policies combined with different task sizes, may considerably affect the efficiency and performance of the seismic wave kernels (MARTINEZ et al., 2015b). Revisiting the standard data-parallelism arising from the finite-difference numerical method, stencil computations, therefore, benefit from StarPU runtime system in order to smoothly schedule the DAG (Directed Acyclic Graph) on the current heterogeneous platform. In this way, the numerical algorithm and the underlying architecture are decoupled by fully exploiting the versatility of modern runtime systems. This approach allows tackling the complexity of this memory-bound problem by minimizing data movements to and from the accelerators.

The efficiency of stencil computation on two heterogeneous architectures, using the maximum number of processing units available, consider a real problem that could not fit in the DRAM memory, for example, a maximum speedup of 25.22 using four CPU cores and eight GPU in comparison with the same experiments using twelve CPU cores. Using a commodity-based architecture with four CPU cores and one GPU was obtained an acceleration of 32% over the CPU version. The analysis of the performance underlines the significant impact of the granularity and the scheduling strategy.

A detailed comprehension of the tradeoff between the granularity and the scheduling policies is necessary in order to tackle more complex architecture. For instance, the regularity of the finite-differences numerical method could allow deriving a cost-model that could help the runtime system to improve task scheduling. This includes potential irregularity in the granularity of the task to maximize efficiency on multiple devices. Including absorbing boundary conditions generates load imbalance that could be smoothly reduced by the fine-grained task-based programming model. Difficulties on this model are related to runtime configuration (i.e., scheduling algorithm, size and number of tasks, choosing of processing units).

As an additional advantage of the proposed task-based implementation, a low-power manycore architecture can be an alternative to compute stencils of seismic modeling with a better energy efficiency, better usage of the available resources (CPU and GPU cores) and a significant reduction of the communication cost (33.60%). Good results in

terms of energy-to-solution compared to common HPC systems are possible. But the size of problems that could be tackled is limited by the amount of memory available on these boards. Finally, the emerging of integrated cores architectures that deliver higher bandwidth between GPU and CPU appears like an opportunity to tackle both the PCIe bottleneck and the energy-efficiency challenge.

8 RELATED WORK

In this chapter, we present the several works that are related to performance optimization of stencil computations. A large scientific literature has been devoted to the adaptation of stencil algorithms on HPC architectures. It involves from hardware implementations, low-level optimization, compiler flags, application tuning, programming of new algorithms, ML models to improve the performance of these applications, and task-based implementations of numerical stencils.

8.1 Perfomance improvement of stencil applications on multicore architectures

Parallelism and HPC are the basis to develop the stencil algorithm and multicore architectures were extensively used to improve the performance of these computations. Performance improvement can be reached by algorithm implementations and cache optimization is a common way to reach a better performance on multicore architectures. These works oriented the performance characterization presented in Chapter 3.

In (DATTA et al., 2009), the authors use a methodology to optimize stencil computations for multiple architectures (multicore and accelerators). Their work reuse cache and methodologies across single and multiple stencil sweeps, examining cache-aware algorithms as well as cache-oblivious techniques. Their results demonstrated that recent trends in the memory system organization have reduced the efficacy of traditional cache-blocking optimizations.

In (DUPROS; DO; AOCHI, 2013), the authors review the scalability issues of seismic wave propagation numerical kernels on x86 architectures, they have underlined the limitations coming from the fine-grained parallelism leading to a degradation of the load balance.

In (MALAS et al., 2015), the authors combine the multicore temporal blocking and a diamond tiling to reduce the memory pressure, the results show performance advantages in bandwidth-starved situations.

In (DUPROS et al., 2015) the authors also implement stencil algorithms focused on cache improvement by spacetime blocking. The trend for these HPC applications is to pay a higher cost in order to optimize the overall performance. In this work is explained the Naive and Space tiling algorithm implementations analyzed in our research.

In (SAXENA; JIMACK; WALKLEY, 2016) we found another work related to

cache optimization, the authors optimize the cache reuse for stencil-based PDE discretizations by a domain decomposition that minimizes cache misses in structured 3D grids.

In terms of regression models and performance prediction applied to multicore architectures, the impact of different optimizations is difficult to predict due to the influence of load imbalance, synchronization overhead, and cache locality.

In (RAHMAN; YI; QASEM, 2011), the authors present a performance study for stencil computations on the Intel Nehalem multicore architecture, they model the overall performance of differently optimized code based on the impact of optimizations on individual architectural components measured by hardware counters, and apply regression analysis to a large collection of empirical data to derive the model and verify the precision of the approach.

In (STENGEL et al., 2015), the authors use the Execution-Cache-Memory (ECM) model, it delivers a prediction of the number of CPU cycles required to execute a certain number of iterations n_{it} of a given loop on a single core, and use this methodology to quantify the performance bottleneck of stencil algorithms on an Intel SandyBridge processor, and they study the impact of typical optimization approaches such as spatial blocking, strength reduction, and temporal blocking.

8.2 Advanced optimizations, low-level and auto-tuning strategies

Application tuning represents a classical methodology to improve the performance on multicore architectures. Finding the optimal value for each parameter requires to search on a large set of configurations and several heuristics or frameworks have been proposed to speed up the process of finding the best configuration for scientific applications. Unfortunately, application tuning leads to the exploration of a huge set of parameters, thus limiting its interest on complex platforms.

In (DURSUN et al., 2009), the authors propose a multilevel parallelization framework that combines inter-node parallelism by spatial decomposition, intra-chip parallelism through multithreading, and data-level parallelism via single-instruction multiple-data (SIMD) techniques.

In (MERCERAT; GUILLOT; VILOTTE, 2009), the authors present an efficient numerical spacetime decomposition for acoustic wave propagation based on the *Parareal* algorithm, it is based on a decomposition of the time interval in time slices. It involves a serial prediction step based on a coarse approximation, and a correction step (computed

in parallel) based on a fine approximation within each time slice.

