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

ALÉCIO PEDRO DELAZARI BINOTTO

A Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms

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

Prof. Dr. Carlos Eduardo Pereira
Supervisor

Prof. Dr. Dieter W. Fellner
Cosupervisor

Porto Alegre, September 2011

CIP – CATALOGING-IN-PUBLICATION

Binotto, Alécio Pedro Delazari

A Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms / Alécio Pedro Delazari Binotto. – Porto Alegre: PPGC da UFRGS, 2011.

129 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2011. Supervisor: Carlos Eduardo Pereira; Cosupervisor: Dieter W. Fellner.

1. High-performance computing. 2. Scheduling. 3. Dynamic load-balancing. 4. Heterogenous systems. 5. Graphics processors. 6. Solvers for systems of linear equations. I. Pereira, Carlos Eduardo. II. Fellner, Dieter W.. III. Título.

To my parents: Mercedes and Jurandyr

ACKNOWLEDGEMENTS

The development of this binational doctoral thesis was composed of studies at the Federal University of Rio Grande do Sul, in Brazil, along 2 years and 2 months, and of studies at the Technische Universität Darmstadt and the Fraunhofer Institute for Visual Computing Research, in Germany, along 2 years and 10 months. It has been a significant research and personal challenge and it is one of the most important steps on my career. To reach this goal, a set of personal, technical, and financial support were needed, which without any of them I could not have developed this work.

I would like to begin with my parents, Mercedes and Jurandyr. They were the personal support to reach motivation and equilibrium on the most difficult moments. They have encouraged my studies from the very beginning. I express here my deepest respect, proud, and gratitude for their dedication and the many years of constant support, care, and friendship. This is for you. Without you, I would have never reached this way and the courage of exploring new frontiers. This gratitude is also extended to all of my family.

I am also grateful to Maria Clara for her encouragement on the beginning of this journey, specially on deciding to go to Germany, and also for the last year.

The other pillar is my main advisor: Prof. Carlos Eduardo Pereira. Carlos believed on my potential, accepting me as a PhD student. There is no bad time for Carlos and he became a friend spontaneously. He has the ability to discuss and to propose several ideas to constructively improve the work, inserting me on a variety of multidisciplinary research projects (I am sure this is a plus on my career). He was the one that believed and advocated for my applied (and not just academic) doctoral thesis. With Carlos, I have learned a lot, mainly to not concentrate just in one point/topic, but to open the eyes to complementary and innovative stuff, combining engineering and informatics. He was also decisive on acquiring the binational degree.

I had also the same feeling with Prof. Dieter Fellner. In our first meeting, he presented me several other ways to go and to question the work regarding the usability of the thesis' result. I will never forget when he challenged that the solutions have to serve for a long period of time and not just till the end of the PhD course. Additionally, I would like to thank Prof. André Stork and Gino Brunetti, whom accepted me on my stay in Germany. André offered me several possibilities of industry projets to experimentally validate the developed methods and put them in practice. This makes a difference at the end, André.

I wish to thank Dr. Arjan Kuijper, who I learned a lot the way of writing scientific texts and to emphasize the importance of the research. Acquiring skills of technical writing helped to clearly identify the contributions of this research for conference and journal committees, leading to constructive external evaluation and scientific publications. He also helped on finding shortcuts for the binational degree.

I also thank Daniel Weber, who I shared the office room during my stay in Germany.

He was important on choosing a case study for this work and to discuss mainly mathematical stuff I should learn. Without his support, I would have taken more time to understand concepts of linear algebra. At the same time, I thank my students Christian Daniel and Bernardo Pedras for the dedication on implementing some of the ideas I had in mind. Together, we have published interesting ideas. Besides, I would like to thank Profa. Judith Kelner, Prof. Cláudio Geyer, and Prof. Philippe Navaux for the considerations made at the thesis qualification. I also thank all my former colleagues at the Fraunhofer IGD-A2 group and Gabriele Knöβ for the administrative support.

The last pillar, but not less important, was the funding I received during this period. I thank CETA, where I worked on the beginning and enabled my first contact with Fraunhofer Institutes, and mainly DAAD - Deutsche Akademischer Austausch Dienst, the Programme Alβan and Fraunhofer IGD for the scholarships during my stay in Germany. In addition, I would like to thank Prof. Valter Roesler and Carlos for the scholarship on the way back to Brazil to conclude the thesis.

I want to say a big "thank you" to all of my friends! I should not forget to mention Angela Seip-Butz, who was a very important friend that I will definitely keep over the time. She was a lovely support for bad and good times and also to introduce typical things of Germany. If I know the real Germany, I know by her hints. In addition, I thank Roberto, Rafael, Jerome, Gilberto, Ulysses, Jorge, Gonçalo, Valéria, Régis, and Frithjof for important moments.

Finally, my gratitude to God for keeping me every time healthy during this journey.

CONTENTS

ACRONYMS

ARBB Array Building Blocks

API Application Program Interface

ASIC Application Specific Integrated Circuit

CFD Computational Fluid Dynamics

CG Conjugate Gradient

CPU Central Processing Unit

Ct C for Throughput Computing

CTM Close To Metal

CUDA Compute Unified Device Architecture

FAP First Assignment Phase

FIFO First In First Out

FPGA Field-Programmable Gate Array

GPGPU General Purpose Computation Using Graphics Hardware

GPU Graphics Processing Unit

HT Hyper Threading

MIMD Multiple Instruction Multiple Data

NFR Non-Functional Requirements

OpenCL Open Computing Language

PC Personal Computer

PCI Peripheral Component Interconnect

PTX Parallel Thread Execution

PU Processing Unit

RAP Runtime Assignment Phase

SDK Software Development Kit

SIMD Single Instruction Multiple Data

SLE System of Linear Equations

TBB Threading Building Blocks

UA Unit of Allocation

UML Unified Modeling Language

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ABSTRACT

A modern personal computer can be now considered as a one-node heterogeneous cluster that simultaneously processes several applications' tasks. It can be composed by asymmetric Processing Units (PUs), like the multi-core Central Processing Unit (CPU), the many-core Graphics Processing Units (GPUs) - which have become one of the main co-processors that contributed towards high performance computing - and other PUs. This way, a powerful heterogeneous execution platform is built on a desktop for data intensive calculations. In the perspective of this thesis, to improve the performance of applications and explore such heterogeneity, a workload distribution over the PUs plays a key role in such systems. This issue presents challenges since the execution cost of a task at a PU is non-deterministic and can be affected by a number of parameters not known a priori, like the *problem size domain* and the *precision* of the solution, among others.

Within this scope, this doctoral research introduces a *context-aware runtime and performance tuning system* based on a compromise between reducing the execution time of the applications - due to appropriate dynamic scheduling of high-level tasks - and the cost of computing such scheduling applied on a platform composed of CPU and GPUs. This approach combines a model for a *first scheduling* based on an off-line task performance profile benchmark with a *runtime model* that keeps track of the tasks' real execution time and efficiently schedules new instances of the high-level tasks dynamically over the CPU/GPU execution platform. For that, it is proposed a set of heuristics to schedule tasks over one CPU and one GPU and a generic and efficient scheduling strategy that considers several processing units.

The proposed approach is applied in a case study using a CPU-GPU execution platform for computing iterative solvers for Systems of Linear Equations using a stencil code specially designed to explore the characteristics of modern GPUs. The solution uses the number of unknowns as the main parameter for assignment decision. By scheduling tasks to the CPU and to the GPU, it is achieved a performance gain of 21.77% in comparison to the static assignment of all tasks to the GPU (which is done by current programming models, such as OpenCL and CUDA for Nvidia) with a scheduling error of only 0.25% compared to exhaustive search.

Keywords: High-performance computing, Scheduling, Dynamic load-balancing, Heterogenous systems, Graphics processors, Solvers for systems of linear equations.

Um Sistema de Escalonamento Dinâmico e Tuning em Tempo de Execução para Plataformas Desktop Heterogêneas de Múltiplos Núcleos

RESUMO

Atualmente, o computador pessoal (PC) moderno poder ser considerado como um *cluster* heterogênedo de um nodo, o qual processa simultâneamente inúmeras tarefas provenientes das aplicações. O PC pode ser composto por Unidades de Processamento (PUs) assimétricas, como a Unidade Central de Processamento (CPU), composta de múltiplos núcleos, a Unidade de Processamento Gráfico (GPU), composta por inúmeros núcleos e que tem sido um dos principais co-processadores que contribuiram para a computação de alto desempenho em PCs, entre outras. Neste sentido, uma plataforma de execução heterogênea é formada em um PC para efetuar cálculos intensivos em um grande número de dados. Na perspectiva desta tese, a distribuição da carga de trabalho de uma aplicação nas PUs é um fator importante para melhorar o desempenho das aplicações e explorar tal heterogeneidade. Esta questão apresenta desafios uma vez que o custo de execução de uma tarefa de alto nível em uma PU é não-determinístico e pode ser afetado por uma série de parâmetros não conhecidos a priori, como o *tamanho do domínio do problema* e a *precisão* da solução, entre outros.

Nesse escopo, esta pesquisa de doutorado apresenta um *sistema sensível ao contexto e de adaptação em tempo de execução* com base em um compromisso entre a redução do tempo de execução das aplicações - devido a um escalonamento dinâmico adequado de tarefas de alto nível - e o custo de computação do próprio escalonamento aplicados em uma plataforma composta de CPU e GPU. Esta abordagem combina um modelo para um *primeiro escalonamento* baseado em perfis de desempenho adquiridos em pré-processamento com um *modelo online*, o qual mantém o controle do tempo de execução real de novas tarefas e escalona dinâmicamente e de modo eficaz novas instâncias das tarefas de alto nível em uma plataforma de execução composta de CPU e de GPU. Para isso, é proposto um conjunto de heurísticas para escalonar tarefas em uma CPU e uma GPU e uma estratégia genérica e eficiente de escalonamento que considera várias unidades de processamento.

A abordagem proposta é aplicada em um estudo de caso utilizando uma plataforma de execução composta por CPU e GPU para computação de métodos iterativos focados na solução de Sistemas de Equações Lineares que se utilizam de um cálculo de *stencil* especialmente concebido para explorar as características das GPUs modernas. A solução utiliza o número de incógnitas como o principal parâmetro para a decisão de escalonamento. Ao escalonar tarefas para a CPU e para a GPU, um ganho de 21,77% em desempenho é obtido em comparação com o escalonamento estático de todas as tarefas para a GPU (o qual é utilizado por modelos de programação atuais, como OpenCL e CUDA para Nvidia) com um erro de escalonamento de apenas 0,25% em relação à combinação exaustiva.

Palavras-chave: Computação de alto desempenho, Escalonamento, Balanceamento de carga dinâmico, Sistemas heterogêneos, Processadores gráficos, Métodos para solução de sistemas de equações lineares.

1 INTRODUCTION

Modern industrial applications of *virtual engineering* commonly require high performance platforms to deal with distinct algorithms and massive calculations. Many of these applications - composed of scientific and engineering algorithms - require a powerful execution platform to perform simulation tasks with a minimum quality of performance. Hence, high performance platforms are a requisite for dealing appropriately with timing constraints towards the achievement of a *real-time* simulation for virtual engineering.

These requirements strongly indicate that a parallel architecture is a necessary approach to accelerate the computational time of such applications. Moreover, in order to obtain an additional performance gain, such systems could be parallel-based designed to benefit also from processing distribution and achieve their final goal with performance maximization.

Until the recent past, grid and cluster computing as well as conventional supercomputers were the options used as powerful execution platforms to deal with most of the scientific and engineering applications. Nowadays, with the recent development of low-cost parallel and "plug-and-play" hardware, the community can profit from an interesting powerful execution platform, locally in a Personal Computer (PC). The many-core Graphics Processing Unit (GPU) is a good example. It is one of the most well known computing unit of this type. It evolved from specific application hardware for Computer Graphics to a multiprocessor architecture for general purposes computation (OWENS et al., 2007). Its computational power showed up along the time to be used beyond graphics utilities and now can be applied in favor of mathematical computations, physical simulations, scientific calculations, among other general processing. This trend is called General Purpose processing using GPU (GPGPU) (GPGPU, 2010).

As another example in the market, the Cell Processor followed a similar way of evolution as the GPU, evolving from a game purpose to compute tasks of generic purpose (KIM, 2008). It was firstly designed and commercialized as an embedded processor into game consoles and at present, based on user needs, it is also offered as an accelerator card (coupling eight heterogeneous processors) that can be plugged over a PCI Express bus.

The Field Programmable Gate Arrays (FPGA) is another example that has become attractive over the years due to its flexibility, even though it is not a common hardware like the GPU or the Cell Processor - it is devoted for more experienced users (CHE et al., 2008). The hardware itself is configurable and one can configure a specific FPGA with a processor (including multiple cores), memory, and logic cells. All these blocks are embedded in a general routing structure (also reconfigurable) which allows their inter-

connections.

From these examples, it can be verified that several alternatives for configuration of execution platforms and application programming can be stated, aiming at better application performance. The resulting execution platform can, then, be viewed as a heterogeneous multi-core architecture, combining some concepts of cluster and grid, but on a unique node of a common desktop or of portable computers. In other words, the combination of these multiprocessor units can be considered as an asymmetric set of Processing Units (PUs) and it is intensified with the new generation of multi-core Central Processing Units (CPUs), which, over the last few years, became indeed more powerful and have turned into processors of multiple cores. Due to their ability to deliver higher system performance more efficiently than single-core processors, the trend towards a higher number of cores is now established and a performance gain can be obtained using parallelization over the cores as well. Additionally, but not less important, all of these PUs are offered for a reasonable market price and as commodity, powering a PC to serve as a type of *supercomputer*.

In this sense, such low-cost hybrid hardware architectures (CPU, GPU, Cell, FPGA, etc) are becoming attractive to compose adaptable execution platforms in a single personal computer, being an alternative to the supercomputers or even clusters. Moreover, as crucial as the hardware evolution, software applications must, at the same time, evolve and benefit from that offer of powerfulness. However, it is a challenge to perform simple programming and perform efficient resource utilization in high level over the computing units in order to enable applications to move between different architectures and automatically scale as new processor generations are introduced (MCCOOL, 2008).

Some tools for parallelization based on threads already consider workload distribution over the multiple cores of a CPU. The same applies for the interfaces (drivers) for GPU or Cell, managing internal parallelization and giving certain flexibility and control to the programmer. On the other hand, an automatic coarser-grained parallelization focused on a high level design of processing distribution over the available PUs is lacking. The task of distributing the computation in high level is made currently at programming time by the designer without considering the entire context of the platform in a specific time. This way, runtime conditions are commonly not taken into consideration in the processing distribution. Below, some limitations are listed of current approaches:

- They are oriented to a specific hardware, i.e., applications are hard-coded using the specific API of the target PU, avoiding the flexibility of executing the same application on other PU without recoding and recompilation.
- They perform the load-balancing of tasks between different PUs in an off-line and statically pre-designed way. This means that the programmer codes which components of the application will execute in which hardware. If the execution platform configuration is changed or there is presence of load-imbalance, an application recoding is necessary.
- They do not take into account the runtime execution conditions of the platform, assuming that the PUs are completely idle without processing third party applications.
- They assume that the execution platform must be composed by the specific PU in which the application was developed, otherwise the application will not execute (even in a worst performance using the CPU, for example).

- They assume that the programmer must know specificities of each target PU architecture. For example, the programmer must manage the data over the memory levels (cache) explicitly using the programming language.

- They force the programmer to think not only about the main goal of his application, but also in low-level details (focused on the PU architecture) to balance the processing and extract a good performance from the application. Such a task considerably increases the programming complexity.

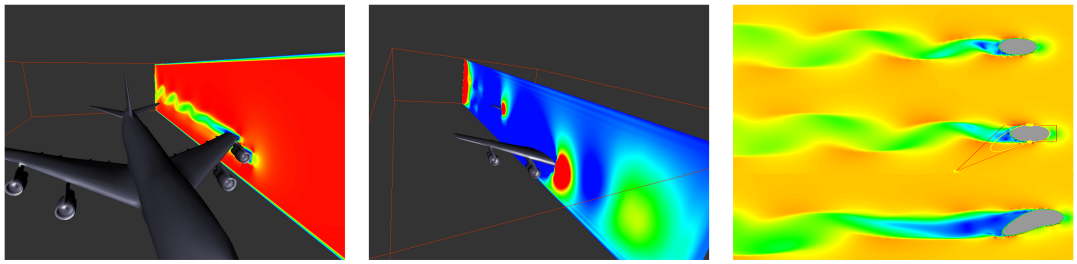
The presented items open possibilities towards the creation of new strategies that overcome these limitations, from a new programming layer, that encapsulates the APIs of the processing units, to automatic strategies that assign application's tasks to the most appropriated computing unit at a given runtime condition and aiming at performance maximization. This doctoral dissertation concentrates on the last approach: it introduces a framework that automatically tunes specific applications and performs a workload distribution of the applications (tasks, algorithms, or even full applications that must run concurrently) by choosing a processing unit as the execution target hardware in order to better meet applications' time requirements, such as performance.

Challenging applications that could benefit from the framework to obtain performance gain are simulation models of physical phenomena, which must precisely reflect the reality. For that, a high performance computing environment is decisive to accelerate numerical computations utilized on the applications, like on Computational Fluid Dynamics (CFD) (LUKSCH, 2000; WANG; YU; MA, 2010) used as case study in this thesis. This is an area of fluid mechanics highly used to simulate the fluid flows that are everywhere in our lives, like liquids and gases. It uses numerical methods that demand large computations for solving, for example, the velocity field and local pressure of the wind on objects like planes and cars. For those cases, a big computational effort is required for processing the complex, and sometimes recursive, mathematical models, clearly leading to the need of techniques that can optimize both computation time and performance.

The demand of improving the performance of a CFD application also emerged within the scope of an applied research project, focused on the industrial prototyping for a leading automobile industry in Germany. In CFD industrial prototyping, commonly default flow simulation is used, while in later stages of product development the models become more geometrically detailed and precise (LUKSCH, 2000). In the case of this project, by using a cluster of several CPUs for a traditional flow simulation, based on Navier-Stokes equations (FERZIGER; PERIC, 2002), the average calculation time takes about 12 hours[1]. At the same time, the error in accuracy between the actual aerodynamic behavior and the subsequent real prototype is around $5\ \%$. This means that the used CFD model brings precision to the engineers, who maybe could accept to trade precision over simulation time in early stages of product development.

Based on that scenario, the industry wants to potentially increase the flow simulation in terms of performance, inserting a new CFD phase on early stages of product development. This phase has the goal to produce a real-time flow simulation executed on the engineers' desktop, commonly composed of a CPU and a GPU. Such simulation could use a less accurate, but still precise enough, CFD method to achieve the real-time requirement

[1] Due to a non-disclosure agreement with the automobile company, details about the cluster configuration or CFD and car models cannot be described.



(a) (b) (c)

Figure 1.1: Real-time CFD Application: (a) velocity field and (b) pressure slice visualization of a 3D simulation; (c) a sequence of three time instances representing the velocity field visualization of a 2D simulation with real-time geometry modification

at the desktop. In this new phase, it is also desired that the engineer could interactively modify the geometry model to analyze its design over a virtual wind tunnel. This implies that the used models are not perfectly precise, but sufficiently accurate for early stages of product development. Additionally, it is desired that the engineer could simultaneously work with multiple simulations for visual comparison purposes.

This way, the main benefit of inserting this new phase is to reduce the number of times that precise CFD models (over accurate geometry models) need to be reevaluated in later stages of product development. Decreasing computation time without compromising the quality is clearly desired. For example, a 3D real-time CFD simulator with surface modification could be used to optimize a rough geometrical structure in early prototyping stages. As advantage, the number of times that accurate CFD models must be reevaluated in later stages of product development is expected to be significantly reduced, gaining time in the design process.

Based on these considerations, a 3D real-time CFD simulation was developed to be applied on the new proposed phase. The method is based on Stam (STAM, 1999), which presents an acceptable trade-off between accuracy and speed. In the method, it was identified that solving a System of Linear Equations (SLEs) used to compute the velocity and pressure, for example, was the most time expensive step on the workflow of the CFD method. Therewith, the performance of the solvers for SLEs were, then, improved in this thesis by using a new GPU-based strategy. Some examples of the developed CFD application are shown by Figure 1.1. Figure 1.1(a) depicts a slice of the developed 3D simulation that represents the velocity field visualization. Figure 1.1(b) shows the pressure visualization. And a real-time geometry modification is shown in Figure 1.1(c), where a 2D sequence of three images, i.e., three time instances represent the manipulation of an object's geometry.

But, the GPU-based solution was still static in terms of platform execution and configuration, and use the GPU or the CPU to compute the solvers. For real-time applications, efficiency with respect to both huge domain sizes and with small problems is important. Thus, a CPU-GPU platform dedicated to address these two different aspects is assumed to offer a better execution scenario than homogeneous ones.

Therefore, since this scenario does present a dynamic behavior (e.g., the domain size can vary, the execution platform is different over the desktops, the engineer can execute several simulations in parallel, etc), imposing thereby dynamic requirements to the ap-

plication, an static solution is no longer efficient. For such scenarios, the presence of strategies that dynamically adapt the application(s) to execute on a heterogeneous platform, like the CPU and the GPU with the requisite to obtain fast computational times using the resources in an efficient manner, is desirable. Following that line, dynamic and reconfigurable load-balancing computing (by means of a set of partitioning, allocation, and scheduling methods) is a potential paradigm for those scenarios. It can provide flexibility and improve efficiency, offering alternatives for programming an application on heterogeneous and multi-core architectures (FREITAS et al., 2008a).

1.1 Goals

As already mentioned, desktop-based co-processors, like many-core GPUs, are nowadays a cost-effective alternative for those execution platforms that aim at better performance. Taking an example, Nvidia has presented its GPU GTX285 that provides a peak performance of 1062 Gflop/s for single precision and 89 Gflop/s for double precision (NVIDIA, 2010b).

As a consequence, heterogeneous platforms with several types of processing units act in essence as powerful asymmetric multi-core clusters and can handle multiple applications and tasks, like CFD and the tasks of solvers for SLEs. This is even intensified with the multi-core CPUs, like the Intel Core2Quad that provides around 100 Gflop/s (INTEL, 2010a). Therefore, *efficiently using all available resources from the heterogeneous execution platform* is a significant challenge to program applications.

In this direction, this thesis has the goal to provide methodologies, strategies and mechanisms that aggregates allocation and scheduling capabilities to tasks that must be executed by heterogeneous systems. By this means, the applications can be dynamically configured over the asymmetric architecture in order to use the most appropriate computational resources that can currently diminish the tasks' execution time.

In order to benefit from the powerfulness of all PUs, a strategy to distribute the application tasks onto such processing units is designed. The strategy lies on *dynamic scheduling*, instead of current static programming and scheduling model used by OpenCL (STONE; GOHARA; SHI, 2010) or, more specifically, by CUDA (NVIDIA, 2010b) for Nvidia GPUs (see also the work of Göddeke et al. (GÖDDEKE et al., 2009)). This need becomes even more essential when dealing with desktop applications that present timing constraints, like the real-time CFD application that partially motivated this work.

1.2 Contributions

The topic "scheduling for hybrid multi-core platforms" has been identified as one important open problem by the recent ICT (Information and Communication Technologies) call for research proposals of the European Commission, named Framework Program 7 (COMMISSION, 2010), shortly FP7, and by the Roadmap on High-Performance Embedded Architecture and Compilation that drives the importance of several open problems in the area of computer science (DE BOSSCHERE et al., 2007). The references explicitly mention that the availability of multiple cores is a trend and will integrate up to 1000 billion devices by the year 2020. It indicates that these devices will provide orders of

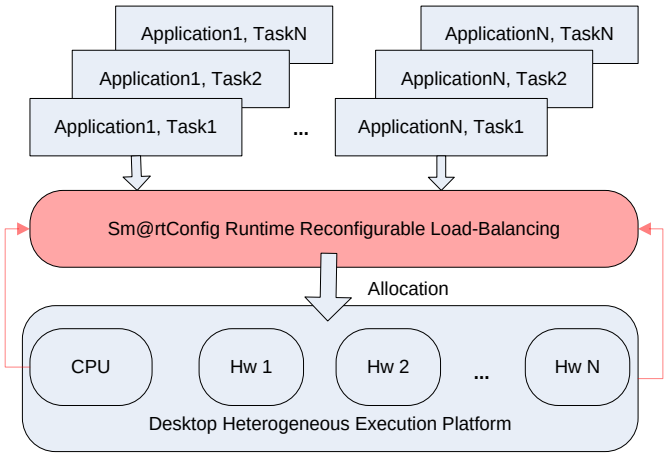


Figure 1.2: Sm@rtConfig system overview

magnitude for performance improvement only with much higher concurrency and with heterogeneous architectures tuned to specific application kernels. In addition, an analogous report made by the Council of Advisors on Science and Technology of the United States of America claims that performance gains due to improvements in algorithms have exceeded the performance gains due to increased processor speed (SCIENCE; TECHNOLOGY, 2010). The report also focuses on the need of system management tools for the next generation of high performance technologies, including research on hardware/-software systems and in both systems software and applications software. Based on these statements, this thesis addresses punctually the assignment methods over heterogeneous platforms, specially composed of CPU and GPU.

Figure C.1 gives an overview of the thesis contributions, where the proposed framework is called Sm@rtConfig. Coupling the discovery of computing unit resources of the platform with the applications' characteristics, an analysis is performed to configure the tasks allocation balance over the available processing units. During runtime, the Sm@rtConfig profiles the performance of tasks feeding the balancer towards a possible new allocation if this procedure can promote a better performance.

This framework presents a new strategy to distribute the workload over the CPU and the GPU, being sufficient generic to consider other PUs coupled in a desktop. The dynamic assignment methods combine a *first assignment phase* for a set of *high-level tasks* (algorithms, for example) with a *runtime phase* that obtains real performance measurements of tasks, feeding a *performance database*. The first assignment is based on a preprocessing benchmark for acquiring basic computational times' samples of the tasks on each PU. This way, after the first assignment, the system considers the history presented on the database to perform further assignments for every task, maximizing the applications' performance with load-balance and minimal overhead.

In summary, the main contributions of this thesis are:

1. The development of a framework that comprises: (i) a *first assignment phase* of tasks, (ii) a runtime *profiler* that feeds a *timing performance database*, and (iii) the *runtime assignment phase* that performs dynamic assignments based on the performance history;

2. The development of a new strategy for storing and retrieving data, used by the tasks of SLEs' solvers, on the GPU memory hierarchy aiming memory coalescing and using the shared memory, with a performance gain compared to state-of-the-art works;

3. The analysis of the solvers' characteristics and their performance on a CPU-GPU platform, expressing the conditions where the solvers (tasks) obtain better execution performance (i.e., finding a so-called performance *break-even point* that indicates the best PU to be used) for performance tuning purposes;

4. The implementation and comparison of three different iterative methods to solve SLEs (Jacobi, Gauss-Seidel, and Conjugate Gradient) on the CPU and multiple GPUs, applied to a real-time CFD simulation with a geometry modification example.

All main parts of this thesis have been published in several conferences, which demonstrate the recognition of this work by the research community. Moreover, it indicates the relevance of the investigations carried out in the scope of this research. An initial study of virtual engineering was performed by Binotto et al. in 2006 (BINOTTO et al., 2006), followed by a study that used the GPU to enhance the performance of engineering based simulations (BINOTTO et al., 2006). Then, the basic concepts of this thesis were further published in 2008 (FREITAS et al., 2008a; BINOTTO et al., 2008), refined by a journal (BINOTTO et al., 2009) and a conference (BINOTTO et al., 2009) publication in 2009, being also applied to other case studies rather than CFD (FREITAS et al., 2008b; BINOTTO et al., 2008; FREITAS et al., 2009).

The solvers for SLEs tasks, used by the CFD application, were presented by Binotto, Pereira, and Fellner in 2010 (BINOTTO; PEREIRA; FELLNER, 2010) and further detailed in the same year (BINOTTO et al., 2010). The works described the execution over the heterogeneous platform composed by a CPU and a GPU, indicating the importance of a tuning system based on performance break-even points and of an efficient method to implement the solvers specifically for the GPU architecture. It is important to note that understanding the most efficient design and utilization of emerging multi-core systems is one of the most challenging questions and a flexible platform composed of the CPU and the GPU of different capabilities can be benefited by performance tuning strategies for switching over the PUs.

Furthermore, some of the assignment algorithms available on the framework were published by the author in 2010 as well (BINOTTO et al., 2010). To finalize, an article about the complete framework description - with additional infrastructure to support the whole dynamic scheduling and performance tuning system of high-level tasks for virtual engineering - was recently invited for extended publication in the special issue on Advanced Software Engineering in Industrial Automation of the journal Control Engineering Practice to be published by Elsevier in early 2012.

1.3 Outline

The thesis is organized as follows:

Chapter 2 provides the necessary background information on theoretical concepts for

a clear understanding of the topics discussed in the subsequent chapters. It emphasizes the modern architectures of the many-core CPUs and multi-core GPUs, their respective programming models, and solutions that couple the CPU and the GPU aiming at a better performance.

Chapter 3 summarizes relevant state-of-the-art work. It includes a survey on the use of the GPU for computations of virtual engineering applications, concentrating on CFD cases and solvers for SLEs. It also describes in details distributed and heterogeneous platforms, based on CPU and GPU, aiming at application performance gain and systems that benefit from the workload distribution and dynamic assignment of tasks over the CPU and the GPU. A comparison between the main related work and the subject of this thesis is emphasized at the end of the chapter.

Chapter 4 introduces the methodologies and strategies adopted by the Sm@rtConfig framework to manage the allocation and reconfiguration of tasks over the heterogeneous architecture. These strategies are mainly based on low complexity heuristic algorithms and have the goal to perform a load-balancing of tasks that are needed to be concurrently executed.

Chapter 5 shows the CFD case study used to validate the proposed strategies, gives a mathematical introduction of the solvers for SLEs, and a method for implementing the solvers on the GPU, focusing on the loading data strategy that provides performance gains for the case study. It also describes the exploitation of the CPU and the GPU concurrently.

Chapter 6 discusses the experimental results based on a performance analysis oriented to the heterogeneous platform approach, presenting the performance break-even points using this heterogeneous execution platform. The chapter explains the use of the proposed framework and its benefits.

Chapter 7 closes the text with a conclusion of the presented work and the ongoing work to improve the framework functionalities. Finally, it signals directions on which further work can be conducted.

2 THEORETICAL BACKGROUND

With the recent development of consumable parallel and low-cost dedicated and generic purpose processors, several applications (including the virtual engineering applications) can be executed on "standard" personal computers or portable devices, like notebooks, with relatively good performance. It is an evolution on the microprocessor die, from single core devices to devices with multiple cores and parallel computational capabilities, being an alternative for high performance platforms. Figure 2.1 exemplifies, in the blue axis, the evolution of the processing units in terms of an increasing number of cores; in the green axis, the types of platforms that can be powered with hybrid processing units; and, on the red axis, the performance demand of applications and their evolution on personal platforms. The figure suggests a 3D relationship since processing units, applications, and platforms are highly evolving at the same time.

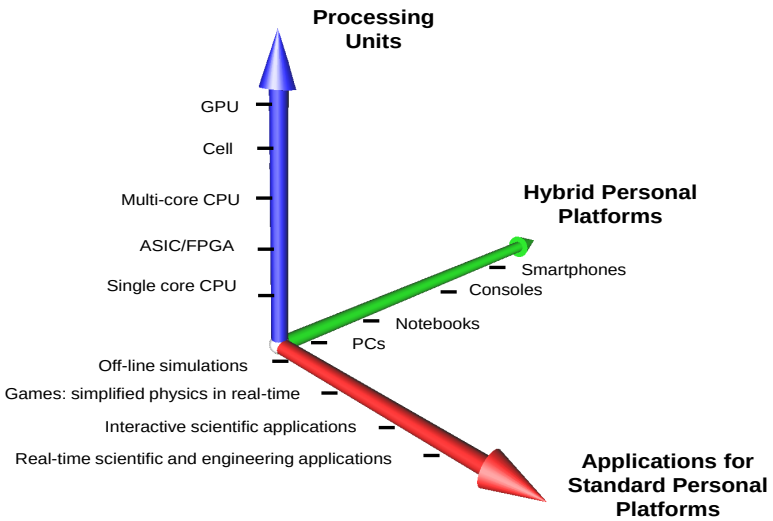


Figure 2.1: Relevant evolution of personal platforms and its processing units in correlation to application's needs

In addition, software libraries and runtime systems must also meet the programmers' and applications' needs for gaining performance, i.e., they must be implemented to explore the performance capabilities from that kind of hardware with multiple cores. In addition, it has to support the composition of heterogeneous processing units at the same

platform. Nevertheless, software techniques and strategies to explore modern multi-core and heterogeneous platforms are still under development, like the OpenCL (STONE; GOHARA; SHI, 2010) which is a framework for coding applications that execute across heterogeneous platforms.

This chapter presents the background information about the related hardware used in this research (CPU and GPU), their controlling APIs, and some approaches oriented to common programming over an execution platform composed of the heterogeneous hardware. Besides, a mathematical overview on the solvers for SLEs is given, since they are used as tasks of the virtual engineering case study. The background of the solvers helps on making clear how they were implemented for the CPU-GPU platform and the need to use the concepts presented on the developed framework.

2.1 Multi-core CPUs

This section gives the necessary background information for the understanding of the modern multi-core CPUs. It describes the hardware fundamentals and current software functionalities, mainly from APIs and standard software tools, to explore the performance given by the parallel hardware technologies.

2.1.1 Hardware characteristics

The main computational engine of modern computers, the CPU, is becoming highly parallelized. The sustained growth in transistor density (Moore's law) allows chip designers to put more and more processor cores on a single silicon die. Until 2004, consumers experienced an increase in performance through a steady growth in core clock frequency. Consequently, most commodity processors only included one processor core. Threads facilitate the concurrent programming and were commonly used in software programs to improve performance by splitting instructions into multiple streams so that multiple processors could act upon them.

As a first on-die indication of parallelization, the Hyper-Threading (HT) technology was produced by the company Intel and was one of the steps with the goal to bring parallelism at a single CPU in a higher program level (INTEL, 2010b). Emulating a dual-core processor, it provided thread-level parallelism, resulting in more efficient use of processor resources, higher processing throughput, and improved performance on multi-threaded software. The single processor supporting HT technology presented itself to the operating systems and applications as two virtual or logical processors and could work on two sets of tasks "simultaneously", use resources that otherwise would sit idle, and get more work done in the same amount of time. This was one of the first indicators oriented to multi-core.

In the last years, the community has seen only little increase in core clock frequency. The trend of the two major players (Intel and AMD) is to offer increased performance over symmetric multi-core commodity chips. Physical dual-core processors on one die have become a mainstream in desktop and mobile devices due to their ability to deliver high system performance for complex applications running at personal platforms. It is also more efficient on energy consumption, since, for example, two processors running on a half of frequency of one single core have the same computational power, but with

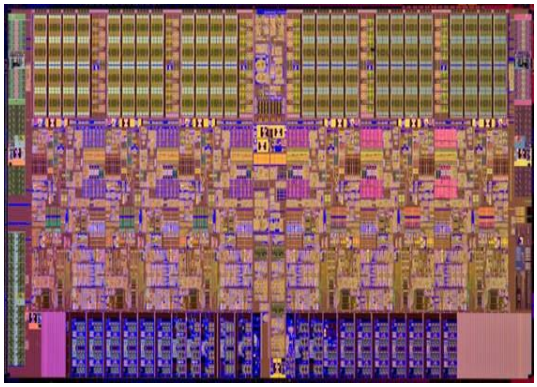


Figure 2.2: 6-core CPU from the Intel company (SCHROUT, 2010)

half of energy consumption.

In multi-core, application threads can be independently scheduled, dispatched, and processed by the available processors. In other words, multi-core processors enable true multitasking on the contrary of the old single-core systems where multitasking usually resulted in decreased performance as operations must wait to be processed in a queue. Thus, presently, on multi-core systems, each core has its own cache, providing the operational system with resources to handle compute-intensive tasks absolutely in parallel (MCCOOL, 2008).

The trend towards a higher number of cores is continuing strong with quad-core processors which are establishing an increasing presence across the market segments over a reasonable price, optimizing the trade-off between silicon material, performance, and energy efficiency. Figure 2.2 shows an example of the circuit of a six-core processor, the Intel's Nehalem Core i7 model.

Therefore, applications can obtain considerable performance gains using multi-core platforms. As an example, following the Amdahl's law, the amount of performance that can be gained depends on the characteristics of the application and states that a fraction of a program's execution time is infinitely parallelizable with minimal or no overhead, while the remaining part is sequential (AMDAHL, 1967). In a multi-core era, increasing core performance, even if it appears locally inefficient, can be globally efficient by reducing the idle time of the rest of the chip's resources, having a demand for global load-balancing "on the fly". Overall, performance gain should be viewed regarding the entire multi-core chip rather than punctually focused on one core or processor. The validity of this concept is intensified when dealing with heterogeneous processors (HILL; MARTY, 2008).

2.1.2 Software functionalities

Although the computing community settled on the random-access machine model for serial computing early in the history of computer science, no single model for parallel computing has gained as wide acceptance (CORMEN et al., 2009). Probably, a major reason is the competitiveness where the vendors have not agreed on a single architectural model for parallel computers. With the advent of the multi-core technology, every new laptop and desktop machine tends to become a shared-memory parallel computer and the trend appears to be toward shared-memory multiprocessing, where each processor can

directly access any location of memory.

The OpenMP API is one of the most known tools utilized to obtain a performance gain via software parallelization, being processor independent (OPENMP, 2010). It is used to explicitly create multithreaded and shared memory parallelism where the user specifies the regions in the code that can be executed in parallel and can also specify necessary synchronization to ensure correct execution of the parallel regions. More concrete, programmers write their code in standard programming languages, like C++, and provide hints to the compiler via *pragmas* about which loops can be parallelized and how the execution of the loop should be distributed among the processors. At runtime, threads are forked for the parallel region and are typically executed in different virtual processors, sharing the same memory.

It became very attractive to programmers because they do not have to use a new programming language, just adapt the code with the key pragmas and the compiler does the job. Unfortunately, the pragmas may differ between compilers, although there is a standard subset which is generally used, but requires a compiler that supports it. In addition, using vendor extensions for better optimization (and targeting a specific machine) comes at the cost of portability. It can also be inefficient on large shared memory machines because of non-uniform memory access effects and makes distributed memory implementations problematic since there is no notion of locality.

Within the multi-core era, some new tools - developed usually by chip vendors - became available. Intel, for instance, developed the open source Threading Building Blocks (TBB) as a template for writing software that specifically takes advantage from multi-core processors. It offers a high-level task-based parallelism that abstracts the complexity and platform details for performance and scalability goals, treating the operations as tasks that are dynamically allocated to the individual cores by the *runtime* module (INTEL, 2010c). The company also developed the C for Throughput Computing (Ct), a commercial software package to ease the exploitation of its multi-core chips. Just very recently, the Ct turned into the Array Building Blocks (ARBB) to provide a generalized vector parallel programming solution that frees application developers from dependencies on particular low-level parallelism mechanisms or hardware architectures (INTEL, 2011). It is comprised of a combination of standard C++ library interface and powerful runtime system. It produces scalable, portable, and deterministic parallel implementations from high-level source description. The ARBB functionalities are more detailed in Subsection 2.3 because it is a merge from Ct and a vendor-independent runtime system for heterogeneous processing units.

Similar to the TBB, the AMD company offers a commercial tool oriented to multiprocessing, named CodeAnalyst, which is a set of tools to analyze software performance devoted to AMD microprocessors (AMD, 2010a). However, it is not dedicated to bring control for software developers about the multiple cores, but it is oriented for profiling the code to identify bottlenecks and opportunities for parallelization (shown to programmer by a visualization tool). AMD, which acquired the ATI GPU manufacturer, is also investing on an tool for heterogeneous platforms: the OpenCL.

Going to a solution that is independent from processor's vendors, Cilk is a set of programming language extensions for C developed by the MIT laboratory since 1994 and focusing on high performance computing (BLUMOFE et al., 1995). With the advent of multi-core architecture, the tool became commercial as a library supporting the C++

programming language. The commercial version was shipped to the market in the end of 2008 and focuses to maximize application performance on CPU multi-core processors (INTEL, 2010d).

As the other libraries cited before, Cilk enables the programmers to develop parallel applications retaining the serial methodologies for programming. The programmer just needs to write a Cilk property command before the function to be parallelized. The Cilk compiler and the runtime module deals automatically with threading parallelization to *spawn* and schedule threads to processing elements, delivering code to the operational system that manages and balances the execution over the multi-core CPU. The main commands are $cilk_spawn$ (posed just before a call of a function to indicate that such function can be executed in parallel) and $cilk_sync$ (for synchronization). Additionally, it presents the $cilk_for$ to perform loop parallelization. Very recently, the company that commercialized Cilk was acquired by the Intel company.

2.2 Many-core GPUs

This section gives the necessary background information for the understanding of the modern many-core GPUs. It describes the complexity of the hardware architecture and the APIs, programming languages, and other software tools and functionalities that a programmer needs to use for developing an application that explores the parallel power of such a processing unit.

2.2.1 Hardware characteristics

Originally, Graphics Processing Units (GPUs) were designed to accelerate graphics applications, having the game industry as the main demand. GPUs are programmable processors based on the scan and rasterization concept presented on the graphics pipeline, which is implemented in a parallel way. It is based on SIMD (Single-Instruction/Multiple-Data) architecture and its processors are composed by several parallel processing group units that implement the so-called flow-based computation pipeline (GPGPU, 2010).

In summary, in its root, the computer graphics application would "prepare and send" a list of vertices (together with their properties) from objects placed on a scenario to the GPU. The hardware will, then, be responsible to apply a set of programmable effects and generates, as a result, the pixel colors of the final image to be visualized. Additional information on such details of programmable graphics hardware history, modules, and functionality applied to computer graphics can be taken from the work of Krüger (KRÜGER, 2006).

The approach of this research focuses on the performance of the GPU as a co-processor unit of the CPU for general calculations, i.e., the generic purpose of the GPU (OWENS et al., 2007). Figure 2.3 depicts the use of a graphics board as a device in a desktop with the CPU as a host. The CPU manages the execution of applications using the RAM memory and can transfer data to the memory of the co-processor using the bus. Once data is transferred to the device, the GPU programs are parallel executed and the final result can be transferred back to the CPU for further interventions.