In (KAMIL et al., 2010), the authors present a stencil auto-tuning framework for multicore architectures that converts a sequential stencil expression into tuned parallel implementations. The algorithm creates a large space of preceding optimizations, run each one of these combinations and reports fastest parameter combination. Overall, the main problem of these works is that the search domain can be very large and searching the best configuration would take too much time.

In (CRUZ; ARAYA-POLO, 2011), the authors predict the performance behavior of stencil computations by using a model based on cache misses and prefetching. They create one configuration vector and one performance vector and use (BACH; JORDAN, 2003) to obtain auto-tuned best configuration.

In (TANG et al., 2011), the authors present the *Pochoir* framework, it allows a programmer to write a simple specification of a stencil in a domain-specific stencil language embedded in C++ which the Pochoir compiler then translates into high-performing Cilk code that employs an efficient parallel cache-oblivious algorithm.

In (CHRISTEN; SCHENK; CUI, 2012), the authors introduce a code generation and auto-tuning framework for stencil computations targeting modern multi and many-core processors. The goals of the framework are productivity and portability for achieving high performance on the underlying platform.

In (MIJAKOVIC; FIRBACH; GERNDT, 2016), the authors develop a tuning framework that integrates and automates performance analysis and performance tuning. They introduce the *Periscope Tuning Framework* (PTF), a flexible plugin mechanism and provides tuning plugins for various different tuning aspects. The output of the framework is tuning recommendations that can be integrated into the code.

In (BREUER; HEINECKE; BADER, 2016), the authors present a detailed description of a clustered local time stepping scheme for the seismic simulation package *SeisSol* and optimize the performance by clustering elements of a similar time step. They turned the experiments on the SuperMUC Phase 2 (TOP500, 2017).

8.3 Machine Learning approaches

Because ML is a methodology for optimization that could be applied to find patterns on a large set of input parameters, recent works use this approach to improve the performance and to build regression models of HPC applications. Many of these works

use cache-related metrics.

In (RAI et al., 2009), the authors compare ML algorithms for characterizing the shared L2 cache behavior of programs on multicore processors. The results show that regression models trained on a given L2 cache architecture are reasonably transferable to other L2 cache architectures.

In (GANAPATHI et al., 2009) the authors apply ML techniques to explore stencil configurations (code transformations, compiler flags, architectural features and optimization parameters). Their approach is able to select a suitable configuration that gives the best execution time and energy consumption.

In (EOM et al., 2013), the authors compare workloads of applications executed on the cloud, they use several ML techniques (Trees, Neural Networks, Bayesian methods) to improve scheduling decisions in mobile offloading. Parameters in input set are local execution time, size of data to be transferred, network bandwidth and the number of invocations for an argument setup.

In (PUSUKURI; GUPTA; BHUYAN, 2013), the authors introduce a framework for co-scheduling of simultaneous programs based on supervised learning techniques for identifying the effects of the interference between multithreaded programs on their performance, a statistical model is trained to predict similar output values when similar input values are observed.

In (WENG; LIU; GAUDIOT, 2013) the authors propose a dynamic scheduling policy based on a regression model to ensure that is capable of responding to the changing behaviors of threads during execution, their scheduling policy could achieve up to 29% speedup.

In (SUKHIJA et al., 2014), the authors improve the performance by selecting a portfolio dynamic loop scheduling (DLS) using supervised ML techniques to build empirical robustness prediction models that are used to predict DLS algorithm's robustness for given scientific application characteristics and system availabilities.

And in (CRUZ; ARAYA-POLO, 2015), the authors improve the performance of stencil computations by using a model based on hardware counter behavior and ML. They have included several features in the model such as multi and many-core support, hardware prefetching modeling, cache interference and capacity misses and other optimization techniques such as spatial blocking and Semi-stencil. If we compare our proposed ML model with this work we have obtained a better accuracy of the prediction results.

8.4 Heterogeneous computing

Seismic wave modeling faces a major challenge to exploit current heterogeneous systems. Moreover, every vendor is actually working on next generation of machines that will drive the community to the Exascale, with millions of heterogeneous cores. Several strategies, frameworks, and programming models have been proposed mainly to optimize CPU and GPU implementations.

In (AUGONNET et al., 2011), the authors introduce a runtime system for heterogeneous platforms called *StarPU* and based on the integration of the data-management facility with a task execution engine. It includes a high-level description of every piece of data manipulated by the task and how they are accessed, it also offers a low-level scheduling mechanism, as we presented in Chapter 7, so that scheduler programmers can use them in a high level.

In (DURAN et al., 2011), the authors introduce *OmpSs*, a programming model based on OpenMP and StarSs, that can also incorporate the use of OpenCL or CUDA kernels. It implements the data dependencies and offers asynchronous parallelism in the form of tasks. A task can be annotated with data directionality clauses that specify the data used by it, and how it will be used (read-only, write-only, read-write)

In (GAUTIER et al., 2013), the authors introduce the *XKaapi* for data-flow task programming model on heterogeneous architectures, a tasking API that provides numerical kernel designers with a convenient way to execute parallel tasks over the heterogeneous hardware on the one hand, and easily develop and tune scheduling algorithms on the other hand.

In (VASUDEVAN; VADHIYAR; KALÉ, 2013), the authors introduce *G-Charm*, a generic framework with an adaptive runtime system for efficient execution of message-driven parallel applications on hybrid systems. The framework is based on a message-driven programming environment. It includes dynamic scheduling of work on CPU and GPU cores, maximizing reuse of data present in GPU memory, data management in GPU memory, and combining multiple kernels.

In (BOSILCA et al., 2013a), the authors introduce *PaRSEC*, an approach based on task parallelism. This strategy allows the algorithm to be decoupled from the data distribution and the underlying hardware, the algorithm is expressed as flows of data. PaRSEC is an event-driven system. When an event occurs, such as task completion, the runtime reacts by examining the data flow to discover what future tasks can be executed

based on the data generated by the completed task.

In (LACOSTE et al., 2014), the authors compare StarPU and Parsec runtime systems, they conclude that both runtime systems are able to benefit from heterogeneous platforms with comparable levels of performance.

The runtime systems can be also defined for code optimization such as in *BOAST*, it applies an automatic S2S transformation to optimize the performance of HPC architectures (CRONSIOE; VIDEAU; MARANGOZOVA-MARTIN, 2013; VIDEAU et al., 2018).

Several references implement stencil applications on these architectures and these runtimes, but most of these works don't exploit all the computing resources available or depend on a low-level programming approach.

In (MICIKEVICIUS, 2009), the authors describe a GPU parallelization approach for the 3D finite difference stencil computation, and they also describe the approach for utilizing multiple GPUs to solve the problem, achieving linear scaling with GPUs by using asynchronous communication and computation.

In (MICHÉA; KOMATITSCH, 2010), the authors describe the implementation of the code in CUDA to simulate the propagation of seismic waves in a heterogeneous elastic medium. They also implement convolution perfectly matched layers on GPUs to efficiently absorb outgoing waves on the fictitious edges of the grid. We used this implementation to analyze the standard implementation in Chapter 6

In (AGULLO et al., 2011a; AGULLO et al., 2011b), the authors implement LU and QR decomposition algorithm for heterogeneous architectures exploiting task-based parallelism on top of the StarPU runtime system and present two alternative approaches, respectively based on static and dynamic scheduling.

In (ABDELKHALEK et al., 2012), the authors design a fast parallel simulator that solves the acoustic wave equation on a GPU cluster. They considered a finite difference approach on a regular mesh, in both two dimensional and three dimensional cases, and studied different implementations and their impact on the application performance

In (CALANDRA et al., 2013) the authors evaluate a 3D finite difference stencil, that is optimized and tuned in OpenCL, executed on CPUs, APUs, and GPUs. Their results show that APU integrated GPUs outperform CPUs and that integrated GPUs of upcoming APUs may match discrete GPUs for problems with high communication requirements.

In (CASTRO et al., 2014), the authors analyzed the use of a low-power manycore

processors for seismic wave propagation simulations, they look at its characteristics such as limited amount of on-chip memory and describe the solution to deal with the processor features, and compared the performance and energy efficiency of a seismic wave propagation model on MPPA-256 to other commonplace platforms such as general-purpose processors and a GPUs.

In (BOILLOT et al., 2014), the authors study the applicability of task-based programming in the case of a Reverse Time Migration (RTM) application for Seismic Imaging. The initial MPI-based application is turned into a task-based code executed on top of the PaRSEC runtime system. Their results show that the approach can exploit much more efficiently complex hardware such as the Intel Xeon Phi accelerator.

In (ROTEN et al., 2016), the authors have implemented a seismic model in both the CPU and GPU versions. The optimized CUDA kernels utilize the GPU memory bandwidth more efficiently and the application has resulted in a significant increase of performance and accuracy for simulations in realistic earth structures. They turned the experiments on the NCSA Blue Waters and the OLCF Titan (TOP500, 2017).

In (TSUBOI et al., 2016), the authors realize large-scale computations by optimizing a widely used community software code (*SPECFEM3D_GLOBE*) to efficiently address all hardware parallelization, especially thread-level parallelization to solve the bottleneck of memory usage for coarse-grained parallelization. They perfomed the experiments on the K computer (TOP500, 2017).

9 CONCLUSION AND PERSPECTIVES

Scientific applications have been developed to understand the physical phenomena. In the case of geological studies, strong motion models have been used to mitigate the risk of building damages derivated from earthquakes. Moreover, geophysics exploration remains fundamental to the modern world to keep up with the demand for energetic resources, oil and gas industries rely on software as an economically viable way to reduce production costs. The fundamentals of many software mechanisms for geophysics are based on simulation engines. For instance, on seismic imaging, geological modeling, migration and inverse problems use simulators of wave propagation at the core.

Wave propagation model approximations are the current backbone for many geophysics simulations. These simulation engines are built based on PDEs solvers. The PDEs in each case define the accuracy of the approximation to the real physics when a wave travels through the earth. It has been extensively applied for earthquake modeling and imaging potential of oil and gas reservoirs, for the last years.

The most common numerical model used to solve the PDEs is the FDM method, it also lies at the heart of a significant fraction of numerical solvers in other fields (i.e., fluids dynamic, or climate modeling); and solving the geophysics simulation models from an FDM method requires a huge quantity of computations. With this in mind, HPC architectures are exploited to develop the geophysics applications and the overall parallel methodology is based on a classical Cartesian grid partitioning with the exchange of information on common edges.

One major challenge in HPC applications is to obtain the optimal performance. The trend at the hardware level is to increase the complexity of available computing node. This includes several levels of hierarchical memories, increasing number of heterogeneous cores or low-level optimization mechanisms. Another concern is coming from the increasing gap between the computing power and the cost of data transfers.

At the software level, the challenge is to develop as much as possible algorithm implementations independent of the knowledge of the underlying architecture. Many programming models that have been developed are oriented into the architecture. Principal methodologies are supported in shared-memory and message passing, for multicore architectures, and streaming multiprocessing, for heterogeneous architectures. But, the evolution of HPC paradigms leads to progressively re-design the current applications that mainly exploit standard programming models.

Additionally, in spite of the good speedups usually reported, the performance obtained with standard implementations of geophysics models on HPC nodes could remain not optimal from the best performance. In this context, the performance improvement implies to search in a large set of programming models, runtime configurations, and architectural features.

Consequently, this research was addressed on the hypothesis that performance optimization of geophysics applications can be done by searching the optimal input configuration set, at runtime. In this sense, our work was focused on developing a model based on an ML approach to finding the input configuration, on multicore architectures, and researching into new programming model implementations to exploit all the computing resources on heterogeneous architectures.