This potential use of the GPU as a common parallel co-processor emerged during the last decade and made this PU evolving from being graphics-fixed functionality processors

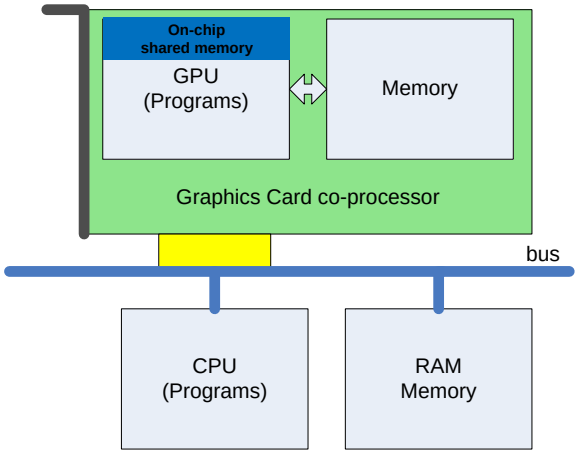


Figure 2.3: The GPU as a co-processor of the CPU

to become very powerful programmable many-core data stream processors (SCHIWIETZ, 2008). That vision was mainly possible due to its parallel conception of having different groups of processors (multiprocessors) working in the same calculation in parallel, but operating on different input data. In other words, at a given clock cycle, each processor of a multiprocessor executes the same instruction, but operating on different data.

Based on that parallelization, the GPUs, along its growth of performance gain, reported better achievements in terms of performance than the ones from CPUs, since it offers an even higher number of parallel cores than the central unit. The Nvidia GPU vendor comments that the main reason behind such an evolution is that the GPU is compute-intensive oriented, supported by highly parallel computation and therefore is designed such that more transistors are devoted to data processing rather than data caching and flow control (NVIDIA, 2010a).

For example, there are 480 cores (2x30 multiprocessors, where each multiprocessor is composed of 8 processors) in the Nvidia GeForce GTX 295 model compared to 4 units of the quad-core CPU. A commonly used comparison about the evolution of GPUs is based on Moore's law. It is a prediction concept positing that the number of transistors placed on an affordable commercial chip doubles in every two years. However, the law seems to be not valid when it comes to the graphics hardware scenario, even when the number of transistors alone does not reflect a gain of performance since there are some overhead using GPUs, like data transmission. Figure 2.4 shows a comparison of the performance growth between CPUs and GPUs over the recent years, emphasizing the peak capacity. It represents the real local peak performance of the processors for floating point calculations.

On one hand, the GPU is individually faster than the central processor, but on the other hand, it needs the CPU as the manager of the whole process. Moreover, the GPU presents some bottlenecks, related here as two that are relevant for the scope of the thesis: specific architecture model and bus communication bitrate. The hardware complexity makes it difficult to program and obtain a maximal gain of performance without a good knowledge of the specific architecture model and programming language. More details are discussed on the next subsection.

The other limitation is noticed on the communication between CPU, the manager or host, and the GPU over the bus, which can directly affect all the benefits from its

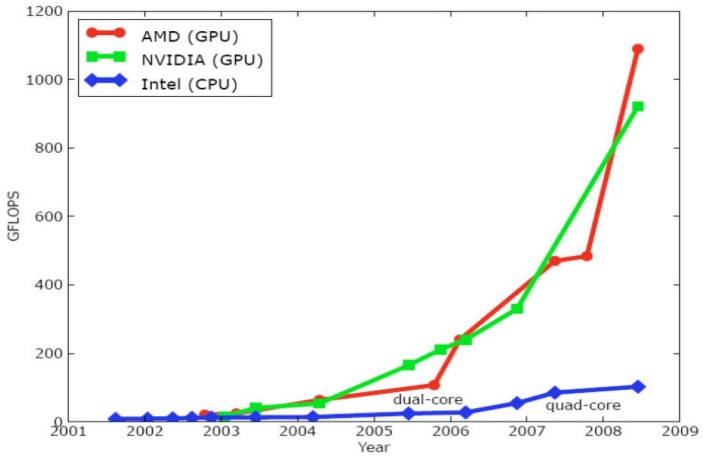


Figure 2.4: CPU (from Intel) versus GPU (from Nvidia and AMD) performance growth

individual general purpose computation performance. A large amount of data (normally used in scientific virtual engineering applications) needs to be transferred from the system memory to the GPU memory; and in most cases the result is transferred back to the CPU to perform further processes. In other words, the PCI Express bus for data intensive upload (from CPU to GPU) and download (from GPU to CPU) can restrict the GPU usability, since the total transferring time can suppress the GPU processing gain.

The bus is responsible for this communication between computer components (input/output). The PCI Express is the fastest communication pattern gateway among devices at the present moment. For example, one of the most common buses, the PCI Express 32x version 1, achieves a bandwidth of 8 GB/s dedicated for each direction (upload-/download). Unfortunately, this bitrate is considered slow when compared with memory accesses, illustrated by Figure 2.5, and some scientific works aim to overlap communication with computation (WHITE; DONGARRA, 2011). Note that the technology of the different hardware changes drastically over a small period of time and for reasons of a stable timeline comparison, the figure adopted average values of data transfer from technologies consolidated on the last year or before. As an example, the PCI Express is currently in the version 2 and the bandwidth speed was doubled when compared with version 1 of the picture, but, at the same time, the other hardware technologies were/are accordingly improved.

2.2.2 Software functionalities

The general purpose computation using GPU is a recent area that emerged from the GPU parallel processing capacity. Albeit being programmable, restrictions have to be met to achieve full performance since, frequently, the parallelization of algorithms needs to take the GPU specificities and architecture into account.

From the emerging of the field of GPGPU in 2002, researchers have been pointing out that the hardware could be also explored beyond graphics (GPGPU, 2010). This way, several *tricky* methods were developed to explore the graphics connotation to perform non-graphics workload, like the work of Binotto, Comba, and Freitas (BINOTTO; COMBA; FREITAS, 2003). The authors employed an approach for large sparse data

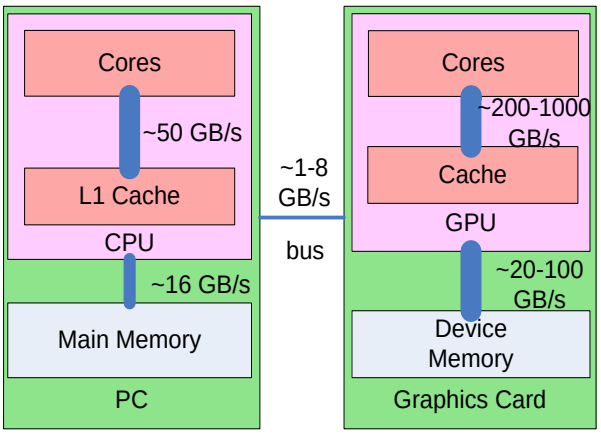


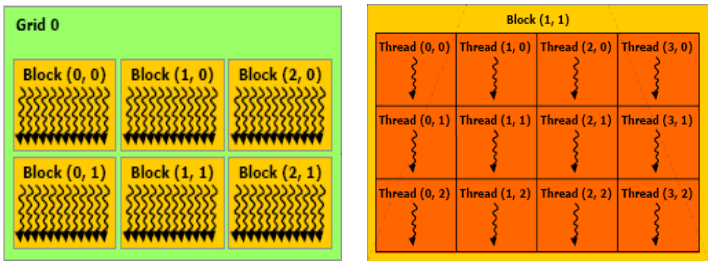
Figure 2.5: Data transfer bandwidth comparison between CPU and GPU and their memory accesses - partially based on (GÖDDEKE, 2010)

structure compression, via *shader* programs, using massively the GPU memory, called *texture* when applying the graphics nomenclature.

Given the promising computational capabilities, there have been several academic and industrial efforts to create languages to generalize the hardware interaction for GPGPU. The approach for general purposes should consider a model that encapsulates a host (CPU) and the GPU as a co-processor. This way, a GPU-based algorithm should result in a multitude of steps (calculations) between host and GPU, where data can be transferred from the host to the GPU for some processing and results are transferred back to the system main memory.

BrookGPU (BUCK et al., 2004) is a streaming language based on C and was the first tentative to abstract a graphics processor as a streaming co-processor. Data should explicitly be transferred to and from GPU memory. As a streaming computation model, the user defines kernels which operate over streams. Kernels are invoked once per output stream element and executed in a data parallel way with no communication or synchronization between kernels. It directly maps streaming concepts to the graphics APIs, i.e., all runtime calls and kernels are mapped to graphics API primitives, such as *textures*, *framebuffers*, and *shaders*. BrookGPU is implemented as an extension to C and it supports Nvidia and ATI GPUs. Although it is no longer supported, AMD has used it as the basis for the AMD Stream (AMD, 2010b).

From that point, both main GPU vendors, Nvidia and ATI (now incorporated by AMD), developed parallel-oriented APIs for their chips. Regarding Nvidia, from the generation G80 on, the device implements a set of SIMD programmable multiprocessor groups that can be accessed via threads. Generically, this new architecture is well-suited to address any application that can be expressed as a data-parallel computation without using the tricks of graphics APIs. Based on that, the CUDA (Compute Unified Device Architecture) architecture was designed (NVIDIA, 2010a). Using CUDA, a GPU program (kernel) is organized as a grid of thread blocks, where a block is a batch of threads that can communicate with each other and that can have synchronization points. It includes a device-independent assembly language (PTX - Parallel Thread Execution) that is the basis on which multiple parallel language and API interfaces are built for CUDA-capable



(a) (b)

Figure 2.6: CUDA's grid, block, and thread organization - extracted from (NVIDIA, 2010a)

Nvidia GPUs. In its specification, threads in different thread blocks cannot communicate nor synchronize with each other. Figure 2.6 exemplifies the concept.

The user must explicitly code the data transfer from host memory to device memory before program execution. However, CUDA does not use streaming during program execution, but uses explicit general reads and writes via pointers and arrays. Each kernel instance corresponds to exactly one thread. The programmer organizes the parallel execution by specifying a so-called grid of thread blocks which corresponds to the underlying hardware architecture. Each thread block is mapped to a multiprocessor and all thread blocks are executed independently. The thread blocks constitute of a set of multiprocessors and their size should always be a multiple of the called *warp* size, being a half-warp (consisting of 16 threads) the atomic set for processing.

In terms of memory hierarchy, CUDA exposes three memories to the user: a per-thread *local memory*, per-block *shared memory*, and the device *global memory* where it is the interface with the RAM host memory. Synchronization is, thus, defined for threads in a block, allowing for data sharing and communication between threads. As many threads can simultaneously access shared memory, the memory is designed as a *multi-banked* memory (GÖDDEKE, 2010). Successive 32-bit words are assigned to successive banks and each bank can be accessed by one memory request per cycle, being the shared memory as fast as registers.

Based on that, the programmer should avoid three main problems. The first deals with accessing the shared memory. Since it is a cache-like memory, used to offer faster deliverables of data, multiple simultaneous accesses to a bank of memory are characterized as a conflict and the memory requests are serialized. The second problem is global memory coalescing. Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in one or two transactions. However, in the presence of an offset, like a loop instruction through an array with offsets on the controlling variable, the accessed locations of the memory can not be close enough to be coalesced, i.e., agglutinated in a single transaction, resulting in a not efficient data gathering from the memory. The third issue is related to conditional statements. The programmer should avoid branches on the code, since it results in a larger number of transactions. In summary, these three considerations depends on the CUDA-capable GPU hardware model, being more severe for some specific architectures, and are also valid for the AMD GPUs.

From the side of AMD, the ATI's Close To Metal (CTM) provide a very low-level access to the graphics hardware (AMD, 2010b). Considered of complex use by most of the programmers, it quickly turned into the AMD Stream SDK after the key player chip

producer AMD acquired ATI. It incorporates the streaming language BrookGPU.

While Nvidia's CUDA is a more abstract high-level API, the Stream SDK exposes the GPU in more details. To overcome this characteristic, not needed by most of the programmers who would be concentrated in the problem they are coding for, the API presents the Compute Abstraction Layer (CAL). It includes a set of methods and data types with the goal to bring - to a high-level - the control of stream processor programs and memory access. Together with a runtime module, the CAL generates optimized code for the AMD hardware architecture, which is distributed automatically over the stream processors. Just very recently, the Stream SDK turned into the Accelerated Parallel Processing (APP) SDK with the goal to enable a more close interaction between the CPU and the GPU for parallel processing towards the new chip called Fusion (AMD, 2011), which merge both PUs on a single die.

The Nvidia hardware and the OpenCL API are the focus of this theses. Therefore, the AMD GPU is only briefly described.

2.3 Processors' Trends

Based on growth of performance requirements of modern applications, like the virtual engineering ones, and on the growth of the number of applications that one single user can deal pervasively, the traditional CPU evolved to multiple cores. Normally, the cores are homogeneous, but could also be heterogeneous. An example of such a CPU is the Cell B.E., having one "fat" core and eight "thin" data parallel cores (BELLENS et al., 2006).

Publications from Intel show that in the future the industry will move beyond a small number to tens or hundreds of processor cores, called the "tera-scale", where performance will be delivered with the integration of more and more cores on a die and also with runtime techniques that distribute and manage the tasks properly on the cores (AZIMI et al., 2007; KUMAR; HUGHES; NGUYEN, 2007). For its control, a programmable model for tera-scale architectures is already being on test, the Ct (INTEL, 2010e). The Intel enforces that Ct will be a deterministic (assuming that program behavior is identical in every core) parallel programming model intended to leverage the best features from GPGPU with full exploration of tera-scale CPU. The vendor also claims that Ct will be much more flexible, using mainly data types and a powerful runtime, opposed to the GPGPU approach that is designed to the underlying constraints of the proprietary architectures.

Following the line to integrate heterogeneous processors, both AMD and Intel have announced that they will produce heterogeneous multi-core processor, combining the data stream cores of GPUs and traditional CPU cores. For example, the Larrabee developed by Intel (SEILER et al., 2008), is a throughput-optimized many-core implementation of the x86 architecture for visual computing, but more flexible than GPUs and "closer" to a conventional CPU cache hierarchy. On the AMD's side, there is an intention to use the ATI GPU as a vector co-processor more closely tied to the CPU, sharing resources such as cache hierarchy. The AMD Fusion was, then, announced to be a heterogeneous multi-core multi-processor architecture, combining general purpose processing core(s) and basic graphics core(s) into one processor package, but with different clocks' frequency for the graphics core and the central processing core (AMD, 2011). It is a heterogeneous execution platform on a single die.

2.4 Managing Heterogeneous Execution Platforms

The current scenario indicates that the era of high numbers of cores and heterogeneity in several levels has just begun. A considerable software effort will be needed to concentrate and aggregate all hardware benefits in one common line: promote better performance and accomplish with application requirements, specially the scientific ones.

The technology advances drives professionals to execute complex applications in their desktop instead of supercomputers. Nowadays, it is possible to gain in parallelization not just individually on each multi-core chip, but also on the combination of various similar chips or asymmetric ones. Merely as an example, a desktop can be composed by a multi-core CPU and several co-processors.

The next subsection explores the programming languages that can guarantee control of heterogeneous hardware in a higher level from the application point of view. At the moment, there is one general-purpose programming API for heterogeneous execution platform that is becoming a standard at the commercial and the academic fields: the OpenCL (STONE; GOHARA; SHI, 2010). It leverages the processing units at all, including accelerators. The goal is to congregate the heterogeneous hardware by means of providing a programming abstraction layer and a workload balance throughout different computing units, like a hardware-software co-design approach. However, before detailing the OpenCL, a first discussion about the RapidMind platform is given, since it was one of the first package that provided a vendor-independent abstraction layer for the CPU and the GPU (and also for other PUs) programming (MCCOOL, 2008).

2.4.1 RapidMind

The RapidMind Multi-core Development Platform was a commercial tool from the company RapidMind and was based on *Sh* (MCCOOL et al., 2004), a meta-programming language for programmable GPUs developed by the University of Waterloo. Sh was a library to be used together with C++, supporting Nvidia and ATI GPUs on Windows and Linux. Sh evolved to a commercial tool (the RapidMind) for simplifying the development of parallel applications and leverage heterogeneous processors. The main goal was to gain performance based on asymmetric hardware, reducing the error of manually multi threading an application and reduce the software developing time.

The tool was defined as a set of API functions, used within the C++ context, a compiler, and a runtime module. It used a Single Program Multiple Data (SPMD) model and a programming model where kernels are specified in the main source code of a C++ program. It encapsulates the directives and provides compatibility to execute kernels in the Nvidia and AMD GPUs, as well in the Cell and in multi-core Intel and AMD CPUs (MCCOOL, 2008). Figure 2.7 depicts the architecture of RapidMind.

The RapidMind extension presented three main types: *Value* (float, int, etc), *Array* (array, vectors of data), and *Programs* (set of instructions using Value and Array). Specifically, Programs were a collection of instructions performed with RapidMind types and operations on one target hardware (CPU, GPU, or the Cell processor). The RapidMind operations were: loops, control flow, comparison, and algebra and arithmetic operations.

Based on the user manual, RapidMind performs dynamic runtime compilation on Programs. There is a "begin" and an "end" macro statements to indicate the interval of Pro-

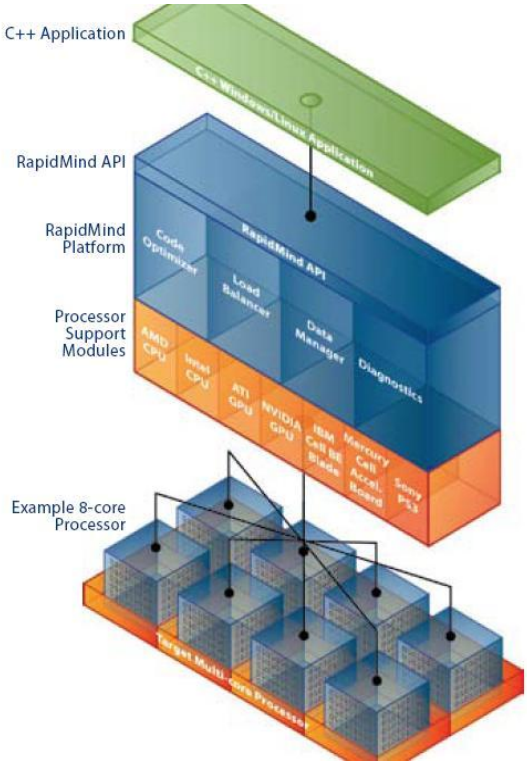


Figure 2.7: RapidMind platform overview - extracted from the web site rapidmind.net, which is no longer available

grams' instructions. The compilation itself is made normally with the C++ compiler, converting the RapidMind programs into a sequence of instructions (like intermediate object instructions) and, at run-time, the tool creates machine code for the target hardware that will be used to execute the code. This way, just before the first execution starts, the RapidMind compiler (not the C++ compiler) converts the intermediate code representation to target machine code and thereafter, this machine code is executed each time the Program is invoked. The compilation can generate different (and optimized) versions of a Program depending on which inputs and outputs are bound. As a feature, the code generated by the RapidMind in the API language of the target hardware can be accessed by third applications.

Regarding the important issue of load-balancing, the software developer does not need to manage such issues. A set of preferred hardware - where the Program has the possibility to be executed - can be configured and, in this case, or in case of no specification of the target hardware, the RapidMind generates automatically machine code and executes the Program according to the "best" target hardware presented in the execution platform. This process to find the "best" unit is done according to an internal cost determination mechanism based on "black-box" strategies, where the platform will choose the processing unit using heuristics to determine which hardware is the most appropriate. As a possibility, the user can specify at programming time if he wants to use one fixed target hardware to process each Program, assuming responsibility for the load-balancing issue. Once the instructions are on the co-processor, each PU manages itself the fine-grained balance inside its internal cores.

Nevertheless, the user manual does not present details about the aforementioned heuris-

tics, probably because it is a commercial product. But, some assumptions can be inferred. Following that the compilation is made just once by RapidMind (and the code generated for the target hardware is also produced just once), the tool presumably does not take into account new runtime conditions presented in the whole execution platform, not dealing with the reasoning for a new balance in further executions. Probably, it deals just with the static characteristics of the processing units to perform the high-level balance.

It is also likely inferred that there are no connection between the profiling module and the balancer in a way that the profiler feeds the balancer with current performance and system workload information as well as application needs for a better reasoning of the load-balancing process.

The RapidMind company was acquired in 2009 by the Intel company and now makes part of the solution named Array Building Blocks (ARBB), which, combined with the Ct programming model (INTEL, 2010e), encapsulates most of the mentioned concepts. It is still in a *beta* version and remains supporting all standard C++ compilers, like Microsoft Visual C++ and GCC C++. The use on non Intel processing units are unclear at the moment of writing this thesis.

A similar commercial product was developed by a company named PeakStream, called the PeakStream Platform for Many-Core Computing, towards supporting CPUs and GPUs. However, the company had an early life and the product was discontinued on its acquirement by Google in 2008.

2.4.2 OpenCL

Based on the previous subsections, it is clear that heterogeneity is starting to drive low-cost technologies and that there is a lack of tools to deal with this evolution on a programming point of view. CPUs and GPUs, for example, are very different in their parallel programming models. While the CPU works normally based on standards and usually assumes a shared address space, the GPGPU programming models address complex memory hierarchies and vector operations, but are traditionally platform-, vendor-, or hardware-specific.

The OpenCL (Open Compute Language) arrived with the goal to fill this gap between standards and specificities, i.e., to support several heterogeneous processing units using the same code. It is an open API for parallel programming of heterogeneous systems, initiated by Apple and managed by the Khronos group[1] in a consortium composed of several companies and research institutes that integrate the board of specifications and development, including Intel, Nvidia, AMD, Fraunhofer, among others.