9.1 Contributions

The first part was dedicated to the performance analysis and efficient programming models of geophysics numerical kernels on multicore architectures (Chapter 2). We presented that the performance of numerical stencils if affected by the runtime parameters (Chapter 3). The challenge to find the optimal performance is related to searching in a large set of input configurations (number of threads, problem size, code optimization of looping, scheduling, and implemented algorithms). We considered that the performance measures are not only related to the execution time and the speedup. Hardware counter events (i.e., cache behavior) can also describe the performance of scientific applications, and understanding their correlations can help us to explain the architecture behavior.

We discussed that finding the optimal runtime set is quite difficult because there are several parameters that influence the performance measures. Thus, we introduced an ML-based model to predict the application performance, to find the optimal input configuration and to improve the performance of stencil applications on multicore architectures (Chapter 4). We presented that the ML-based model can be adapted according to the algorithm implementations (i.e., naive and space tiling) and the architectural features (i.e., number of available hardware counters, and memory mode on many-core architectures). Since the prediction model have been trained, it can be used to find the input configuration set to reach the optimal performance. The results of proposed ML model proved that performance prediction of geophysics numerical kernels can be done by building a regression model with high accuracy (up to 99%), by using a tiny set of experiments (less

than 1% of configuration set).

The second part was oriented to the performance analysis and efficient programming models of geophysics numerical kernels on heterogeneous architectures (Chapter 5). In this circumstances, we used a standard implementation of seismic wave propagation, implemented in CUDA, to analyze the performance of the application (Chapter 6). As we presented, the performance increments when we increment the number of available accelerators; but, there are other factors that also increase (data movement and communications, unbalancing processing, and memory consumption), and it does not allow to reach an optimal performance. On the other hand, the CPU processors are only used to manage the co-processing work, neither of them is used to stencil computing.

To exploit all the available computing power on heterogeneous architectures is necessary to implement new programming models, beyond the standard implementations. In this sense, we introduced a task-based implementation of a seismic wave propagation model (Chapter 7). This implementation allows a set of input parameters from a runtime configuration set to improve the performance. We analyzed this implementation by changing the input parameters: the type of processing units (CPU only, GPU only, and CPU/GPU hybrid), the memory consumption (related to data movement and communications), the scheduling algorithms (related to cost of computing on CPU or GPU cores), and the task size (related to the number of parallel task). This implementation is faster and gained a better performance when compared with standard implementations. The proposed heterogeneous task-based implementation can reach a performance improvement near to 25 times when compared to multicore architectures.

9.2 Future Work

This research can be extended in several ways, and we delight three possibilities as follows:

- **Exploring on new input configurations.** We proposed an improvement of performance based on the finding of optimal input configuration at runtime. We proved that in shared-memory programming the performance can be predicted by a trained ML-based model, and this prediction helps to find the optimal performance. We used a set of available parameters in OpenMP defined by `OMP_NUM_THREADS` and `OMP_SCHEDULE` variables. We believe that other parameters can be included as in-

put variables of the ML-based model for performance prediction. These parameters could be related to programming models (i.e., MPI-based, over-decomposition in virtual processes), compiler options (i.e., optimization flags, SIMD vectorization), or architectural features (i.e., FPGAs, embedded systems).

- **Extending the ML-based model to heterogeneous architectures.** The performance of our task-based implementation for the seismic wave propagation model is affected by an input configuration set. The different scheduling algorithms, the type of processing units and the variation in task size exhibited changes of application performance. Making decisions, as choosing if computations run on GPU or CPU cores, depend on parameters like memory consumption. Expanding the possibilities in configuration runtime create a large set of input parameters. As we demonstrated, this optimal input set can be found by an ML-based model. Then, one direction of future work could be to develop a prediction model based on available hardware performance counters in heterogeneous architectures.

- **Developing a new model based on unsupervised ML algorithms.** The main limitation of the proposed ML-based model is related to the time consumed in training and testing stages because we used a supervised method. We think that an unsupervised-based model would find the input set by auto-tuning techniques that converge to optimal performance. In this context, we may address the research towards clustering methods, genetic algorithms, or neural networks as self-organizing maps.

REFERENCES

ABDELKHALEK, R. **Évaluation des accélérateurs de calcul GPGPU pour la modélisation sismique**. Dissertation (Master) — ENSEIRB, Bordeaux, France, 2007.

ABDELKHALEK, R. et al. Fast seismic modeling and reverse time migration on a gpu cluster. In: **2009 International Conference on High Performance Computing Simulation**. [S.l.: s.n.], 2009. p. 36–43.

ABDELKHALEK, R. et al. Fast seismic modeling and reverse time migration on a graphics processing unit cluster. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd, v. 24, n. 7, p. 739–750, 2012.

AGULLO, E. et al. LU factorization for accelerator-based systems. In: **Proceedings of the 2011 9th IEEE/ACS International Conference on Computer Systems and Applications**. Washington, DC, USA: IEEE Computer Society, 2011. (AICCSA '11), p. 217–224.

AGULLO, E. et al. QR factorization on a multicore node enhanced with multiple GPU accelerators. In: **25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings**. [S.l.: s.n.], 2011. p. 932–943.

AUGONNET, C. et al. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., v. 23, n. 2, p. 187–198, feb. 2011. ISSN 1532-0626.

BACH, F. R.; JORDAN, M. I. Kernel independent component analysis. **J. Mach. Learn. Res.**, JMLR.org, v. 3, p. 1–48, mar. 2003. ISSN 1532-4435.

BALAJI, P. Opencl: the open computing language. In: ____. **Programming Models for Parallel Computing**. MIT Press, 2015. p. 488–. ISBN 9780262332248. Available from Internet: <https://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7352782>.

BHUYAN, L. N.; YANG, Q.; AGRAWAL, D. P. Performance of multiprocessor interconnection networks. **Computer**, v. 22, n. 2, p. 25–37, Feb 1989. ISSN 0018-9162.

BLAKE, G.; DRESLINSKI, R. G.; MUDGE, T. A survey of multicore processors. **IEEE Signal Processing Magazine**, v. 26, n. 6, p. 26–37, November 2009. ISSN 1053-5888.