Basically, the OpenCL should support applications varying from embedded and consumer software to high performance computing solutions using a hardware abstraction and allowing portability over the execution processing units. On the contrary of dedicated APIs, like CUDA or other vendor languages, the OpenCL concept is to be a non-proprietary and royalty-free standard for a low-level layer interface that targets several processing units (CPUs, GPUs, Cell, and others from different vendors) from an independent spectrum, without having to understand the architecture of the underlying hardware. Developed and adopted by several important companies and academic institutions in less than two years, each vendor should deliver their products together with the respective

[1] http://www.khronos.org/opencl/

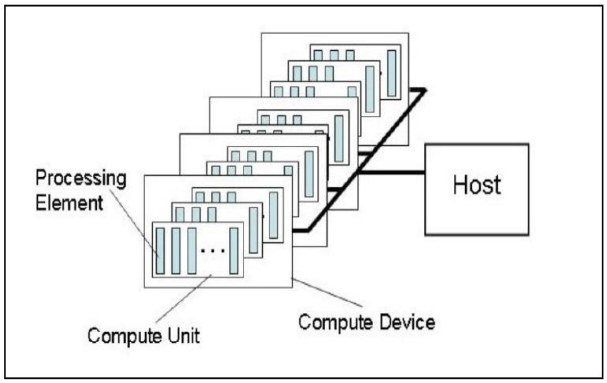


Figure 2.8: OpenCL execution platform organization - extracted from (STONE; GOHARA; SHI, 2010)

OpenCL support driver/wrapper that translates the coded OpenCL program into the proprietary low-level instructions.

For that, OpenCL uses the following general architecture (STONE; GOHARA; SHI, 2010): one Host plus one or more Compute Devices (processing units). The devices are multi-core and subdivided in one or more Compute Units which, in turn, are composed by one or more Processing Elements. With a simple call for querying the devices at the execution platform, the application would know about the platform issues, like number of processing units, computing cores, maximum work-group size, sizes of the different memory spaces (constant, local, global), among other static features. Figure 2.8 outlines the hierarchy, where an OpenCL application submits commands from the host to execute computations on the processing elements within a device.

The execution of an OpenCL program occurs in two parts: kernels that execute on one or more devices, and a host program that executes on the host and manages kernels' execution. When a kernel is submitted for execution by the host or compute device, an instance of the kernel is executed. This kernel instance is called work-item, having a global identification (ID). Each work-item executes the same code, but the operated data can vary per work-item. The work-items are organized into work-groups, which provide a more coarse-grained decomposition denoted also with an ID. A work-item is executed by a processing element and the work-group by a compute unit. Parallelization is done automatically when the work-items in a given work-group execute concurrently on the processing elements of a single computing unit. It is also done for several compute units when driven by the programmer.

Taking specifically the CUDA for the GPUs, developers experienced relevant speedups in fields such as scientific application, but with the cost of understanding the hardware characteristics. Leveraging the massively parallel processing power of Nvidia's GPUs, OpenCL running on the CUDA architecture should encapsulate the complexity of the hardware. This way, the Nvidia CUDA driver for OpenCL has the role to match the simplicity and generality of OpenCL commands to the intrinsic features of the graphics unit in an optimized way (NVIDIA, 2010c).

Technically, it seems that both C coding for CUDA and OpenCL implementations perform conceptually the same steps. They are syntactically and semantically very similar. The main differences should basically be the naming schemes and how data is accessed

using the API. The memory hierarchy presents a private area and the work-items and work-groups for constant and global memory. This reminds the concepts of blocks and shared memory per block in CUDA.

Both OpenCL and the CUDA driver APIs require the developer to manage the contexts and parameter passing. One noteworthy difference is that C for CUDA programs are compiled with an external tool (the NVCC compiler) before executing the application. This compilation step is typically performed when the current application is built. In contrast, the OpenCL compiler is invoked at execution time, like the approach of RapidMind, generating, for example, CUDA code to execute in the Nvidia GPUs. The runtime module acts in the sequence and is supposed to take the generated code, partition it, and balance it over the similar compute devices that can be found in the system. For that accomplishment, it probably uses a set of strategies that are still not published.

Furthermore, OpenCL brings a powerful and significant set of querying functions at execution time, i.e., dynamically, like the profiling of platform and devices information (memory amount, device model, device configuration, etc). It can be acquired also execution time information from tasks and also information from memory, program, and kernels' objects.

Additionally, there are no concrete indications that OpenCL will have a high-level load-balancing module that will choose the best Compute Unit to execute its tasks (kernels). This is actually done by the programmer. However, it supports both data- and task-parallel computing models targeting all heterogeneous processing units in the execution platform, i.e., all Compute Devices. It is just known that, using the API, the programmer can gather parameters and data from the available Compute Units presented on the platform and "hard-code" decides in which device the execution will be performed. There are also some indications that there are no connection between a profiler and the balancer in a way that the profiler feeds the balancer with runtime information during a period of system lifetime towards dynamic assignment.

At the time of writing this thesis, the OpenCL with the specification 1.0 was used, implemented by the driver of the AMD Stream, which can be used to execute code for Intel and AMD CPUs (it supports x86 processors with SSE3 - Streaming SIMD Extensions 3) and for AMD and Nvidia GPUs. The OpenCL web page can be referred for current specification details.

2.5 Chapter Remarks: CPU-GPU heterogeneity trend on personal computers

The field of high performance computing has become increasingly significant at a common desktop composed of, for example, multi-core CPUs and many-core GPUs. This is extrapolated to other computing units, like the Field-Programmable Gate Arrays (FPGAs), which can currently be also multi-core configured, and other application specific hardware. The processing units are becoming faster, less expensive, and more cost-effective, which is resulting in a proliferation of the use of parallel and distributed systems from applications based on personal computers. Scientific and virtual engineering application domains are key areas where the solution of large and complex problems could profit from a CPU/GPU-based execution platform to fulfill tight timing require-

ments. This is evidenced by several state-of-the-art works (HWU; KEUTZER; MATTSON, 2008; YEH et al., 2008; MCCOOL, 2008; INTEL, 2010e; AMD, 2011).

Nevertheless, coupling different architectures imposes challenges. The programming models to manage such environments are still under constant development and several applications are non-trivial to be parallelized by some current architectures, like for the GPUs. For CPUs, non uniform memory access problems are considerably increased as the number of cores get higher. Additionally, the joint work of CPUs and GPUs must be well designed by the programmer so that the application does not "suffer" from the bandwidth bottleneck imposed, for example, by the PCI Express bus.

In summary, this chapter highlights the main features of the modern CPUs and GPUs and how they can be combined to improve the performance for modern personal execution platforms for high performance computing. As the RapidMind was discontinued, the OpenCL framework has been introduced as being the mainstream technology to work with the CPU and the GPU concurrently and as being platform independent. It was verified that, currently, there is no load-balancing of high-level tasks (Kernels on the OpenCl nomenclature) across multiple processing units (Compute Device on the OpenCl nomenclature). Evidence for that are explicit commands presented on the OpenCL guidelines for specifying, at programming time, which device a kernel would execute. The investigations carried out in the scope of this work rely on this specific issue, which is still being improved by the scientific community as shown in the next chapter.

3 STATE OF THE ART SURVEY

This chapter starts giving an overview on distributed systems conceived for multi-core and heterogeneous-based architectures, concentrating on those envisioned for high performance computing. In the sequence, important and relevant results related with methods and strategies for the tasks of solvers for SLEs oriented to be executed on the CPU and the GPU are presented. Based on that, different works related to the scheduling of tasks, mainly based on heuristics, on a CPU-GPU platform are emphasized. Such techniques may represent gains on performance for executing the solvers for SLEs tasks on a CPU-GPU platform using runtime and tuning systems as a support for the execution of virtual engineering applications.

The intention, thereby, is to complement the last chapter, which depicted the state of the art of mainly the industry community, identifying what are the trends and results already achieved by the academic community in this heterogeneous research field. At the end of the chapter, some of the open problems for further research are itemized.

3.1 Distributed Processing on Multi-core Platforms

Concerning distributed processing in cluster computing relevant related work is given in, for example, Wang et al. and Linderman et al. (WANG et al., 2007; LINDERMAN et al., 2008). The first work presented, at a level closer to the chip, a shared virtual memory and methods for multi-thread programming. The second work contributed with a general purpose programming model for heterogeneous multi-core systems, being both works complementary. In addition, Brandenburg et al. analyzed a set of global, clustered and partitioned scheduling algorithms for multi-core platforms, coming to the conclusion that for multi-core platforms with on-chip shared caches, preemption and migration costs of a task can be considerably more costly if caches are small or memory bandwidth is limited (BRANDENBURG; CALANDRINO; ANDERSON, 2008).

Additionally, the PhD research of Houston conceived a platform-independent runtime interface for moving data and computation through parallel machines with multi-level memory hierarchies (HOUSTON, 2008), targeting clusters of CPUs and the Cell Processor (BELLENS et al., 2006).

However, the approach of this thesis concentrates on single desktop platforms composed by different processing units where tasks are assigned within these PUs. In this way, the work presented by Götz, Dittmann, and Xie implements dynamic reconfiguration methods for Real-Time Operating System services running on a Reconfigurable

System-on-Chip platform based on CPU and FPGA (GÖTZ; DITTMANN; XIE, 2007). The method, based on heuristics, take into account the idleness percentage of the CPU and unused FPGA area to perform a load-balancing of tasks and to decide about a reconfiguration of tasks in runtime by means of task migration. These findings resulted in the PhD research of Götz (GÖTZ, 2007), which partially motivated this thesis, being both works complementary.

Another approach, but focusing on the performance improvement of spheres collision detection simulation, was proposed by Joselli et al. (JOSELLI et al., 2008) where some strategies were presented to perform data-balancing over CPU and GPU, both in an automatically and manually options. The work takes into account the performance of a kernel implemented on the CPU and the GPU. After the execution starts, both versions of the programs are executed with the same input data and time performance is verified. More data are, then, dynamically assigned to the processor that executed faster the previous data, indicating that the approach uses data decomposition instead of task decomposition.

Additionally, coupling GPU and FPGA, Che et al. presented a study to accelerate compute-intensive applications using GPUs and FPGAs, listing some of their pros and cons (CHE et al., 2008). The work performed a qualitative comparison of application behavior on both computing units taking into account hardware features, application performance, code complexity, and overhead. Although the GPUs can offer a considerable gain for application performance, the results of the work showed that FPGAs can be an interesting computing unit and could promote a higher performance compared to GPU when applications require flexibility and deal with large input data sets. However, it comes with the cost of configuring the hardware before using it as a computing unit, making it suitable to experienced users.

3.2 Scheduling on a CPU-GPU Platform

According to Garey and Johnson, the task scheduling problem is considered to be NP-complete (GAREY; JOHNSON, 1990) and several heuristics were developed to better meet a good scheduling with little overhead, like, for example, the distinct approaches used by the authors of (TOPCUOGLU et al., 2002; AHMADINIA et al., 2004) for heterogeneous PUs. The first work makes use of performance prediction to calculate the expected time a processor will be available in order to assign a task to the processor that finishes firstly its tasks. The second work implemented an scheduler for the operating system to place tasks to be processed by software and hardware, called software-hardware co-design. In this case, over the CPU and an FPGA using a similar scheduling technique, based on the *earliest deadline first* to sort the queue of tasks.

Just very recently, some techniques are starting to be directly applied to a CPU-Co-processors execution platforms. Targeting GPUs, the work of Takizawa, Sato, and Kobayashi presented a programming framework to achieve energy-aware computing (TAKIZAWA; SATO; KOBAYASHI, 2008). On the proposed strategy, the compiler translates the framework code to a C++ code for CPU and a CUDA code for Nvidia GPU. Then, a runtime module dynamically selects the appropriate processor to execute the code taking into account the difference in energy efficiency between CPU and GPU based on energy consumption estimation models. However, it has no runtime energy measurements (profiling).

Involving load-balancing of GPU tasks, a comparison of dynamic scheduling methods based on lock and lock-free strategies - supported by atomic operations, like CAS (Compare-And-Swap) and FAA (Fetch-And-Add) - was presented (CEDERMAN; TSIGAS, 2008). It was focused only on the load-balancing of CUDA-based tasks inside the groups of processors of the GPUs (grids and blocks). The work showed that the traditional Task Stealing load-balancing scheme obtained a good performance when dealing with runtime creation of CUDA tasks. Also exploring just GPUs, Hong and Kim proposed a model to estimate the execution time of tasks based on the number of parallel memory requests, considering the number of running GPU threads and memory bandwidth (HONG; KIM, 2009). The PhD thesis of Brodtkorb also explored the solvers for SLEs' computations on the GPU, indicating the benefits from a heterogeneous platform for scientific applications and the need for a good assignment strategy. The work used a simple criteria based on dependencies, i.e., it places dependent tasks in the same processor. The group of tasks are, then, placed considering a pre-compilation phase where a basic set of tasks, like matrix-vector multiplication with a small problem size, is executed and a performance approximation for every processing unit is benchmarked and used for scheduling decision (BRODTKORB, 2010).

Focusing exclusively on dynamic task scheduling, Song, Yarkhan, and Dongarra described an approach to execute dense linear algebra algorithms (based on factorization methods) on a distributed-memory CPU cluster (SONG; YARKHAN; DONGARRA, 2009). Recently, they discussed a task scheduling approach to execute dense linear algebra algorithms (Cholesky, LU, and QR factorizations) on a CPU-GPU platform, but assigning independent functions to the PUs in a statically way, i.e., scheduling sequential functions to the CPU and data parallel ones to the GPU (TOMOV et al., 2010). This way, they proposed hybrid factorizations over the CPU and the GPU. Latest results from the same authors showed a technique to optimize the communication bottleneck over the PUs, overlapping it with parallel calculations (WHITE; DONGARRA, 2011), being suitable for runtime systems.

3.3 Runtime Systems for SLEs and CPU-GPU Platforms

The authors of (DIAMOS; YALAMANCHILI, 2008) developed a runtime system oriented to abstract the compute kernels for CPU and GPU, ensuring dynamic binary portability, configuration, and compilation over the PUs, but it does not address scheduling strategies. Following, Jimenez et al. focused on a runtime code scheduling based on past performance history and classical scheduling algorithms for matrix-multiply tasks over the CPU-GPU execution platform (JIMéNEZ et al., 2009). Their experiments demonstrated a speed up to 40% comparing to a scheduling just on CPUs or GPUs. Nevertheless, the system worked just with a restricted number of tasks and did not consider several input data sizes for scheduling, which is important for matrix-multiplication tasks.

Additionally, Augonnet et al. (AUGONNET et al., 2009) also described a history-based runtime system that mainly used the priority-based HEFT (Heterogeneous-Earliest-Finish-Time) static model presented by Topcuoglu for scheduling tasks (TOPCUOGLU et al., 2002). The tasks are represented as *codelets* to abstract the PUs' programming languages. However, it needs a calibration phase to perform a first good assignment of tasks on the CPU-GPU execution platform. The scheduling was improved by Augonnet

and his colleagues, which extended the StarPU to minimize the cost of transfers between processing units in order to efficiently cope with multi-GPU platform configurations (AUGONNET et al., 2010). To achieve this goal, the runtime system implemented data prefetching based on asynchronous data transfers, and uses data transfer cost prediction to influence the decisions taken by the task scheduler.

Recently, the work presented in (AGULLO et al., 2011) showed an efficient CPU-GPU-based hybrid algorithm for QR factorization. For multiple kernels to be executed in a hybrid way by GPUs and CPUs distributed over a large cluster, they concluded that static schedulers can achieve very high performance when the platform is relatively homogeneous (same number of GPUs and CPUs). However, when the node is more heterogeneous a runtime system performing dynamic scheduling is more appropriate to exploit the machines in a more optimal way. This point of view, however, reflects a layer above the heterogeneous processing units, i.e., their algorithms consider a cluster node (composed of CPU and GPUs) to be homogeneous (same number of GPUs and CPUs) or heterogeneous (just CPU, for example). This way, dynamic scheduling is more appropriate for heterogeneous execution platforms. They also emphasized the importance of highly tuned tasks with the aim to extract all the performance a processor can deliver for a specific kernel.

3.4 Tuning Systems for SLEs and CPU-GPU Platforms

The authors of (DATTA et al., 2008, 2009) described an automatic generation of many versions of a code kernel that incorporate various tuning strategies. They discussed their performance benchmark issues to select the best performing version for multi-core CPUs. The application was focused on stencil computations, applied to partial differential equation solvers, that consists on sweeping over a spatial grid, performing nearest neighbor computations like the iterative solvers for SLEs. Complementing, Nguyen et al. optimized the 7-point-stencil calculations obtained a performance result 1.5 time faster for CPUs and 1.8 time faster for GPUs than other previously published approaches (NGUYEN et al., 2010). Both works propose an auto-tuning approach to select appropriate block and grid parameters for the GPUs.

The work of Lee and Eigenmann proposed a fully automatic compilation and user-assisted tuning system supporting OpenMP and CUDA for Nvidia GPUs using solvers for SLEs as the main application (LEE; EIGENMANN, 2010). The parameters assigned for tuning include, among others, number of blocks and their size for a kernel and the use of shared or global memory. As the case study for validation, they used the Jacobi solver for SLEs because of its simplicity on the CPU. However, its basic translation to GPU code performs poorly because od the uncoalesced global memory accesses. To overcome this problem, a special code was produced with a runtime tuning support of the CUDA parameters. Their results are comparable to those generated in this thesis (BINOTTO et al., 2010), which identified the problem size as one important parameter for tuning purposes. It was followed by Gharaibeh and Ripeanu that itself ascertained that size, in terms of space/time tradeoff, is crucial to improve the performance of GPGPU applications (GHARAIBEH; RIPEANU, 2010).

Complementing, Patus is a code generation and auto-tuning framework for stencil computations targeted at multi- and many-core processors, such as multi-core CPUs and

GPUs (CHRISTEN; SCHENK; BURKHART, 2011). The work is based on the OpenCL framework and, together with this thesis, is one of the pioneers to present a runtime and tuning systems based on OpenCL. It has the difference to also generate hardware-specific tuned code instead of concentrating on scheduling and tuning of application parameters.

Finally, Maestro is a tuning system based also on the OpenCL (SPAFFORD; MEREDITH; VETTER, 2010). Because OpenCL requires that the programmer explicitly controls data transfer and device synchronization, the work provides automatic data transfer, task decomposition across multiple devices, and auto-tuning of dynamic execution parameters. Its main concept is very similar to the concepts of a previous work that makes part of this thesis (FREITAS et al., 2008a), but do not deal with scheduling strategies to promote load-balancing of tasks. The used strategy is to optimize a variety of parameters including local work group size and data transfer size. Based on a first preemption of execution time (using, for example, the GPU characteristics, like peak FLOPS) for the work items, it computes the average rate at which each device completes work items, and updates a running, weighted average.

3.5 Open Problems

Creating applications for heterogeneous parallel processing platforms is challenging since, as mentioned before, there are several processing models for parallel computers. The main two categories of processing models, also related here, are based on task decomposition and data decomposition. The approach of this thesis would be better classified on the task decomposition category, brought to a higher level. In this way, the load balancing arises as an important issue to deal with applications that are posed to heterogeneous architectures, like the one composed of CPU and GPU.

Overall, asymmetric multi-core designs associated with flexible and dynamic use of resources can offer grater potential speedups compared with other models, bringing naturally a considerable set of challenges. For example, assigning one task to each processing unit can lead to load imbalance. A load imbalance is characterized when one PU has more work than another and it is still executing when the other tasks have already finished. This leads to an inefficient use of the computational power of the underloaded processing units of the system.

Thereby, the load balance module must be responsible to assign an equal (or near uniform) amount of work to each processing unit, so they can process their workload in parallel, finishing the work at around the same time and, consequently, gaining in performance. This can be done statically or dynamically. Statically, the user can, for example, manage and take the responsibility to decompose his application.

Dynamically, the module is responsible to decompose at runtime and assign the tasks to be processed, being automatic and may consider some aspects just known after the execution starts. In order to make dynamic load-balancing efficient, a set of parameters must be evaluated and a great demand from the application is needed, i.e., it is useful in the scenario where there are a larger number of kernel invocations than the number of processing units for computing. For a even more efficient dynamic load-balancing, a profiler plays an important role on the process, since it can verify the PUs' usage as well as tasks' performance in an on line modus. Integrated to the load balancer, it can give the necessary feedback in order to produce rich information to perform more accurate online

reasoning and configuration (tuning) of tasks.

Based on the previous chapter and this state-of-art review, some open problems are identified:

- Lack of a dynamic load balancer for high-level approach targeting low-cost heterogeneous hardware, known as CPU and co-processors, like GPUs, Cell, FPGAs, and others. In other words, an automatic coarse-grained distribution of kernels over the asymmetric computing units;
- Lack of a consideration of runtime parameters to balance and tune the tasks, like the domain size of a problem to be solved;
- Lack of a performance study for iterative solvers for SLEs on the GPU;
- Lack of an integration between a runtime profiler and a load balancer to promote better strategies for distribution over the co-processors;
- Lack of an adaptation under new runtime conditions due to dynamic alterations, i.e., lack of a reconfiguration reasoning (done by the load balancer) in terms of task rescheduling for a new target computing unit;
- Lack of an answer for the question: in which processing unit an algorithm executes optimally (or better) under certain runtime execution platform conditions?

3.6 Chapter Remarks

The continuing demand for quality, by means of realism and precision, in high performance computing applications, like the engineering and other numerically-intensive workload applications, has significantly contributed to the appearance of novel computer architectures and software technologies tailored for such functionality. The CPUs became multi-core as well as powerful parallel hardware emerged, like the many-core GPU on its general purpose model.

Therefore, there is a need for studying the performance and scheduling of tasks oriented to this heterogeneous execution platform in a way to cover some of the technical open problems related on the last section. Based on related work, there is also a need to apply those concepts to solvers for SLEs or general tasks over the asymmetric CPU-GPU platform. The development of new strategies toward a dynamic load-balancing that considers workload distribution over a CPU-GPUs platform is proved as an important contribution.

3.6.1 Related work

The research investigated in this thesis will rely on the obtained results presented in the PhD thesis of Götz, namely the ability to perform an efficient scheduling over the CPU and a co-processor. However, it is not intended to use the FPGA as a co-processor (GÖTZ, 2007). Furthermore, this research presents completely different goals and the developed framework targets to achieve better performances for the used applications, instead of optimizing the FPGA area.

Two runtime frameworks, the Maestro (SPAFFORD; MEREDITH; VETTER, 2010) and the Patus (CHRISTEN; SCHENK; BURKHART, 2011), extend the OpenCL to promote mainly an automatic data transfer to the devices, that needed do be done explicitly by the programmer, and an automatic hardware-specific code generation applied to stencil computations. This thesis, instead, concentrates on load-balancing issues and scheduling of high-level OpenCL tasks and can be viewed as a complementary approach for these two solutions.

Regarding the scheduling strategy, the runtime system StarPU (AUGONNET et al., 2009) has similarities to the present thesis. Both works use a performance database to keep task performance history. However, their best scheduling technique is oriented to work with low-level tasks, like vector operations (TOPCUOGLU et al., 2002). The approach of Sm@rtConfig goes beyond and presents: new dynamic scheduling algorithms; its application for high-level tasks instead of low-level, i.e., it is focused on a whole algorithm in order to obtain knowledge whether it "fits" better to a specific PU under certain application and platform conditions; and a study of reconfiguration benefits, where a task can change its execution PU at runtime, enabling flexibility and adaptability under system changes. Further, their technique is based on codelets, while in this work the goal is to extend the OpenCL for the scheduling of high-level tasks.

The cited thesis of Brodtkorb also explored the scheduling of a group of dependent tasks on the GPU or CPU based on a pre-compilation phase that gathers a primary profile of execution times for basic tasks, like matrix-vector operations with fixed matrix sizes (BRODTKORB, 2010). This technique was also previously considered by Binotto et al. on the work (BINOTTO et al., 2010), and it is extended within this thesis based on scheduling algorithms that takes into account the execution platform conditions at runtime, i.e., it considers whether there are another tasks influencing the execution times that were not preempted a priori.

3.6.2 Closing remarks

The Sm@rtConfig framework, proposed in this thesis, applies new concepts that extend some related works and fulfill the analyzed gaps with respect to a framework that present a cost-effective dynamic assignment strategy for one CPU and one GPU and its generalization for multiple heterogeneous PUs using a *greedy* approach. This thesis also contributes with a performance analysis and tuning of iterative solvers over a CPU-GPU platform, in particular for the creation of a strategy for data access on the GPU and for choosing the most appropriate PU for processing a solver. The next chapters detail these contributions.

4 THE SM@RTCONFIG SYSTEM

A set of OpenCL kernels/tasks to be executed by multiple processing units requires the driver to perform dynamic load-balancing. The multiple available devices in the same context may have different capabilities, like the CPUs and the GPUs. Tasks that may run on one device may not run in the other device, and if they can be executed by both devices one may promote better performance than the other. In more complex and real scenarios, multiple OpenCL-based applications can be executed at the same time, interfering on the performances of each other. In other words, one task could not perform as expected in one processing unit and another PU could deliver better performances under such a runtime condition.

This load-balancing issue is nontrivial and OpenCL delivers the decision of choosing the processing unit to the programmer/application. Third party solutions still have to be developed for load-balancing and programmers still have to specify which device a kernel would execute on.

In a broad sense, this load-balancing strategy has the goal to automatically assign Units of Allocation (UA) over a CPU-Co-processors execution platform. The term UA is generically defined since the proposed framework is intended to deal with different granularities – the granularity is designed to change in accordance to the platform to be used – and different types of decomposition – task or data decomposition, according to application characteristics. At the presented framework stage, an UA is represented as a task, which is characterized (has the granularity) as iterative algorithms for solving systems of linear equations over the heterogeneous execution platform composed by the CPU and the GPU. Based on that, the framework will automatically choose the most appropriate PU to execute the tasks.

Starting with a *first assignment* just when the application starts, an online *profiler* monitors and stores tasks' execution times and characteristics in a *timing performance database*. During execution time, a *runtime assignment* is performed for new arriving tasks, considering the performance history stored in the database. A task that was assigned but is still not executed can then be reassigned to another PU if this change promotes a system performance gain.

Figure 4.1 illustrates this approach and the next sections present the details and functionalities of each of the three modules in red color.

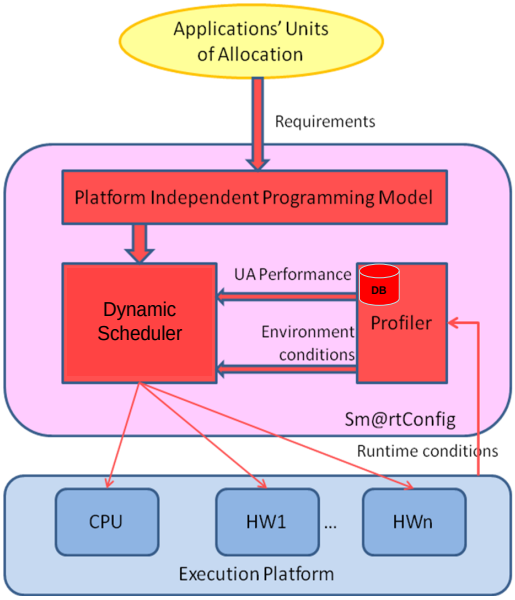


Figure 4.1: Overview of the proposed system

4.1 Platform Independent Programming Model

The proposed approach abstracts the PUs using the OpenCL API (STONE; GOHARA; SHI, 2010) as the *platform independent programming model*. Instead of working with *pragmas* (JIMéNEZ et al., 2009) or *codelets* (AUGONNET et al., 2009), the OpenCL was adopted since there are big efforts on making this API the standard for programming on heterogeneous PUs. The OpenCL tasks are characterized as new instantiations of an OpenCL kernel. The kernel should be programmed to explore the parallel cores of the processing units. Loops, for example, are automatically parallelized through the cores when parallelization is possible, like with the SIMD model. This way, using OpenCL instead of another method, like special and proprietary codelets, keeps the solution flexible and more useful and it can avoid extra commands, like pragmas, to indicate, following the example, a loop that can be parallelized. The OpenCL driver of each processing unit will, then, parallelize an OpenCL kernel according to the hardware features.

In its basic principle, the OpenCL encapsulates implementations of a task (methods, algorithms, parts of code, etc.) for different PUs, leveraging intrinsic hardware features and making them platform independent. Once the application developer statically assigns a task for a PU at programming level, a so-called *driver* translates the instructions coded in OpenCL to the PU language or intermediate code (the latest driver already supports the Intel and AMD CPUs as well as ATI and Nvidia GPUs). In case of Nvidia GPUs, for example, OpenCL converts code to CUDA. This way, tasks coded in OpenCL can be statically assigned at programming time to PUs (Compute Devices on its nomenclature), which are internally designed to be composed of Computing Units and Processing Elements.

The proposed strategies are applied to the OpenCL queue of kernels (tasks) to perform the task assignments dynamically at the device level and not internally for the computing units or the processing elements. Once a method, e.g., a solver for a system of linear

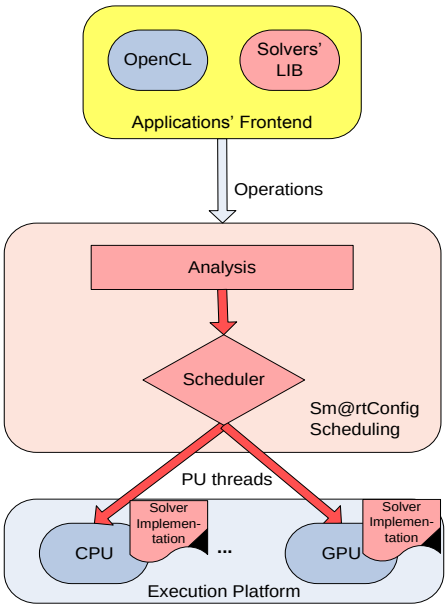


Figure 4.2: The frontend interface design

equations, is coded in OpenCL, the scheduler performs the assignment of the method to the most appropriate PU, i.e., to the PU that will execute the task in the minimal time. However, the solution of this problem is not trivial because a driver translates not only the OpenCL code to the PU language, but also the way of exploring the architecture of the processing unit. For example, the same method programmed to the CPU and to the GPU have different strategies to benefit from the hardware characteristics.

Moreover, related work studies show that the drivers are still not producing optimal code when compared to a method directly coded on the device programming language and exploring the PU features (KOMATSU et al., 2010; KARIMI; DICKSON; HAMZE, 2010). This issue, nevertheless, is being improved by the companies that produce the PUs and the drivers.

At the moment of writing this thesis, although OpenCL allows to access various processing units in a unified way, the code still needs to be further optimized for each of those devices to obtain better execution times. In order to partially solve this provisional problem, a solution is adopted that explores an automatic performance tuning methodology based on profiling to enhance the performance portability of OpenCL applications.

The Sm@rtConfig framework provides a library for three different solvers for systems of linear equations (SLEs) with different implementations of the same solver for each PU presented on the execution platform: CPU and GPU. These implementations will be presented in more details at the next chapters. When using a solver of the library, the proposed strategies will choose the processing unit for execution and, consequently, its encapsulated implementation after the assignment decision. Figure 4.2 depicts the interface, where the applications use the OpenCL API and the Sm@rtConfig library of encapsulated iterative solvers for SLEs. The Sm@rtConfig is responsible for analyzing its tasks and their schedule on the processing units, which will receive the appropriate implementation and the correct task parameters for execution.

This solution, by its specificity, does not compromise the generic methods and con-

cepts presented in the next sections, which could be further full integrated into the OpenCL API.

4.2 Profiler and Database (DB)

Time profiling is an analysis that considers several parameters at execution time that could be not known a priori, like the input data (size and type) and data transfers between PUs, among others. In this case, the Profiler executes at runtime and is focused on measuring the following aspects:

1. the task execution time,
2. the input data (problem size domain, in the case of the solvers) and type (int, float, double) to be processed by the task,
3. the data transfer time from the host to the PU, and
4. platform characteristics (number of processing units, computing units, processing elements).

The performance is measured using host (CPU) counting clocks, which intrinsically takes into account the data transfer times from/to the CPU to/from the PU, possible initialization and synchronization times on the PUs, and any latency. Additionally, the task execution time at the PU counting clocks is also measured. When a task is executed by the CPU, there will be no data transfer costs over the bus.

Given a first initialization of the Time Performance Database (DB), the profiler will update the database with real execution time performances, creating a history that represents the recent past execution of a task. This history will base the scheduler on finding an allocation for the tasks based on further balancing constraints. This first initialization is a *first estimation* of execution times and can represent a task performance reference in optimal conditions.

There are two ways of filling the database with a first estimation. The library provided by the Sm@rtConfig framework includes a performance profile of the solvers at specific processing units, containing an outline of which solver performs better in which processing unit according to the problem size domain. This profile is obtained using an idle execution platform and can deliver the best possible performance for the implemented algorithms, which are discussed on the next chapter. The other option is to select a set of problem size domains and execute the core of the solvers in a pre-precessing phase just when the application starts. This pre-processing phase will acquire more realistic execution times since it considers the current execution platform conditions. The discrepancy compared to the first estimation will thus be lower. Moreover, considering the generalization aspect of the whole strategy, this pre-processing solution could accomplish with a larger variety of tasks rather than just the ones with known performance profiles. On the other hand, it comes with the cost of the pre-processing computing.

The current implementation of the system is based on the first option to fulfill the database with the first estimation to facilitate the use of the proposed methods. Figure 4.3 exemplifies the database, where the number of unknowns to be processed by the solvers

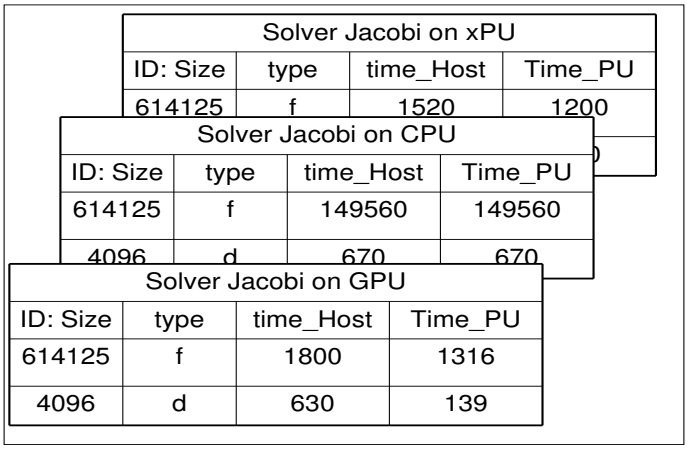


Figure 4.3: Performance history database: ID represents the task (number of unknowns for the SLE case study), type represents double or float, and time_Host and time_PU stores the last task execution time using the CPU and the PU clock time

(problem size domain) is used as the key for performing a history lookup. When there is no entry for a task with a specific domain size, the lookup function retrieves the data that represents the task with the most similar domain size.

In summary, the history database has two goals:

1. Expecting that the platform is being simultaneously used by several applications, to measure/store execution times, which brings implicitly the information whether the PU is idle or the system is being used by other applications, as well as the timing for data transfers;

2. To predict future allocation of tasks based on its recent past, i.e., assuming that similar high-level tasks are going to be executed several times, they tend to be executed by the same PU (promoting the locality concept).

4.3 Dynamic Scheduler

The core of the developed system is the Dynamic Scheduler which has the goal to assign the high-level tasks to the processing units that compose the execution platform considering load-balancing issues. All high-level tasks are independent and their dependency is characterized by the order or time instance that each task or set of tasks arrives at the scheduler queue, representing the workflow of the applications.

This module is composed of two phases that are executed dynamically. First, it establishes an initial scheduling estimation over the PUs just when the application (or several applications) starts, producing a set of tasks. This is described in Section 4.3.1. Second, for every new arriving task, it performs a scheduling considering the execution platform conditions, the tasks that were previously assigned but not executed, and the timing database. This is presented in Section 4.3.2.

Algorithm 1 First Assignment Heuristic

```
1: Initialize the timing database with costs acquired by a profiling benchmark
2: CostGPU[nTasks] // Initialize a vector of estimated costs on GPU
3: CostCPU[nTasks] // Initialize a vector of estimated costs on CPU
4: C=0; G=0 // Initialize CPU and GPU usage;
5: assigned[nTasks] // Initialize the assignment with -1
6: for i == 1 to nTasks do
7:    if G <= C then
8:       Find a Task i in CostGPU where assigned[i] == -1 and CostGPU[i] is minimum
9:       assigned[i] = 0 // 0 for GPU
10:      G += CostGPU[i]
11:   else
12:      Find a Task i in CostCPU where assigned[i] == -1 and CostCPU[i] is minimum
13:      assigned[i] = 1 // 1 for CPU
14:      C += CostCPU[i]
15:   end if
16: end for
17: Return assigned
```

4.3.1 First Assignment Phase - FAP

Given a set of tasks with predefined execution costs for the PUs (stored at the database), the first assignment phase performs a scheduling of tasks over the asymmetric PUs. In this sense, analogous to the work of Götz in 2007 (GÖTZ, 2007), a set of tasks $i = 1 \ldots n$ have an implementation x and an execution cost c acquired using a performance benchmark (the performance profile as mentioned in the last section) on each PU j. The assignment is then designed as follows:

- If $x_{i,j} = 0$, the task i is not allocated on the processor j and
- if $x_{i,j} = 1$ the task i is allocated on the processor j.

The best allocation is found using the objective function that maximizes the time performance of the tasks, i.e., has the lowest total execution time, where the execution time T_e is defined as:

$$T_e = \sum_{j=1}^{m} \sum_{i=1}^{n} x_{i,j} c_{i,j},$$

with the assignment variables $x_{i,j}$ the solution, *m* the number of PUs and *n* the number of considered tasks. For an execution platform composed of one CPU and one GPU, like in this first approach, *m* is 2.

Finding the optimal assignment is of NP-hard complexity, as mentioned before. With the goal to perform a balanced assignment with low overhead, Algorithm 1 is proposed with the goal to obtain an effective and low-cost assignment approach. This heuristic is applied on the first assignment for a platform composed of one CPU and one GPU. It starts intuitively assigning the tasks with the fastest performance. To maintain a certain load-balancing of tasks, it verifies the *usage* of each PU and assigns a task to the PU with less accumulated usage. The usage concept is not acquired using the processing units' API as there is currently no support for that on the GPUs, but it is a variable used by the

Algorithm 2 First Assignment with Swap Heuristic

```
1: X[nTasks]; nPairs = maximum number of pairs // Result obtained from Algorithm 1
2: Build M[nPairs][2] // Loop to find all possible pairs that could be swapped in X and
   set all as unlocked
3: gain = 0 // Initialize the gain
4: for k == 1 to nPairs do
5:    Find a pair o,p in M so that o and p are unlocked and gain is maximum;
6:    if gain > 0 then
7:       Swap o and p in X
8:       Lock o and p
9:    end if
10:   if gain < 0 or all pairs are locked then
11:      break
12:   end if
13: end for
14: Return X;
```

runtime system to control the allocations. The complexity of the algorithm is of O(n^2), where *n* is the number of tasks, as it searches for a task I with a minimal cost through a list of *n* tasks.

The load-balancing strategy has the goal to minimize the total execution time of the tasks to be executed on the heterogeneous PUs, This is, however, not achieved by the heuristic, since the proposed assignment does not take into consideration a global knowledge of the allocations, i.e., the assignment of a previous task could have been non optimal when considering the presence of a new task. This issue becomes even more relevant when dealing with a large set of tasks, and mainly with individual tasks that present a considerable performance difference on their execution costs over the PUs. This scenario is very common using the CPU and the GPU as part of the execution platform, where the same task executed on the GPU can be individually hundred times faster than being executed on the CPU.

Consequently, some improvements on the algorithm were performed, leading to Algorithm 2. The improvement is partially based on (LIN; KERNIGHAN, 1973; GÖTZ, 2007) and its main concept was shortly exposed in (BINOTTO et al., 2010). It performs an efficient swap on pairs on the *assignment* result provided by Algorithm 1 verifying if such swap can promote a gain in the total execution time of the tasks. The complexity of this improvement, presented in lines 2, 4, and 5, is of O(m^2), where *m* is the maximum number of pairs (in the worst case, m = (n/2)^2). The total complexity remains on the quadratic class O(n^2) because of the complexity presented by Algorithm 1. In practical cases, Chapter 6 will show that this overhead is minimum in comparison to the obtained performance gain.

Aiming to achieve better scheduling results, performing near to the optimal, and to diminish the algorithms' overhead, an improvement on the heuristics is further developed. The strategy used in Algorithm 1 is improved in Algorithm 3 by exploring an intrinsic characteristic of the system: the execution time of a task can be drastically different when executed at the CPU and at the GPU. This way, a vector of performance differences, in the case of one CPU and one GPU, was introduced. Following this concept, for each

Algorithm 3 First Assignment Considering Performance Differences Heuristic

```
1: Initialize the timing database with costs acquired by a pre-processing benchmark
2: CostGPU[nTasks] // Initialize a vector of estimated costs on GPU
3: CostCPU[nTasks] // Initialize a vector of estimated costs on CPU
4: C=0; G=0 // Initialize CPU and GPU usage
5: PerformanceDifferences[nTasks] // Initialize a list of cost differences where performanceDifferences[i] = CostCPU[i]-CostGPU[i]
6: Sort(PerformanceDifferences) // Sort the vector of differences ordering from the smallest to the biggest difference
7: assigned[nTasks] // Initialize the assignment with -1
8: for i == 1 to nTasks do
9:    if G <= C then
10:      Take the Task i in PerformanceDifferences with the grater difference (last one)
11:      assigned[i] = 0 // 0 for GPU
12:      G += CostGPU[i]
13:      Remove I from PerformanceDifferences
14:   else
15:      Take the Task i in PerformanceDifferences with the smaller difference (first one)
16:      assigned[i] = 1 // 1 for CPU
17:      C += CostCPU[i]
18:      Remove I from PerformanceDifferences
19:   end if
20: end for
21: Return assigned
```

Algorithm 4 First Assignment Considering Performance Differences and Swap Heuristic

```
1: X[nTasks] // Result obtained from Algorithm 3
2: gain = 0 // Initialize the gain
3: Find a single Task I that if switched to the other PU, it yields a maximum gain
4: Swap the allocation of I in X
5: Return X;
```

task an auxiliary structure stores the difference of performances on both PUs in an ordered fashion. This proposed strategy drastically reduces the scheduling overhead and promotes a significant improvement on the scheduling results, points that are verified in the detailed analysis presented in Chapter 6. It presents a quasilinear complexity of O($nlogn$), since there is a loop over the *n* tasks to identify a task I by using a binary search.