BOILLOT, L. et al. Task-based programming for seismic imaging: Preliminary results. In: **High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on**. [S.l.: s.n.], 2014. p. 1259–1266.

BOITO, F. Z. et al. Automatic I/O scheduling algorithm selection for parallel file systems. **Concurrency and Computation: Practice and Experience**, v. 28, n. 8, p. 2457–2472, 2016. ISSN 1532-0634. Cpe.3606.

BORDEAUX, U. de; CNRS; INRIA. **StarPU Handbook**. 2014. <http://starpu.gforge.inria.fr/doc/starpu.pdf>.

BOSILCA, G. et al. Parsec: Exploiting heterogeneity to enhance scalability. **Computing in Science Engineering**, v. 15, n. 6, p. 36–45, Nov 2013. ISSN 1521-9615.

BOSILCA, G. et al. Scalable dense linear algebra on heterogeneous hardware. **Advances in Parallel Computing**, 2013.

BREUER, A.; HEINECKE, A.; BADER, M. Petascale local time stepping for the ADER-DG finite element method. In: **2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016**. [S.l.: s.n.], 2016. p. 854–863.

BUCHTY, R. et al. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. **Concurrency and Computation: Practice and Experience**, John Wiley & Sons, Ltd, v. 24, n. 7, p. 663–675, 2012. ISSN 1532-0634.

CALANDRA, H. et al. Evaluation of successive CPUs/APUs/GPUs based on an OpenCL finite difference stencil. In: **Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on**. [S.l.: s.n.], 2013. p. 405–409.

CANTIELLO, P.; MARTINO, B. D.; MOSCATO, F. Compilers, techniques, and tools for supporting programming heterogeneous many multicore systems. In: ____. **Large Scale Network-Centric Distributed Systems**. Wiley-IEEE Press, 2014. p. 31–51. ISBN 9781118640708. Available from Internet: <https://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6674271>.

CASTRO, M. et al. Energy efficient seismic wave propagation simulation on a low-power manycore processor. In: **Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on**. [S.l.: s.n.], 2014. p. 57–64.

CASTRO, M.; GÓES, L. F. W.; MÉHAUT, J.-F. Adaptive thread mapping strategies for transactional memory applications. **Journal of Parallel and Distributed Computing**, v. 74, n. 9, p. 2845 – 2859, 2014. ISSN 0743-7315.

CHRISTEN, M.; SCHENK, O.; BURKHART, H. Automatic code generation and tuning for stencil kernels on modern shared memory architectures. **Comput. Sci.**, Springer-Verlag New York, Inc., v. 26, n. 3-4, p. 205–210, jun. 2011.

CHRISTEN, M.; SCHENK, O.; CUI, Y. Patus for convenient high-performance stencils: evaluation in earthquake simulations. In: **Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis**. [S.l.]: IEEE Computer Society Press, 2012. (SC '12), p. 11:1–11:10.

CLARKE, L.; GLENDINNING, I.; HEMPEL, R. The mpi message passing interface standard. In: DECKER, K. M.; REHMANN, R. M. (Ed.). **Programming Environments for Massively Parallel Distributed Systems**. Basel: Birkhäuser Basel, 1994. p. 213–218. ISBN 978-3-0348-8534-8.

CORTES, C.; VAPNIK, V. Support-vector networks. **Machine Learning**, v. 20, n. 3, p. 273–297, 1995. ISSN 1573-0565.

CRONSIOE, J.; VIDEAU, B.; MARANGOZOVA-MARTIN, V. Boast: Bringing optimization through automatic source-to-source transformations. In: **Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on**. [S.l.: s.n.], 2013. p. 129–134.

CRUZ, R. de la; ARAYA-POLO, M. Towards a multi-level cache performance model for 3d stencil computation. **Procedia Computer Science**, v. 4, p. 2146 – 2155, 2011. ISSN 1877-0509.

CRUZ, R. de la; ARAYA-POLO, M. Modeling stencil computations on modern hpc architectures. In: ____. **High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers**. Cham: Springer International Publishing, 2015. p. 149–171. ISBN 978-3-319-17248-4.

DAGUM, L.; MENON, R. Openmp: An industry-standard api for shared-memory programming. **IEEE Comput. Sci. Eng.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 5, n. 1, p. 46–55, jan. 1998. ISSN 1070-9924.

DATTA, K. et al. Optimization and performance modeling of stencil computations on modern microprocessors. **SIAM Rev.**, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, v. 51, n. 1, p. 129–159, feb. 2009. ISSN 0036-1445.

DATTA, K. et al. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: **Proceedings of the 2008 ACM/IEEE Conference on Supercomputing**. Piscataway, NJ, USA: IEEE Press, 2008. (SC '08), p. 4:1–4:12. ISBN 978-1-4244-2835-9.

DATTA, K. et al. Auto-tuning stencil computations on multicore and accelerators. In: ____. **Scientific Computing with Multicore and Accelerators**. Taylor & Francis Group: CRC Press, 2010.

DRUCKER, H. et al. Support vector regression machines. In: **Advances in Neural Information Processing Systems 9**. [S.l.]: MIT Press, 1997. p. 155–161.

DUPROS, F. et al. Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. In: **Proceedings of the 11th IEEE CSE'08, International Conference on Computational Science and Engineering**. São Paulo, Brazil: [s.n.], 2008. p. 253–260.

DUPROS, F. et al. Communication-avoiding seismic numerical kernels on multicore processors. In: **High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on**. [S.l.: s.n.], 2015. p. 330–335.

DUPROS, F.; DO, H.; AOCHI, H. On scalability issues of the elastodynamics equations on multicore platforms. In: **Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013**. [S.l.: s.n.], 2013. p. 1226–1234.

DURAN, A. et al. Ompss: A proposal for programming heterogeneous multi-core architectures. **Parallel Processing Letters**, v. 21, n. 02, p. 173–193, 2011.

DURSUN, H. et al. A Multilevel Parallelization Framework for High-Order Stencil Computations. In: **Euro-Par 2009 Parallel Processing, 15th International Euro-Par Conference**. Delft, The Netherlands: [s.n.], 2009. p. 642–653.

ELANGOVAN, V. K.; BADIA, R. M.; PARRA, E. A. Ompss-opencl programming model for heterogeneous systems. In: ____. **Languages and Compilers for Parallel Computing: 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. p. 96–111. ISBN 978-3-642-37658-0.

EOM, H. et al. Machine learning-based runtime scheduler for mobile offloading framework. In: **Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on**. [S.l.: s.n.], 2013. p. 17–25.

FERNÁNDEZ, A. et al. Task-based programming with ompss and its application. In: ____. **Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II**. Cham: Springer International Publishing, 2014. p. 601–612. ISBN 978-3-319-14313-2.

GAN, L. et al. Scaling and analyzing the stencil performance on multi-core and many-core architectures. In: **2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)**. [S.l.: s.n.], 2014. p. 103–110. ISSN 1521-9097.

GANAPATHI, A. et al. A case for machine learning to optimize multicore performance. In: **Proceedings of the First USENIX Conference on Hot Topics in Parallelism**. Berkeley, CA, USA: USENIX Association, 2009. (HotPar'09), p. 1–1.

GANAPATHI, A. S. **Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning**. Thesis (PhD) — EECS Department, University of California, Berkeley, Dec 2009.

GAUTIER, T. et al. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: **27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013**. [S.l.: s.n.], 2013. p. 1299–1308.

GONZALEZ, R. C.; WOODS, R. E. **Digital Image Processing (2nd Edition)**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2002. ISBN 9780201180756.

GREEN500. **Green 500 Supercomputer Sites**. 2017. <http://www.green500.org>.

HARRIS, M. **Maxwell: The Most Advanced CUDA GPU Ever Made**. 2014. <https://devblogs.nvidia.com/maxwell-most-advanced-cuda-gpu-ever-made/>.

HASSEN, S. B.; BAL, H. E.; JACOBS, C. J. H. A task- and data-parallel programming language based on shared objects. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 20, n. 6, p. 1131–1170, nov. 1998. ISSN 0164-0925. Available from Internet: <http://doi.acm.org/10.1145/295656.295658>.

HWU, W. W. et al. Compiler technology for future microprocessors. **Proceedings of the IEEE**, v. 83, n. 12, p. 1625–1640, Dec 1995. ISSN 0018-9219.

INC., T. K. G. **OpenCL Reference Pages**. 2018. <http://man.opencl.org/>. Accessed: 2018-01-01.

INTEL. **OpenMP* Loop Scheduling**. 2014. <https://software.intel.com/en-us/articles/openmp-loop-scheduling>. [Online; accessed 11-jan-2018].

INTEL. **OpenMP Loop Scheduling**. 2016. <https://software.intel.com/en-us/articles/openmp-loop-scheduling>. Accessed: 2016-01-01.

INTEL. **Intel® C++ Compiler 17.0 Developer Guide and Reference**. 2018. <https://software.intel.com/sites/default/files/managed/08/ac/PDF_CPP_Compiler_UG_17_0.pdf>. [Online; accessed 1-march-2018].

JAIN, A. K.; DUIN, R. P. W.; MAO, J. Statistical pattern recognition: a review. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 22, n. 1, p. 4–37, Jan 2000. ISSN 0162-8828.

KAMIL, S. et al. An auto-tuning framework for parallel multicore stencil computations. In: **Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on**. [S.l.: s.n.], 2010. p. 1–12. ISSN 1530-2075.

KRAKIWSKY, S. E.; TURNER, L. E.; OKONIEWSKI, M. M. Graphics processor unit (GPU) acceleration of finite-difference time-domain (FDTD) algorithm. In: **ISCAS (5)**. [S.l.: s.n.], 2004. p. 265–268.

KRISHNAN, S. P. T.; VEERAVALLI, B. Performance characterization and evaluation of hpc algorithms on dissimilar multicore architectures. In: **2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)**. [S.l.: s.n.], 2014. p. 1288–1295.

KUMAR, M. S.; BALAMURUGAN, B. A review on performance evaluation techniques in cloud. In: **2017 Second International Conference on Recent Trends and Challenges in Computational Models (ICRTCCM)**. [S.l.: s.n.], 2017. p. 19–24.

LACOSTE, X. et al. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In: **2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014**. [S.l.: s.n.], 2014. p. 29–38.

LI, Y. et al. Capes: Unsupervised storage performance tuning using neural network-based deep reinforcement learning. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2017. (SC ’17), p. 42:1–42:14. ISBN 978-1-4503-5114-0. Available from Internet: <http://doi.acm.org/10.1145/3126908.3126951>.

LINDHOLM, E. et al. Nvidia tesla: A unified graphics and computing architecture. **IEEE Micro**, v. 28, n. 2, p. 39–55, March 2008. ISSN 0272-1732.

MACHADO, R. S. et al. Comparing performance of c compilers optimizations on different multicore architectures. In: **2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)**. [S.l.: s.n.], 2017. p. 25–30.

MAGNI, A.; DUBACH, C.; O'BOYLE, M. Automatic optimization of thread-coarsening for graphics processors. In: **Proceedings of the 23rd International Conference on Parallel Architectures and Compilation**. New York, NY, USA: ACM, 2014. (PACT '14), p. 455–466. ISBN 978-1-4503-2809-8. Available from Internet: <http://doi.acm.org/10.1145/2628071.2628087>.

MALAS, T. M. et al. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. **SIAM J. Scientific Computing**, v. 37, n. 4, 2015.

MARTINEZ, V. et al. Stencil-based applications tuning for multi-core. In: **Latin American High Performance Computing Conference (CARLA 2016)**. [S.l.: s.n.], 2016. p. 1–15. Oral presentation.