Consequently, taking advantage of such sorting of performance differences, Algorithm 4 improves the strategy presented in Algorithm 2 by swapping just the tasks with similar performances. This situation depends on the input data not known a priori, but does not appear frequently due to the PUs and tasks' characteristics, leading to low overhead in practical cases and maintaining the algorithm complexity.

It is important to note that the Algorithms 1 - 4 consider a system of one CPU and one GPU. However, internally, the CPU can be composed of multiple cores (example: 2, 4, 6, 8) and the GPU of many cores (example: 112, 240, 480). By the point-of-view of the Sm@rtConfig system, which is developed to explore the scheduling of high-level algorithms oriented on a processing unit level, the code translated by the OpenCL

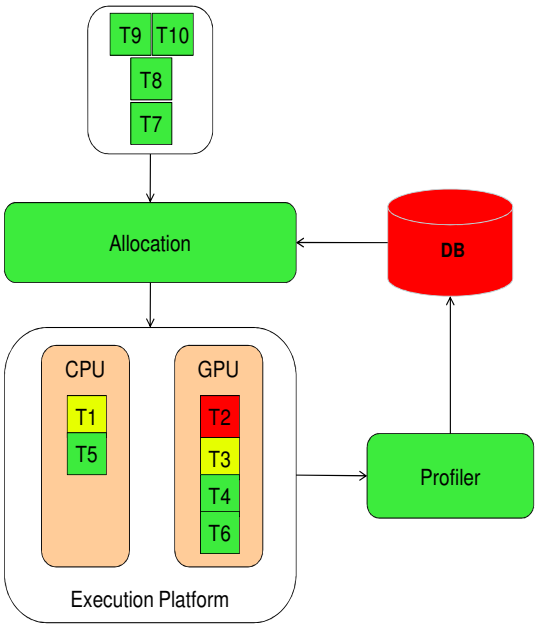


Figure 4.4: Dynamic arrival of new tasks at system execution time (green: tasks to be executed; yellow: tasks in execution; red: executed tasks)

implementation has to be optimized in order to obtain the benefit from the internal cores and from the hierarchies of the processing units cache memories. As mentioned before, it is difficult to achieve such effectiveness as drivers are still not effective. The used strategy was to tune the code implemented in OpenCL by using specific implementations of the tasks presented as a Sm@rtConfig library that supports a CPU/GPU execution platform (BINOTTO et al., 2010).

4.3.2 Runtime Assignment Phase - RAP

After the first assignment phase, i.e., during system execution, the arrival of new tasks to be scheduled by the system is modeled as a First In First Out (FIFO) queue. For a better illustration, Figure 4.4 depicts the scenario. As observed in the figure, new and non-processed tasks are the green ones, the yellow tasks are under execution status, and the red ones have already executed. A new task is designed to arrive solely or grouped in a new set. For each new green task, a dynamic scheduling strategy should be performed considering the current execution platform conditions and tasks' input data characteristics.

The scenario of this runtime assignment phase differs from the FAP phase, presented in the last subsection, which has the goal to find a close-optimal assignment for a set of tasks just when the applications start. In the RAP phase, the goal is slightly different. The objective is to *dynamically* assign a task on its arrival, finding the most appropriated processing unit just by considering tasks that were already assigned but still not executed.

Additionally, the presented algorithms for the FAP are designed towards a platform composed of just two processing units: one CPU and one GPU. This issue can represent a drawback for current desktop platforms. Thus, although the depicted heuristics could be extended to attend multiple processing units (which are not in a large number, since the

Algorithm 5 Earliest First Termination - EFT

```
1: Use the timing database, which presents costs acquired by a pre-processing benchmark or updated costs
2: CostPU[pu][nTasks] // Initialize a vector of estimated costs for every PU
3: Usage[pu] // Initialize the PUs usage with 0 or calculate the usage based on assigned tasks
4: assigned[nTasks] // Initialize the assignment with -1
5: for i == 1 to nTasks do
6:    for j == 1 to pu do
7:       Usage[j] += CostPU[j][i]
8:    end for
9:    Find the minimum cost minCost on Usage
10:   for j == 1 to pu do
11:      if minCost == Usage[j] then
12:         assigned[i] = 1 // Allocate to the PU with minimum usage
13:      else
14:         Usage[j] -= CostPU[j][i]
15:      end if
16:   end for
17: end for
18: Return assigned
```

focus is oriented to a desktop platform), a generic *greedy* technique, called here Earliest First Termination (EFT), was developed to accomplish with the assignment of tasks over several processing devices.

Based on the thesis' findings (FREITAS et al., 2008a) and with the support of the performance database, Algorithm 5 performs the assignment of a new task obeying the following rule: the task will be scheduled to the PU that

1. will finish its pending tasks(s) at the earliest time and

2. that can terminate the execution of this "under assignment" task at first.

In other words, the task is scheduled to the PU that will take less time to execute the task to be scheduled. This calculation is performed using the stored execution times at the performance database. With the values that represent the estimation of recent execution costs for tasks indexed according the problem size domains, Algorithm 5 calculates the equations that represent the rules 1 and 2 in a straightforward way.

One important characteristic of Algorithm 5 is its sufficiency to consider the scheduling over several processing units. The algorithm has a O($m \times n$) complexity, where *m*, in this case, is the number of PUs, and it is generic to deal with any number of processing units and overcomes the limitation of the other strategies that deal with two processing units. It can also be used at the FAP phase, but, for comparison purposes, the scheduling module of Sm@rtConfig was designed to use Algorithms 3 and 4 for the FAP and Algorithm 5 for the RAP. Chapter 6 presents a comparison of all algorithms, showing that Algorithms 3 and 4 delivered a slightly better result than Algorithm 5 and considerably improve the scheduling result of Algorithms 1 and 2.

Moreover, to assure the consideration of some changes on the platform conditions at runtime (e.g., if a task is taking much more time to be executed than its database expectation), the RAP was implemented with a feature called *assignment reconfiguration*. Using this feature, previous tasks that were already assigned but not executed are resubmitted for scheduling by their insertion on the FIFO queue as the first to be scheduled. This way, new tasks can be scheduled together with previous non-executed tasks and, in some cases, there will occur the case when those previous tasks change their original assignment if the change promotes a system performance gain. The evaluation of this feature is shown in Chapter 6 as well.

Lastly, it is important to emphasize once more that the timing database is updated on every task finalization with its new execution cost acquired at runtime. For every task, there will be an entry that represents the problem size domain and its real execution cost on the processing units.

4.4 Chapter Remarks: gaining performance with a load-balancing approach

Contrary to most of the related work that schedule computation at a low-level task granularity, this research proposes to select the most appropriate processing unit to execute a high-level algorithm. This way, the problem of scheduling tasks on a high-programming level for a desktop platform consisting of CPUs and GPUs is tackled. The solution comes to partially answer the question of which processing unit is the best device to execute an algorithm under certain runtime conditions. This way, the goal of performing a dynamic scheduling of tasks onto both types of computing units is to obtain a better performance in comparison with static and programming-time-based scheduling. It shows possibilities to extend the OpenCL architecture by accomplishing with automatic scheduling of high-level kernels under certain execution platform conditions and kernels' input data.

For that, a profiling and scheduling module is designed with the support of a history database. The core of the system is based on the scheduling strategies that take into account the recent past of tasks' execution costs. Firstly, a number of four heuristics were proposed to assign high-level tasks over the CPU and the GPU. However, these strategies contained a drawback to work with more than two PUs, which could compromise the use of the system on modern desktop platforms that work with, for example, two GPUs and one CPU at the same time. This issue was solved with the proposal of the EFT heuristic which generalizes the scheduling module for several processing units.

The proposed methods have been implemented under the OpenCL concept, keeping the methods as platform independent as possible. A full integration of Sm@rtConfig and the OpenCL API is still needed towards a complete platform independence. This need can also generate an unique platform to code applications without having to design which PU will be used under certain conditions and to exclude the programmer responsibility for managing the data over PUs' memories.

Finally, from the proposed system point-of-view, the exploitation of the internal cores of a processing unit is reflected on the tasks' performance measurement, making it transparent to the scheduler. The responsibility of exploring the internal cores of a PU is given

to the implementation of the task. In this sense, the Sm@rtConfig system provides a library with different implementations of the same task targeting, specially, the CPU and the GPU. One could argue how well these different implementations are optimized in comparison to the same OpenCL code on different processing units, but the main goal of the system is not to deliver a library with the most optimal implementation for a PU. Instead, the aim is to perform the most efficient load-balancing of high-level algorithms by means of assigning the available implementations over the available processing units to obtain a system performance maximization.

The next Chapter presents the used case study: iterative solvers for systems of linear equations. It provides details on how the solvers were implemented, specially on the GPU. These implementations are used to build the Sm@rtConfig solvers' library and to evaluate the efficiency of the proposed scheduling strategies, which is presented in Chapter 6.

5 APPLICATION: ITERATIVE SOLVERS FOR SLES APPLIED TO A REAL-TIME 3D CFD SIMULATION

The case study selected for experimental validation of the proposed dynamic scheduling methods is motivated by a research project focused on a 3D Computational Fluid Dynamics simulation applied to a wind tunnel scenario with real-time performance and interactive geometry modification requirements. For this purpose, a CFD framework is being developed at the Fraunhofer Institute for Visual Computing Research (IGD). The proposed workflow consists of the four sequential steps outlined in Figure 5.1. An analysis of computational costs of this workflow yields the following observations: The first phase performs a simple vector addition operation by applying an external force on the particles over time instances. In the second phase, a particle tracer is used to calculate the convection (or advection) effect of the fluid. This is done at each time step by moving the particle according to the velocity of the previous particle location. Both phases are mathematically simple and with low processing cost.

On the contrary, the last two phases require a big computational effort, as they involve the calculation of a system of linear equations (SLE). The diffusion phase computes the fluid viscosity along time, needing a discretization step that leads to a sparse linear system for the unknown fields. The same applies for the *Poisson* problem resulting from the projection phase to calculate, for example, the fluid pressure. Therefore, both the projection and viscosity steps involve the solution of a large sparse system of equations that needs an efficient and accurate solver with fast convergence.

In this chapter the CFD background and iterative solvers used on it are discussed, as well as the proposed GPU implementation. It briefly summarizes the mathematical concepts before presenting a GPU approach to accelerate such computations.

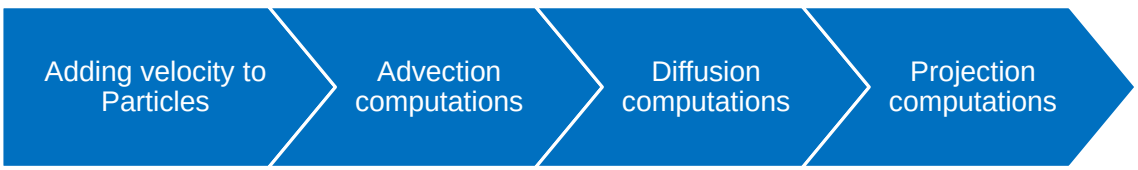


Figure 5.1: The CFD workflow

5.1 Introduction to Iterative Solvers for SLEs

This section introduces the mathematical concepts that result on a sparse system of SLEs applied to the CFD application. It is followed by a short description of the methods of Jacobi, Gauss-Seidel, and Conjugate Gradient, that are commonly used for solving these systems.

5.1.1 The System of Linear Equations

The most time consuming part of the CFD approach is solving SLEs in the so-called diffusion and projection phase. They depend directly on the problem domain size and for large domains this becomes very expensive in terms of execution time.

To determine the velocity and pressure fields around objects like planes, a simple setup for a fluid simulator with the *incompressible Euler equation* is used (FEDKIW; STAM; JENSEN, 2001):

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \mathbf{v}\nabla^2\mathbf{u} + \mathbf{f} \tag{5.1}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{5.2}$$

with velocity $\mathbf{u}$, pressure p, viscosity v, external forces $\mathbf{f}$, the vector of partial derivatives ∇ (where $\nabla = (\frac{\partial}{\partial \mathbf{x}}, \frac{\partial}{\partial \mathbf{y}}, \frac{\partial}{\partial \mathbf{z}})$ for three dimensions), and density ρ, as extensively described in (STAM, 1999), (CRANE; LLAMAS; TARIQ, ????), and (BRIDSON, 2008), to mention only some. Assuming a constant density, $\rho = 1$ can be set without loss of generality for the computational issues, as this is only a scaling of the pressure. As the effects of viscosity are negligible in gases (on coarse grids, numerical dissipation prevail physical viscosity and molecular diffusion) (FEDKIW; STAM; JENSEN, 2001), like in an wind tunnel, $\mathbf{v}\nabla^2\mathbf{u} = 0$ can be analogously defined. These equations represents the phenomena when the velocity conserves both mass and velocity, zero divergence, maintaining a constant density. The discrete version of Equation (5.1) is given by

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} = -(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n - \nabla p^{n+1} + \mathbf{f}. \tag{5.3}$$

Equation (5.3) is decomposed by operator splitting to be calculated, i.e., the advection, pressure correction, and external forces are computed apart from each other. In the operator splitting method, to advance the solution in time, a sequence of stages is solved and each stage involves only one operator. In other words, each component that composes the sum of the right hand side on Equation (5.3) is computed in each stage. The left hand side is decomposed by using $\mathbf{u}^{n+1} - \mathbf{u}^n = \mathbf{u}^{n+1} - \hat{\mathbf{u}} + \hat{\mathbf{u}} - \mathbf{u}^n$.

In the first step, an intermediate velocity field $\hat{\mathbf{u}}$ is computed by solving the Equation (5.3) over a timestep Δt without the pressure operator. In this stage, an intermediate velocity that does not satisfy the incompressibility constraint is computed from a time step n as follows:

$$\frac{\hat{\mathbf{u}} - \mathbf{u}^n}{\Delta t} = -(\mathbf{u}^n \cdot \nabla)\mathbf{u}^n + \mathbf{f}, \tag{5.4}$$

where $\mathbf{u}^n$ is the known velocity at n^{th} time level.

Secondly, in the projection step, the pressure field at time level $n+1$ needs to be computed, after which the updated velocity field is obtained by solving

$$\frac{\mathbf{u}^{n+1} - \hat{\mathbf{u}}}{\Delta t} = -\nabla p^{n+1}. \tag{5.5}$$

The pressure is computed by projecting the left hand side of Equation (5.5) onto a space of divergence-free velocity fields to get the next update (n+1) of the velocity, i.e, $\mathbf{u}^{n+1}$, given a pressure p^{n+1}. This way, to proceed on solving Equation (5.5), the following method is applied. In order to make the fluid given by $\hat{\mathbf{u}}$ incompressible, it should be projected to an incompressible flow $\mathbf{u}^{n+1}$. For that, a projection method (CHORIN, 1997) is used, which is equivalent to compute the pressure using a *Poisson equation* (STAM, 1999). This is valid following the *Helmholtz-Hodge decomposition*, which states that any vector field $\hat{\mathbf{w}}$ can be decomposed into the form:

$$\hat{\mathbf{w}} = \mathbf{w} + \nabla q, \tag{5.6}$$

where $\mathbf{w}$ has zero divergence by making $\nabla \cdot \mathbf{w} = 0$ and where q is a scalar field. In addition, any vector field is the sum of a mass conserving field and a gradient field, leading to the establishment of an operator $\mathbf{P}$ that projects $\hat{\mathbf{w}}$ onto its divergence free part $\mathbf{w} = P\hat{\mathbf{w}}$. The operator is then defined by multiplying both sides of Equation (5.6) by ∇:

$$\nabla \cdot \hat{\mathbf{w}} = \nabla^2 q. \tag{5.7}$$

Analogously, forcing the vector field $\hat{\mathbf{u}}$ to be incompressible is equivalent to compute the pressure of Equation (5.5) rewritten now in the form of the Equation (5.6):

$$\hat{\mathbf{u}} = \mathbf{u}^{n+1} + \Delta t \nabla p^{n+1} \tag{5.8}$$

By solving this equation over the timestep Δt and applying the analogy of Equation (5.7) with the zero divergence constraint, the pressure is computed by solving the following Poisson equation

$$\nabla^2 p^{n+1} = \frac{1}{\Delta t} \nabla \cdot \hat{\mathbf{u}} \tag{5.9}$$

with homogeneous *Neumann* boundary conditions $\frac{\partial p}{\partial \mathbf{n}} = 0$, where *n* is the normal for the boundary point.

Thus, the intermediate velocity is made incompressible by subtracting the gradient of the pressure from the velocity field itself by rearranging the Equation (5.5). In Equation (5.4) $\hat{\mathbf{u}}$ is computed and in Equation (5.9) ∇p^{n+1} is computed to calculate the updated velocity:

$$\mathbf{u}^{n+1} = \hat{\mathbf{u}} - \Delta t \nabla p^{n+1}. \tag{5.10}$$

Based on this introduction, a focus is given to the most timing consuming part, which is the Poisson equation (5.9). This equation can be calculated using a numerical method

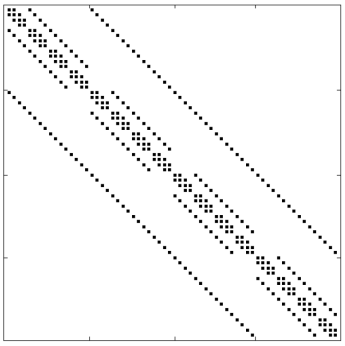


Figure 5.2: Sparsity pattern of a finite difference discretization in three dimensions

for improving intermediate solutions. Using this approach, a simple spatial discretization of Equation (5.9) results in a large system of linear equations over each timestep:

$$A\mathbf{x} = \mathbf{b}, \tag{5.11}$$

where $\boldsymbol{A}$ is the matrix of coefficients related to the derivative operations (∇^2), $\mathbf{b}$ is the vector related to $\frac{1}{\Delta t}\nabla \cdot \hat{\mathbf{u}}$, and $\mathbf{x}$ is the vector of unknowns to be solved: p^{n+1}. The dimension depends on the number of degrees of freedom.

Using a regular *Cartesian* grid and approximating the derivatives by finite differences, it leads to a sparsity pattern of $\boldsymbol{A}$, shown in Figure 5.2 and known as the *7-point Laplacian*. The system matrix $\boldsymbol{A}$ has some specificities, being sparse, positive semi-definite, and symmetric (BRIDSON, 2008).

5.1.2 Iterative methods for solving sparse SLEs

There are several choices for computing or approximating the solution of Equation (5.11). Direct methods are not appropriate because of the huge dimension, which is $n \times n$, with n being the number of unknowns (equations in the linear system). Factorization methods are also not suitable since they are focused on dense matrices and cannot handle the sparsity efficiently. In computational fluid dynamics, the governing equations are nonlinear and the number of unknown variables is typically sparse and very large. Under these conditions, implicitly formulated equations are almost always solved using iterative techniques.

Therefore, three implicit iterative methods are analyzed: Jacobi, Gauss-Seidel, and Conjugate Gradient. All these methods compute the Equation (5.11) on the form

$$b_i = \sum_{j=1}^{n} A_{ij} x_j, \tag{5.12}$$

which is the most time consuming part of the methods (BUATOIS; CAUMON; LÉVY, 2007).

The iterations are used to advance the partial solution through a sequence of steps from a starting state to a final converged state, allowing a larger timestep (in comparison to explicit methods) and promoting the unconditional stability (because the number of

iterations required for a solution is often much smaller than the number of time steps needed).

Bellow, a brief overview of the methods is given. For detailed information, including pseudo-code descriptions, one is referred to the work of Barrett et al. (BARRETT et al., 1994).

5.1.2.1 The Jacobi Method

The Jacobi method is based on solving for every variable locally with respect to the other variables. The resulting method is relatively easy to implement.It is done for the *i*th Equation (5.12), where it is solved for the value x_i while assuming that the other entries of *x* remain fixed. Then, the method iteratively improves an initial estimation $\mathbf{x}^0$ for the SLE. For one complete iteration, given $x_i^{(m)}$, the next approximation of the solution $x_i^{(m+1)}$ is computed. By rearranging and isolating each equation of the SLE, the method is obtained:

$$x_i^{(m+1)} = \frac{1}{A_{ii}} \left[b_i - \sum_{j=1, j \neq i}^{n} A_{ij} x_j^{(m)} \right] \qquad i = 1, \ldots, n. \tag{5.13}$$

As the system matrix A has the regular pattern as depicted in Figure 5.2, the sum consists of only six values. The iteration needs two vectors, $\mathbf{x}^{(m+1)}$ and $\mathbf{x}^{(m)}$, for storing the values before and after each iteration. Additionally, the convergence of this iterative method is considered slow.

5.1.2.2 The Gauss-Seidel Method

The Gauss-Seidel method is similar to the Jacobi method, except that it uses updated values as soon as they are available. In general, if the Jacobi method converges, the Gauss-Seidel method will converge faster than the Jacobi method, though still relatively slowly.

In contrast to the Jacobi method, the Gauss-Seidel method uses the values computed beforehand for advancing to a new approximation:

$$x_i^{(m+1)} = \frac{1}{A_{ii}} \left[b_i - \sum_{j=i+1}^{n} A_{ij} x_j^{(m)} - \sum_{j=1}^{i-1} A_{ij} x_j^{(m+1)} \right] \quad , \tag{5.14}$$

where $i = 1, \ldots, n$.

Thus, the sum operation is split into two components containing the old and the new values. One advantage is that only one vector $\mathbf{x}$ is needed, keeping both old and new results. The common Jacobi method works with two $\mathbf{x}$ vectors, one for the current values and one where the new results of the iteration are stored. The Gauss-Seidel strategy improves the convergence since the vector's current values are calculated based on previous timesteps. However, data dependency arises and because each computation needs the newly calculated values, an internal parallelization of the method is inapplicable.

This way, the Gauss-Seidel method is a derivation from the Jacobi method, but, unlike the Jacobi, the computations for each element cannot be done in parallel. Both methods have no restriction concerning the order in which the Equation (5.12) is solved. An

ordinary approach is a lexicographical ordering, i.e., the equations are computed as the unknowns are stored. A specific reordering of those equations removes the data dependency in the iteration and makes it applicable for parallelization.

The *Red-Black Gauss-Seidel* iteration divides the unknowns in a *red* and a *black* set in a way that all neighbors of a red cell is black and vice versa. As a consequence of this classification, the computation of one type of cells only needs the other type as input. Thus, one complete iteration is splitted into an even (red) and an odd (black) iteration, which processes, respectively, the equations: $x_{2i}^{(m+1)}$ and $x_{2i+1}^{(m+1)}$.

The red-black version of the Gauss-Seidel is then used along this work.

5.1.2.3 The Conjugate Gradient Method

The method of Conjugate Gradient (CG) is an effective method when the coefficient matrix is symmetric positive definite, since the storage for only a limited number of vectors is required. In every iteration, two inner products are performed in order to compute updated scalars that are defined to make the sequences satisfy certain orthogonality conditions. On a symmetric positive definite linear system, these conditions imply that the distance to the true solution is minimized.

The Conjugate Gradient generates a sequence of conjugate (or orthogonal) vectors. It combines the ideas of *gradient descent* and the method of *conjugate directions* in two steps. Step one finds the gradient direction and step two determines how large the step should be in order to get as fastest to the optimum.

The first step minimizes the function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b} \tag{5.15}$$

iteratively by using a direction, which reduces the error optimally in one iteration. This results in solving Equation (5.11) in the case *A* is positive definite ($\mathbf{x}^T A\mathbf{x} > 0$ for all non-zero vectors $\mathbf{x}$) and symmetric ($A^T = A$). When the function becomes smaller in an iteration, the final solution is closer to be found (converged).

The second step uses conjugate search directions, which performs the same calculations for a direction perpendicular to the previous ones in order to optimally exploit the space of search. The combination of these two approaches leads to the conjugate gradient algorithm, which minimizes the distance to the true solution in each iteration by finding the nearest local minimum (detailed information can be found in (SHEWCHUK, 1994)).

In overall, the most time consuming part of the algorithm is the inner product $\mathbf{x}^T A\mathbf{x}$, composed. like the other described solvers, of a matrix-vector operation.

5.2 Related Work on Solvers for SLEs using the CPU and the GPU

A number of works has contributed with strategies to solve SLEs approaching the GPU. Buatois, Caumon and Lévy presented a symmetric sparse system solver and compared its performance on CPUs and GPUs (BUATOIS; CAUMON; LÉVY, 2007), strategy also followed by Bell and Garland, but with a deep analysis of several formats for sparse matrix-vector multiplications (BELL; GARLAND, 2009). The performance of the

solvers were compared on CPUs and GPUs and there were situations where some formats presented better performance on the CPU, depending mainly on the problem size domain and on the strategy for data storage.

Krüger and Westermann (KRÜGER; WESTERMANN, 2005) provided data structures and operators for a linear algebra toolbox on the GPU for the Conjugate Gradient algorithm. Based on that, Bolz et al. presented an application that makes use of this toolbox oriented to problems on unstructured grids, extending the work with the Multigrid solver for regular grids (BOLZ et al., 2005). Both approaches used *shaders*, instead of GPGPU functionalities, for programming the graphics pipeline and textures for data storage.

The authors of (ZHANG; COHEN; OWENS, 2010) recently presented three parallel algorithms for solving *tridiagonal* linear systems on a GPU using its shared memory (GPGPU), obtaining a 12 times speedup compared to a multi-threaded CPU solver. Moreover, Volkov and Demmel (VOLKOV; DEMMEL, 2008) presented a performance benchmarking of linear algebra algorithms implemented on GPUs and its comparison to CPUs, mentioning that a hybrid architecture is more appropriate even if the GPU performance power outperforms the CPU in several circumstances. This conclusion is also similar to the aforementioned work of Hill and Marty (HILL; MARTY, 2008).

Introducing multiple GPUs, Cevahir, Nukuda, and Matsuoka described a method for Conjugate Gradient, obtaining fast results when working with data decomposition (CEVAHIR; NUKADA; MATSUOKA, 2009). Following, Ament et al. improved the work with a parallel pre-conditioner that outperformed classical ones in the CPU and the GPU, like over-relaxation, targeting the GPU (AMENT et al., 2010).

Additionally, Göddeke et al. presented a performance comparison with a static domain size partition to be computed by the CPU-GPU platform, depicting the need of a dynamic load-balancing (GÖDDEKE et al., 2009). Those findings resulted on a PhD work specifically designed for calculating the finite-element Multigrid solvers for partial differential equations simulations on GPU clusters (GÖDDEKE, 2010).

Based on the described works, a correlation with this thesis can be done based on two works that explore linear algebra using the GPU (TOMOV et al., 2010; GÖDDEKE et al., 2009). This research differs from the cited works on the techniques for achieving memory coalescing and identifying brake-even points where the GPU promotes better performances than CPUs for iterative solvers for SLEs applied to sparse matrices. The related works concentrate, however, on factorization-based solvers for dense matrices, but they showed that there is still research needed to, for example, directly compare the performances of SLEs' solvers based on a CPU-GPUs execution platform.

5.3 Implementing Iterative Solvers on the GPU platform

For the presented CFD application, solving the sparse SLEs is the most expensive, i.e., time consuming, part of the system. The use of a GPU approach for solving the SLEs is one of the strategies utilized to accomplish the real-time computation requirement of the application. In this scope, the three different iterative solvers for SLEs are implemented to be executed on a scenario composed of a CPU and multiple GPUs. The solvers represent the high-level tasks that are to be further submitted to the load-balancing procedure.

In this section, a novel implementation of the algorithms on the GPU is presented. It explores the shared memory – or local memory using the OpenCL nomenclature – and the architecture provided by CUDA 2.0 of Nvidia. It is important to make a note that another kind of GPU cache could be used, like the texture cache. The authors of (COHEN; TARIQ; GREEN, 2010), for example, employed such computer graphics-based approach, but focused on a fluid particle simulation. However, this strategy leads to an extra method to organize the data, since 3D textures are read-only. In this work, the GPU using the shared memory cache is explored, like the other GPGPU related approaches presented in Chapter 3.

5.3.1 Four Concerns to be Met Towards an Efficient GPU Implementation

A focus is given on the GPU approach for calculating Equation (5.12), which is the core of the three described solvers, exploring the characteristics of the system matrix $\boldsymbol{A}$ (positive definite and symmetric) that need to be efficiently implemented on the GPUs (JOST; CONTASSOT-VIVIER; VIALLE, 2009). This way, in order to obtain the full computational power provided by the GPUs, four requirements have to be met:

1. Global memory access has to be coalesced.

2. Multiple access to global memory should be buffered in shared memory.

3. Data should be available independently of the thread.

4. Branching of threads in one block should be avoided.

In order to meet the last requirement, a buffer of zeroes is introduced to the matrix, ensuring that no illegal data access may occur, while obliterating the need for boundary queries. Although this approach has already been proposed (THIBAULT; SENOCAK, 2009), it was not mentioned how the size of the buffer has to be adjusted for different problem domains. This size is crucial to meet the requirements for an aligned starting address in order to enable coalesced access to the data. The same holds for the number of threads in one CUDA block.

In addition, this approach uses ghost cells (padded area) only in "front" and "behind" of the problem domain and no additional ghost cells between the layers are introduced, reducing the memory load and simplifying computations. The ghost cells, in this case, are not used for any kind of boundary conditions (as opposed to (THIBAULT; SENOCAK, 2009)) and serve for the unique purpose of avoiding illegal memory accesses. The implementation of boundary conditions is achieved by a modification of the matrix $\boldsymbol{A}$, where a zero is stored in those entries that would lead to a multiplication of a border element of the domain by a cell from another control volume in a different layer or line. This meets the third requirement.

To fulfill the first two requirements, the use of the GPU shared memory is also improved from the related work (THIBAULT; SENOCAK, 2009), since shared memory delivers better performance for the cases where the same data has to be accessed multiple times or with the goal to allow coalesced loading/writing of the data. Basically, the GPU, just as CPU, has several layers of memory that differ in size and bandwidth as mentioned in the Section 2.2 of Chapter 2.

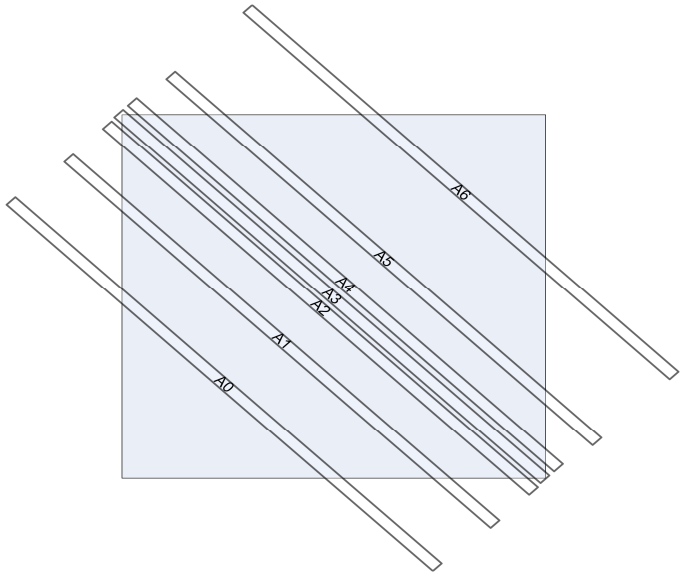


Figure 5.3: Representation by seven linear vectors (A0-A6) of the resulting matrix from a regular grid

The shared memory of the GPU may be compared to the cache of a CPU in terms of speed and size. However, as opposed to the CPU cache, the GPU shared memory is not automatically managed, being this task - to take advantage of the higher bandwidth offered by the shared memory - explicitly performed by the developer. For GPUs, access to multiple elements, should be performed in an aligned way and stored in a consecutive profile, resulting in much higher bandwidths than accesses to elements that are either not aligned or not stored consecutively. This is described as *coalesced access* (NVIDIA, 2010a).

Therefore, this access should be coalesced in order to obtain the best performance of the GPU. Thus, even when data is only accessed once, by a single thread, it may be advantageous to first load data into the shared memory, using a coalesced access pattern. In this work, instead of using the shared memory for coalesced loading of the matrix $\boldsymbol{A}$, the adapted way in which $\boldsymbol{A}$ is internally represented allows coalesced access. This enables the use of larger CUDA block sizes (work-group sizes, in OpenCL) and the use of latency hiding mechanisms, which both improve the speed of computations.

5.3.2 The Matrix-vector Multiplication on the GPU

The proposed implementation uses only 5 rows of the vector $\mathbf{x}$ plus the padding that are loaded into shared memory, allowing that multiple threads access the same cells and increasing the limit for the CUDA block size. Additionally, to enable latency hiding mechanisms presented in CUDA, the implementation works with batches always on the limit of shared memory, allowing the GPU to switch to another block while the current block is waiting for data. This approach works when shared memory size accommodates multiple CUDA blocks and no other data is loaded. In combination with the coalesced loading model presented here it leads to a significant speedup of computations, as idle time is prevented.

The coalesced access to the main memory is achieved by splitting the matrix $\boldsymbol{A}$ in 7 vectors (**A0** to **A6**) as in Figure 5.3 and ensuring that the starting addresses for data access in a kernel will always be aligned.

Figure 5.4 shows how the coalescing strategy is developed. Starting from a naive approach where each thread (or work-item in OpenCL) would have to read/write seven continuous entries and would therefore not fulfill the requirements for achieving coalesced loading (Figure 5.4(a)). Thus, a storing/loading strategy is developed, where a series of threads would access a continuous part of the global memory in an ordered fashion (Figure 5.4(b)). However, each thread would not have access to its respective entries. Taking as example threads T1 and T2, using shared memory, T1 may load data that will only be processed by T2 and not by T1 itself, being a typical example of stride memory access. This method almost performs coalesced access, but will not work on first generation Nvidia CUDA capable devices since consecutive segments of each block starts at the

$$7 * blocksize * size\_of\_float \tag{5.16}$$

position, allowing only threads in the very first block to achieve coalesced loading. For further blocks, the starting address is not necessarily a multiple of the block size (128). To overcome this issue, the matrix was decomposed into seven single vectors, ensuring that the starting addresses for data access in a kernel will always be aligned and allowing coalesced access on every Nvidia CUDA device (Figure 5.4(c)).

For such an implementation, the lexicographic ordering $i = 0, \ldots, n$ is replaced by a component-wise representation (i, j, k) with $i = 0, \ldots, n_x, j = 0, \ldots, n_y, k = 0, \ldots, n_z, n = n_x \cdot n_y \cdot n_z$, which accounts for the three dimensional setting. In this representation, the neighbors of one cell (i, j, k) are $(i \pm 1, j, k)$, $(i, j \pm 1, k)$, and $(i, j, k \pm 1)$. This representation is a base for the so-called *stencil computation*, illustrated by Figure 5.5.

In CUDA, threads are numbered in a specific disjoint pattern, being allowed to construct consecutive indices ix (the same applies using the OpenCL API). Here, one ix represents the ix-th equation of the SLE and simultaneously implies a position (i, j, k) in the simulation domain

$$ix = k \cdot n_x \cdot n_y + j \cdot n_x + i. \tag{5.17}$$

The vector of unknowns $\mathbf{x}$ and the right hand side $\mathbf{b}$ of Equation (5.11) represent the position and are simply stored in the linear pattern. The system matrix $\boldsymbol{A}$ can be represented by the seven vectors of length $n_x \cdot n_y \cdot n_z$ due to the implicit topology of the simple Cartesian grid (Figures 5.4(c) and 5.3).

The essential part of the three iterative solvers for large sparse matrices, Jacobi, Gauss-Seidel, and Conjugate Gradient, is algorithmically equivalent to a matrix-vector product. All of the algorithms perform an iteration on the non-zero entries of the sparse matrix and combine them with some data. Therefore, a focus on the explanation of computing the stencil on the GPU is given.

For computing one component y_{ijk}, represented by b_{ijk} on the Equation (5.12), the following data need to be accessed: the memory of y_{ijk} for writing the result, the matrix entries A_{ijk}, $A_{(i\pm1)jk}$, $A_{i(j\pm1)k}$, $A_{ij(k\pm1)}$, and the corresponding entry on the right hand

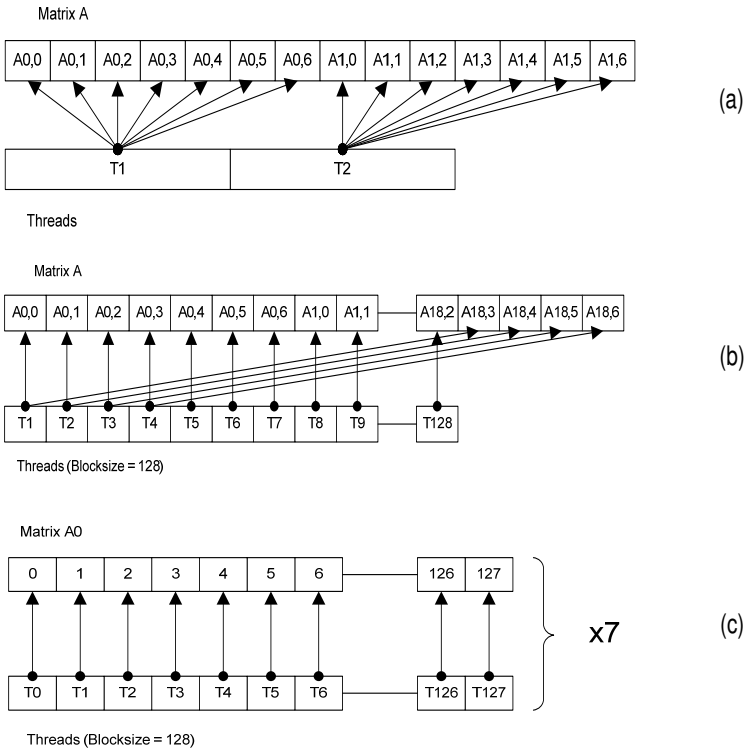


Figure 5.4: Enabling memory coalescing access: (a) simple loading, where different threads access different addresses; (b)improved loading, where coalesced access is partially achieved; (c) final loading strategy, where the starting address for each block will be aligned to a multiple of 128

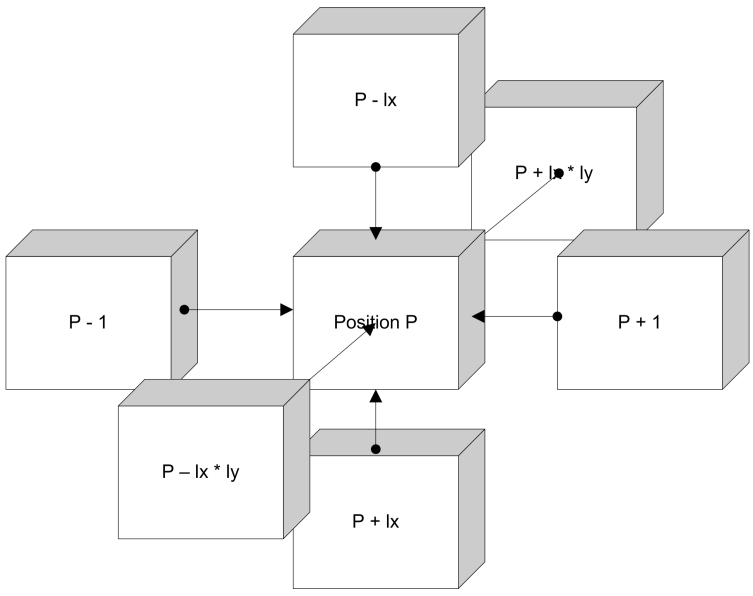


Figure 5.5: Stencil representation: relative positions of the Control Volumes

side x_{ijk} and its neighbors $x_{(i\pm1)jk}$, $x_{i(j\pm1)k}$, $x_{ij(k\pm1)}$. Those access patterns can be interpreted such that only data from all adjacent cells is needed. In that way, an iteration or a matrix multiplication can be executed for one equation with a coalesced memory access pattern, except for the values $x_{(i\pm1)jk}$, $x_{i(j\pm1)k}$, $x_{ij(k\pm1)}$, located at the adjacent cells.

Then, the shared memory to buffer the access for $x_{(i\pm1)jk}$ is used. For the remaining values $x_{i(j\pm1)k}$, $x_{ij(k\pm1)}$, it is noticed that the access is not coalesced in that pattern.

5.4 Chapter Remarks: gaining performance with a GPU approach

Specifically for exploring the best performance of GPUs, a novel method to improve locality of data on GPU memory accesses focused on the computations of the SLEs' solvers was developed. For that purpose, the GPU shared memory, which is a small cache-like memory with high access bandwidth, was explored together with the coalesced loading of data from the global memory.

Such kind of strategies are mandatory to highly benefit from the GPU computational power. For the specific solvers, they are crucial when dealing with the iterative stencil computations. Similar works have also developed specialized strategies for other specific solvers, like the ones based on factorizations for dense matrices (TOMOV et al., 2010; WHITE; DONGARRA, 2011). For comparison purposes, the next chapter shows that the presented methods applied for iterative solvers achieve a better performance in comparison to the related work of Thibault and Senocak (THIBAULT; SENOCAK, 2009).

6 EXPERIMENTAL VALIDATION & PERFORMANCE ANALYSIS

Although the GPU can be more powerful to deal with those kind of data-intensive tasks, there are many scenarios where the CPU provides better performance when working with tasks with different problem size domains (BINOTTO et al., 2010). There are cases where 1 GPU performs better than 2 GPUs. This chapter exposes the performance comparison of the three iterative solvers for SLEs tasks implemented on the CPU and implemented on the GPU using the methods described in the previous chapter. The limits where the tasks have better performance on the CPU and on the GPU are characterized and even more important the benefits of the load-balancing approach in an execution scenario of concurrent tasks.

Firstly, a performance analysis of the solvers is performed with the goal to show a clear need for scheduling methods. Then, the proposed scheduling strategy is evaluated, showing that a significant improvement can be achieved by dynamic load-balancing. The experiments of the implementations were made with respect to the performance between the CPU and GPUs. Of special interest are the following two issues:

1. The conditions where the solvers, individually, obtain better execution performance, i.e., the *break-even points* that can be considered as the decision point to schedule a solver for a PU;

2. The conditions where the solvers are seen as several high-level tasks, produced by the execution of multiple applications, to be assigned over the PUs of an execution platform.

Three heterogeneous PUs were used in the experiments:

- CPU 4-core (Intel Q6600) of 2.4GHz, 8MB of L2 cache, and 4GB of main memory with 6.4GB/s of bandwidth;

- GPU Geforce 8800GT (14 streaming multiprocessors - 112 cores - with a core clock frequency of 600MHz and 512MB of memory with bandwidth of 57.6GB/s);

- GPU Geforce GTX285 (30 streaming multiprocessors - 240 cores - with a core clock frequency of 1476MHz and 1000MB of memory with bandwidth of 159.6GB/s).