MARTINEZ, V. et al. Performance improvement of stencil computations for multi-core architectures based on machine learning. **Procedia Computer Science**, v. 108, p. 305 – 314, 2017. ISSN 1877-0509. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

MARTINEZ, V. et al. Task-based programming on low-power nvidia jetson tk1 manycore architecture: Application to earthquake modeling. In: **Latin American High Performance Computing Conference (CARLA 2015)**. [S.l.: s.n.], 2015. p. 1–11. Oral presentation.

MARTINEZ, V. et al. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In: **Computer Architecture and High Performance Computing (SBAC-PAD), 2015 27th International Symposium on**. [S.l.: s.n.], 2015. p. 1–8. ISSN 1550-6533.

MARTINS, L. de O. et al. Accelerating curvature estimate in 3d seismic data using GPGPU. In: **26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014**. [S.l.: s.n.], 2014. p. 105–111.

MERCERAT, D.; GUILLOT, L.; VILOTTE, J.-P. Application of the Parareal Algorithm for Acoustic Wave Propagation. In: Simos, T. E.; Psihoyios, G.; Tsitouras, C. (Ed.). **American Institute of Physics Conference Series**. [S.l.: s.n.], 2009. (American Institute of Physics Conference Series, v. 1168), p. 1521–1524.

MEYER, D. et al. **e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien**. [S.l.], 2015. R package version 1.6-7.

MICHÉA, D.; KOMATITSCH, D. Accelerating a 3D finite-difference wave propagation code using GPU graphics cards. **Geophysical Journal International**, Blackwell Publishing Ltd, v. 182, n. 1, p. 389–402, 2010.

MICIKEVICIUS, P. 3D finite-difference computation on GPUs using CUDA. In: **Workshop on General Purpose Processing on Graphics Processing Units**. Washington, USA: ACM, 2009. p. 79–84.

MIJAKOVIC, R.; FIRBACH, M.; GERNDT, M. An architecture for flexible auto-tuning: The periscope tuning framework 2.0. In: **International Conference on Green High Performance Computing (ICGHPC)**. [S.l.: s.n.], 2016. p. 1–9.

MITTAL, S. A survey of techniques for architecting and managing asymmetric multicore processors. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 48, n. 3, p. 45:1–45:38, feb. 2016. ISSN 0360-0300.

MITTAL, S.; VETTER, J. S. A survey of cpu-gpu heterogeneous computing techniques. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 47, n. 4, p. 69:1–69:35, jul. 2015. ISSN 0360-0300.

MOCZO, P.; ROBERTSSON, J.; EISNER, L. The finite-difference time-domain method for modeling of seismic wave propagation. In: **Advances in Wave Propagation in Heterogeneous Media**. [S.l.]: Elsevier - Academic Press, 2007, (Advances in Geophysics, v. 48). chp. 8, p. 421–516.

MUCCI, P. J. et al. Papi: A portable interface to hardware performance counters. In: **In Proceedings of the Department of Defense HPCMP Users Group Conference**. [S.l.: s.n.], 1999. p. 7–10.

MULLER, K. R. et al. An introduction to kernel-based learning algorithms. **IEEE Transactions on Neural Networks**, v. 12, n. 2, p. 181–201, Mar 2001. ISSN 1045-9227.

NGUYEN, A. et al. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In: **2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. [S.l.: s.n.], 2010. p. 1–13. ISSN 2167-4329.

NVIDIA. **Whitepaper. NVIDIA's Next Generation CUDA™Compute Architecture: Fermi™**. 2009. <http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf>.

NVIDIA. **NVIDIA: Jetson TK1 Development Kit (Specification)**. 2014. <http://developer.download.nvidia.com/embedded/jetson/TK1/docs/3_HWDesignDev/JTK1_DevKit_Specification.pdf>.

NVIDIA. **Whitepaper. NVIDIA's Next Generation CUDA™Compute Architecture: Kepler™GK110/210**. 2014. <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf>.

NVIDIA. **CUDA Toolkit Documentation**. 2016. <http://docs.nvidia.com/cuda/index.html>. Accessed: 2016-01-01.

NVIDIA. **Whitepaper. NVIDIA Tesla P100. The Most Advanced Data-center Accelerator Ever Built. Featuring Pascal GP100, the World's Fastest GPU**. 2017. <http://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper-v1.2.pdf>.

PATERSON, D. **The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges**. 2009. <http://www.nvidia.com.br/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf>.

PEREZ, J. M.; BADIA, R. M.; LABARTA, J. A dependency-aware task-based programming environment for multi-core architectures. In: **2008 IEEE International Conference on Cluster Computing**. [S.l.: s.n.], 2008. p. 142–151. ISSN 1552-5244.

PEREZ, J. M.; BADIA, R. M.; LABARTA, J. Handling task dependencies under strided and aliased references. In: **Proceedings of the 24th ACM International Conference on Supercomputing**. New York, NY, USA: ACM, 2010. (ICS '10), p. 263–274. ISBN 978-1-4503-0018-6.

PUSUKURI, K. K.; GUPTA, R.; BHUYAN, L. N. Adapt: A framework for coscheduling multithreaded programs. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 9, n. 4, p. 45:1–45:24, jan. 2013. ISSN 1544-3566.

RAHMAN, S. M. F.; YI, Q.; QASEM, A. Understanding stencil code performance on multicore architectures. In: **Proceedings of the 8th ACM International Conference on Computing Frontiers**. New York, NY, USA: ACM, 2011. (CF '11), p. 30:1–30:10. ISBN 978-1-4503-0698-0. Available from Internet: <http://doi.acm.org/10.1145/2016604.2016641>.

RAI, J. K. et al. On prediction accuracy of machine learning algorithms for characterizing shared l2 cache behavior of programs on multicore processors. In: **Computational Intelligence, Communication Systems and Networks, 2009. CICSYN '09. First International Conference on**. [S.l.: s.n.], 2009. p. 213–219.

ROSALES, C. et al. Performance prediction of hpc applications on intel processors. In: **2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)**. [S.l.: s.n.], 2017. p. 1325–1332.

ROTEN, D. et al. High-frequency nonlinear earthquake simulations on petascale heterogeneous supercomputers. In: **Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016**. [S.l.: s.n.], 2016. p. 957–968.

SAMUEL, A. L. Some studies in machine learning using the game of checkers. **IBM Journal of Research and Development**, v. 3, n. 3, p. 210–229, July 1959. ISSN 0018-8646.

SANTANDER, U. I. de. **Super Computación y Cálculo Científico UIS**. 2015. <http://www.sc3.uis.edu.co/servicios/hardware/>.

SAXENA, G.; JIMACK, P. K.; WALKLEY, M. A. A cache-aware approach to domain decomposition for stencil-based codes. In: **2016 International Conference on High Performance Computing Simulation (HPCS)**. [S.l.: s.n.], 2016. p. 875–885.

SERPA, M. S. et al. Strategies to improve the performance of a geophysics model for different manycore systems. In: **2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)**. [S.l.: s.n.], 2017. p. 49–54.

SHUKLA, S. K.; MURTHY, C.; CHANDE, P. K. A survey of approaches used in parallel architectures and multi-core processors, for performance improvement. In: ____. **Progress in Systems Engineering: Proceedings of the Twenty-Third International

Conference on Systems Engineering. Cham: Springer International Publishing, 2015. p. 537–545. ISBN 978-3-319-08422-0.

SODANI, A. et al. Knights landing: Second-generation intel xeon phi product. **IEEE Micro**, v. 36, n. 2, p. 34–46, Mar 2016. ISSN 0272-1732.

SPAZIER, J.; CHRISTGAU, S.; SCHNOR, B. Efficient parallelization of matlab stencil applications for multi-core clusters. In: **2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)**. [S.l.: s.n.], 2016. p. 20–29.

STALLMAN, R. M.; COMMUNITY the G. D. **Using the GNU Compiler Collection. For GCC version 6.4.0**. 2017. <https://gcc.gnu.org/onlinedocs/gcc-6.4.0/gcc.pdf>. [Online; accessed 1-dec-2017].

STENGEL, H. et al. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In: **Proceedings of the 29th ACM on International Conference on Supercomputing**. New York, NY, USA: ACM, 2015. (ICS ’15), p. 207–216. ISBN 978-1-4503-3559-1. Available from Internet: <http://doi.acm.org/10.1145/2751205.2751240>.

STOJANOVIC, S. et al. An overview of selected hybrid and reconfigurable architectures. In: **Industrial Technology (ICIT), 2012 IEEE International Conference on**. [S.l.: s.n.], 2012. p. 444–449.

SUKHIJA, N. et al. Portfolio-based selection of robust dynamic loop scheduling algorithms using machine learning. In: **Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International**. [S.l.: s.n.], 2014. p. 1638–1647.

TANG, Y. et al. The pochoir stencil compiler. In: **ACM Symposium on Parallelism in Algorithms and Architectures**. New York, NY, USA: ACM, 2011. (SPAA ’11), p. 117–128. ISBN 978-1-4503-0743-7.

TOP500. **Top500 Supercomputer Sites**. 2017. <http://www.top500.org>.

TSUBOI, S. et al. A 1.8 trillion degrees-of-freedom, 1.24 petaflops global seismic wave simulation on the K computer. **IJHPCA**, v. 30, n. 4, p. 411–422, 2016.

VAPNIK, V. N. An overview of statistical learning theory. **IEEE Transactions on Neural Networks**, v. 10, n. 5, p. 988–999, Sep 1999. ISSN 1045-9227.

VASUDEVAN, R.; VADHIYAR, S. S.; KALÉ, L. V. G-charm: an adaptive runtime system for message-driven parallel applications on hybrid systems. In: **ICS 2013 - Proceedings of the 2013 ACM International Conference on Supercomputing**. [S.l.: s.n.], 2013. p. 349–358.

VIDEAU, B. et al. Boast. **Int. J. High Perform. Comput. Appl.**, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 32, n. 1, p. 28–44, jan. 2018. ISSN 1094-3420. Available from Internet: <https://doi.org/10.1177/1094342017718068>.

VILELA, R. F. **Perfilagem do Problema de Resolução da Equação da Onda por Diferenças Finitas em Coprocessador Xeon Phi**. http://www.pee.ufrj.br/index.php/pt/producao-academica/dissertacoes-de-mestrado/2017/2016033170–53/file: [s.n.], 2017. Dissertação (mestrado).

VIRIEUX, J. P-SV wave propagation in heterogeneous media; velocity-stress finite-difference method. **Geophysics**, v. 51, n. 4, p. 889–901, 1986.

VLADUIC, D.; CERNIVEC, A.; SLIVNIK, B. Improving job scheduling in grid environments with use of simple machine learning methods. In: **International Conference on Information Technology: New Generations**. [S.l.: s.n.], 2009. p. 177–182.

WANG, Z.; O'BOYLE, M. F. P. Mapping parallelism to multi-cores: A machine learning based approach. In: **Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. New York, NY, USA: ACM, 2009. (PPoPP '09), p. 75–84. ISBN 978-1-60558-397-6.

WENG, L.; LIU, C.; GAUDIOT, J.-L. Scheduling optimization in multicore multithreaded microprocessors through dynamic modeling. In: **Proceedings of the ACM International Conference on Computing Frontiers**. New York, NY, USA: ACM, 2013. (CF '13), p. 5:1–5:10. ISBN 978-1-4503-2053-5.

WILLIAMS, S.; WATERMAN, A.; PATTERSON, D. Roofline: An insightful visual performance model for multicore architectures. **Commun. ACM**, ACM, New York, NY, USA, v. 52, n. 4, p. 65–76, abr. 2009. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/1498765.1498785>.

XU, R.; WUNSCH, D. Survey of clustering algorithms. **IEEE Transactions on Neural Networks**, v. 16, n. 3, p. 645–678, May 2005. ISSN 1045-9227.

ZHU, X. et al. Analyzing mpi-3.0 process-level shared memory: A case study with stencil computations. In: **2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing**. [S.l.: s.n.], 2015. p. 1099–1106.