The processing units' communication was made via PCIe x16 v.1, which bounds the bandwidth of the CPU-GPU link by 4GB/s. The next sections describe the achieved results on the two aforementioned aspects.

6.1 Performance of the Iterative Solvers over the CPU-GPU Platform

The implementation of the solvers for the GPU followed the technique described on the last chapter, focusing on the stencil computation of the matrix-vector computation. The CPU version, on the other hand, was implemented using the OpenMP API for exploring the multiple cores of the CPU. It was also focused on the *loop* for computing the stencil, making both implementations more appropriately correlated as a direct performance comparison between different implementations targeting the CPU and the GPU is of difficult comparison.

The used measure of convergence for the solvers, where the iteration process stops, is the residual. The residual indicates how far the solution is from the correct value of *b*. For its calculation on every iteration, it is used the root mean square of the approximation vector ($x^{(m)} \rightarrow x^{(m+1)}$) as accuracy:

$$R = \sqrt{(x^{(m+1)} - x^{(m)})^2}. \tag{6.1}$$

The used convergence for the solvers is the residual smaller than 1e-4, i.e., when the residual is minimal the solution is converging and cannot be considerably improved on next iterations.

Figure 6.1(a) shows the performance of the solvers without exploring memory coalescing and Figure 6.1(b) exposes the results with the proposed strategy for data locality on the GPU. Particularly, for 8M of unknowns, the Jacobi solver reached 406 milliseconds (ms) on the GTX285 with the new strategy and 609ms using the common approach. It also achieved 2637ms with the new strategy and 4370ms without it on the 8800GT. The Red-Black Gauss-Seidel implementation executed using the coalescing strategy at 5139ms on the 8800GT and at 586ms on the GTX285.

The experiments show that the Conjugate Gradient solver obtained the best performances over the GPUs. For the same 8M of unknowns, the CG took 2042ms with its default implementation and 1198ms with the presented memory strategy on the 8800GT. This represents a speedup factor of 1.7. The same experiment executed at 369ms without the strategy and 313ms with the strategy on the GTX285 (Figure 6.1(c)), with a speedup factor of 1.2. For comparison (not illustrated on the pictures due to proportionality scale), a Jacobi-preconditioned Conjugate Gradient solver obtained a performance of 39219ms on the CPU using OpenMP, representing a magnitude of hundred times slower that the GPU.

It is interesting to note the *bumps* on the graphs. For the mentioned CPU implementation, the graph is represented by more stable and smoother curves. For the GPU version without the shared memory technique, the bumps are considerably bigger than the ones with the presented strategy. This effect is related to memory accesses. Without using the GPU shared memory, which is a GPU cache-like memory, the bumps are higher since there is a considerable latency to gather the data at the global memory for processing. Transporting batches of data from the global memory to the shared memory is a way of diminishing this effect as data are accessed using a higher memory bandwidth speed. By introducing several layers of cache at the GPU, like the L1 and L2 caches from the CPU, it is believed that this effect can be minimized.

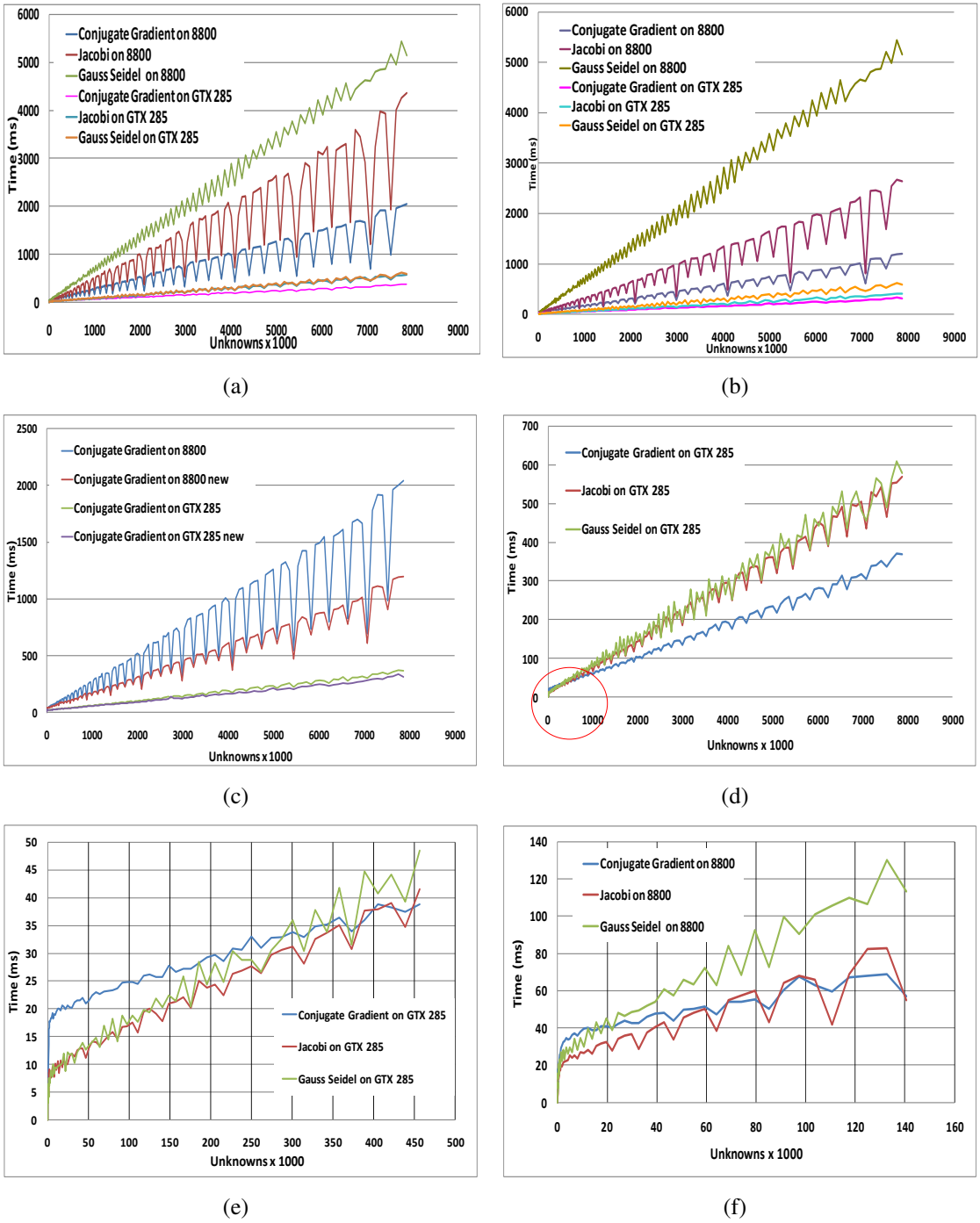


Figure 6.1: Performance of the solvers exclusively on the GPUs: (a) without a coalesced access strategy; (b) with the *new approach*; (c) the Conjugate Gradient solver with and without the proposed approach; (d) on the GTX285 with the proposed approach; (e) break-even point on the GTX285 with the proposed approach (zoom of the red circle area of (d)); (f) break-even point on the 8800GT with the proposed approach.

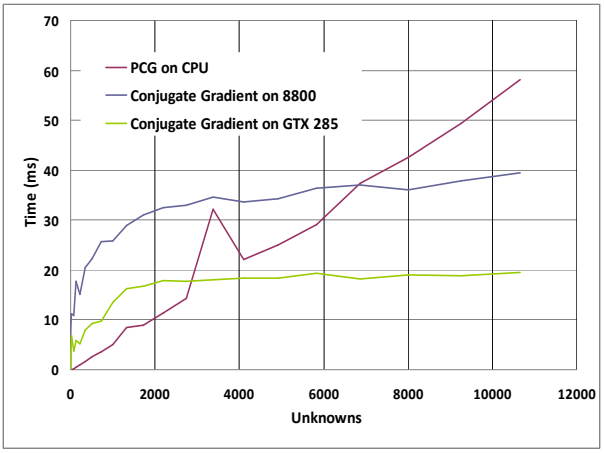


Figure 6.2: Performance break-even point on CPU and GPU.

In all cases, the CG will naturally converge faster than the other solvers with a given "big enough" problem at the GPU. However, Figure 6.2 shows its performance over the PUs for small problems. The CPU obtained better performance until 3K unknowns compared to the GTX285, and until 7K unknowns compared to the 8800GT. In the cases that the CPU obtained better performance, few threads were launched to enable latency hiding on the GPU. After these break-even points, the GPUs processing power was fully utilized.

Moreover, taking the PU that obtained a global better performance, Figure 6.1(d) depicts the performance of the solvers on the GTX285. The area of the picture with a red circle indicates that the performance behavior with few number of unknowns differ from the overall tendency. The Conjugate Gradient solver will become faster than the Jacobi and the Red-Black Gauss-Seidel after reaching the border of approximately 500K unknowns as shown in Figure 6.1(e). This gain indicates that in the CG algorithm, many operations will always be "naturally" coalesced, since a sequential strategy is used, i.e., vector-vector operations in which one block of threads can always load a sequential segment of data to be computed. The same is valid for the reduction kernel used to sum up values of a vector. The Jacobi algorithm (and, thus, the matrix multiply kernel in the CG) will also profit from such loading strategy. However, the Conjugate Gradient just needs an accelerated (improvement) strategy for the matrix-multiply operation, being all other computations coalesced. Applied to the Geforce 8800GT, Figure 6.1(f) points out the break-even point of 140K unknowns for the CG being faster than other solvers.

In addition, the implementation was extended to a multiple GPU approach. In general, a SLE cannot be simply divided to be computed in parts by different processing units, since each element depends on other neighbor elements. Nevertheless, in the case of structured grids, the elements needed to compute one iteration on one part of the SLE are known. The earliest element needed is one layer of elements *ahead* of the starting element of the partial x vector. The last element needed is one layer of elements *behind* the last element of the partial x vector. Those are the elements that have to be additionally loaded to the elements that will be computed. These elements have the same size of the padding used to avoid illegal memory accesses. This way, instead of filling the padding with zero entries, it will be filled with the current values of x. Moreover, because of the partial x solution depends on the elements in the padding zone, it would not converge towards the real solution if these interfaces are not updated. Then, those elements will have to be

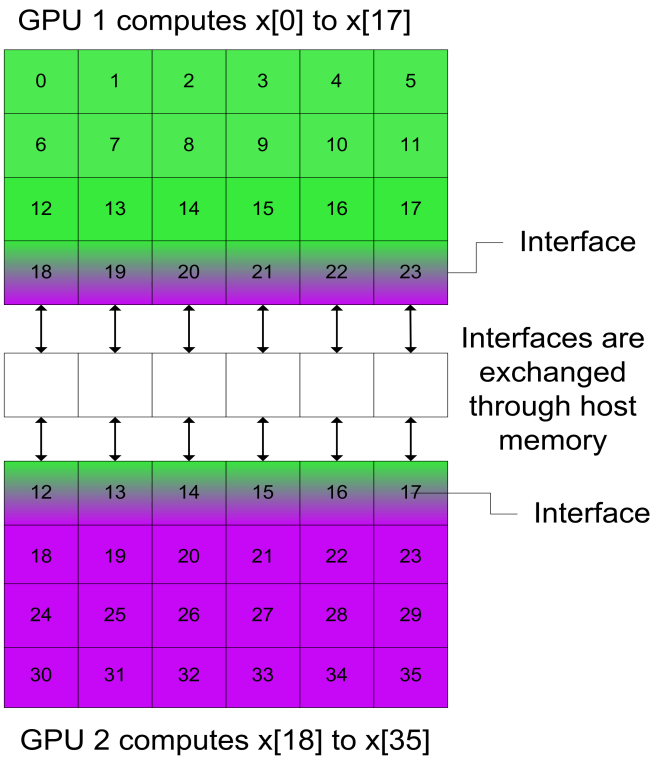


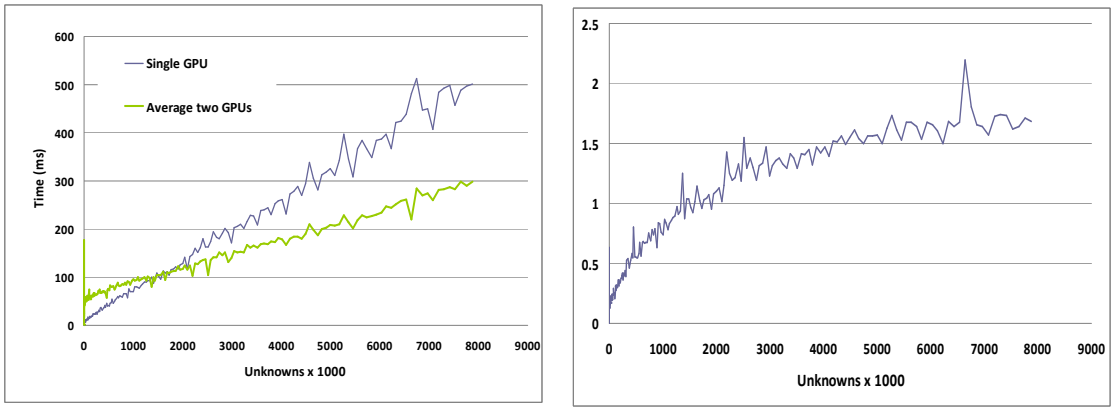
Figure 6.3: Using 2 GPUs for computing the stencil: data is divided in two sets with a redundant interface.

updated after each iteration.

When using more than one GPU, this communication is expected to diminish performance, resulting in a speedup smaller than the number of available GPUs. However, the GPUs have high computing power and even with the negative effects introduced by the communications effort, the implementation of this approach can prove efficiency. Figure 6.3 shows an example of a fixed decomposition of the problem domain to be computed by different devices, depicting the need of the redundant interface raw. This can be generally extrapolated for two other types of processing units.

The communication needed for a system with two processing units is exactly one layer of control volumes. This layer has to be "downloaded" from each device and, then, "uploaded" to each device after each iteration. Figure 6.4(a) illustrates that a 2-GPU implementation will need about 2M unknowns to be faster than the execution on one GPU, considering two GTX285 GPUs and using the Conjugate Gradient solver as benchmark. The use of two devices with less than 2M elements results in an increased communication effort that cannot be balanced by the combined higher processing power.

The multiple GPU approach demonstrates that the speedup depends strictly on the problem size as illustrated on the Figure 6.4(b). In the case study used in this research, each GPU computed half of the elements in the domain plus the borders' elements. The maximum speedup achieved was approximately 1.7 for 8M unknowns. This result is comparable to the work of Thibault and Senocak (THIBAULT; SENOCAK, 2009), which reached a speedup using 2 GPUs of approximately 1.5 for the same domain size. Although a better speedup is obtained, a direct comparison is difficult due to system differences.



(a) (b)

Figure 6.4: Performance of the solvers using 2 GTX285 GPUs: (a) comparison with one GPU; (b) speedup using two GPUs.

They used a TeslaC870 GPU (128 cores and 1.5GB of memory) over a Nvidia TeslaS870 server.

For a system with more than two GPUs, twice of such communication bandwidth is necessary, since the devices need information from the *neighbors*. For example: using three GPUS, GPU 1 can compute the beginning of the solution, GPU 2 the second part of the solution, and GPU 3 the last part of the solution. GPU 1 and GPU 3 will need information from GPU 2, since GPU 2 is adjacent to both of them. Because of that, GPU 2 will need information from both GPU 1 and GPU 3.

Nevertheless, communication will be performed in parallel, since each processing unit is controlled by a single thread. Thus, the addition of more than three devices will increase the speed of the computation, but will not have effect on the communication time. The initial time needed to upload the data to the device will decrease, since each device will only need its respective part of the right hand side and the matrix, as opposed to a single GPU implementation, where all data has to be transferred to one GPU. The data transfer to each co-processor will be performed in parallel and the required processing time is divided by the number of used devices.

Based on that, the bandwidth through which data is transferred to the GPU directly depends on the size of data to be transferred. A small problem can be processed just by a limited number of GPU threads, under-utilizing the GPU. In these cases, the CPU will achieve better performances. In detail, this is mainly due to memory bandwidths in the different stages of communication. First, the data has to be sent from CPU memory to the GPU and, then, loaded from the GPU main memory into the shared memory or into the GPUs registers/textures. For all of these stages, the real bandwidth depends on the size of the data chunk that is processed. Secondly, CUDA and OpenCL enable the GPU to switch between blocks while some of them are waiting for data. An idle block will be set to an inactive state until its data has arrived and, in the meantime, another block may be processed. This will almost eliminate idle time on the PU. If the problem size is small in a way that there are not enough blocks to fulfill the shared memory, the latency hiding strategy cannot be employed efficiently.

Thereby, larger data arrays will be transmitted more effectively to the GPU, improv-

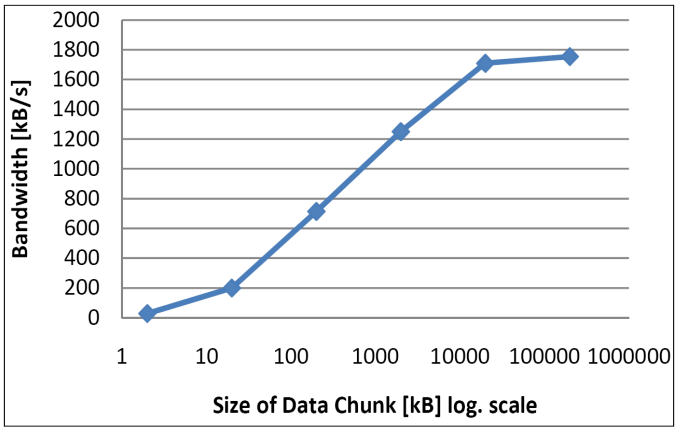


Figure 6.5: Real consumed bandwidth for the solvers.

ing bandwidth. This implies that data should always be grouped into larger chunks to exploit the best possible bandwidth flow and also be processed more efficiently by the GPU memory controller. Figure 6.5 shows how the size of data chunk is important to reach the optimal bandwidth, which differs from the theoretical one provided by the bus/memory specifications (4GB/s in the used PCIe x16 v.1 bus). The real achieved bandwidth was the half of the theoretical one.

6.2 Performance Analysis of the Sm@rtConfig Runtime System

Although data decomposition and assignment seems an interesting approach for gaining performance towards the exploration of hybrid implementations by correlating the subdivision of the problem size domain with the computational power of the processing units, the core of this research attacks a more complex problem. It is based on *dynamic* task assignment.

The results presented in this section are focused on an evaluation analysis of the scheduling strategies presented in Chapter 4. Emphasized is the gain obtained using a CPU-GPU heterogeneous platform to compute a set of tasks, characterized in Chapter 5 and evaluated in the last section of this chapter, and the overhead of the proposed methods.

Just when the application starts, the first assignment phase allocates a set of tasks onto the aforementioned processing units. In the experiments, initially 12 tasks were used that represented the three different solvers applied to different sizes of work domain. Table 6.1 shows the set of tasks with their respective number of unknowns and the real costs for the CPU and the GPU GTX285. In the experiment, it was used two of each task to compose the set of 12. These costs were obtained on the profiling benchmark using an average of 5 single executions for each task and represent tasks where i) GPU is faster, ii) GPU and CPU have almost the same performance, and iii) CPU is faster.

Based on these execution costs, the proposed heuristics were evaluated. Firstly, for a simple demonstration, Table 6.2 presents a comparison with the optimal assignment using exhaustive search. The heuristic using just Algorithm 1 produced a poor allocation, having all tasks executed at 154110ms. On the other hand, its swap improvement in Algorithm 2, produced better allocations. The execution of all tasks using that improvement

Unknowns	CPU (ms)	GPU (ms)
614125	149560	1800
592704	151380	1600
12167	2200	640
9261	1700	630
4096	670	630
2048	330	630

Table 6.1: Domain sizes and execution costs of the tasks on the CPU and GPU.

Optimal	Algorithm 1	Algorithm 2
0	0	0
0	1	0
0	0	0
1	0	1
1	1	0
1	0	0
0	0	0
1	0	1
0	0	0
0	1	0
1	1	1
0	0	1
cost=8500ms	cost=154110ms	cost=9120ms

Table 6.2: Comparison of the FAP allocation heuristics: 0- assigned to the GPU, 1- assigned to the CPU.

achieved 9120ms, which is not far from the total execution cost of the tasks using the optimal assignment (8500ms). It represents an error of 7.29% and a gain of 14.27% in comparison to statically execute all tasks on the GPU.

Performing several executions of Algorithm 2, in a total of a hundred executions, the error reached an average of 5.57%. Figure 6.6 shows the average error of this first method for an increasing number of tasks to be allocated.

Nevertheless, due to some special characteristics of the GPU, the results discussed above where acquired instantiating a new *context* for each task. In the GPU, using OpenCL or CUDA, a context is analogous to a CPU *process*. All resources and actions performed within the driver API are encapsulated inside a context and the system automatically cleans up these resources when the context is destroyed (STONE; GOHARA; SHI, 2010; NVIDIA, 2010b). By creating just one "general" context and, afterwards, by instantiating the multiple tasks on that single general context, more than 38% of gain was achieved for an experiment composed of 24 tasks in comparison to the static assignment of all tasks to the GPU. These results include the costs for data transfers, initializations, and possible latency. Table 6.3 shows a comparison of the scheduling gain and the produced overhead in milliseconds for 3 sets of tasks composed of 12, 24, and 36 tasks. For the case studies of our experiments, such overheads are considered low or even negligible.

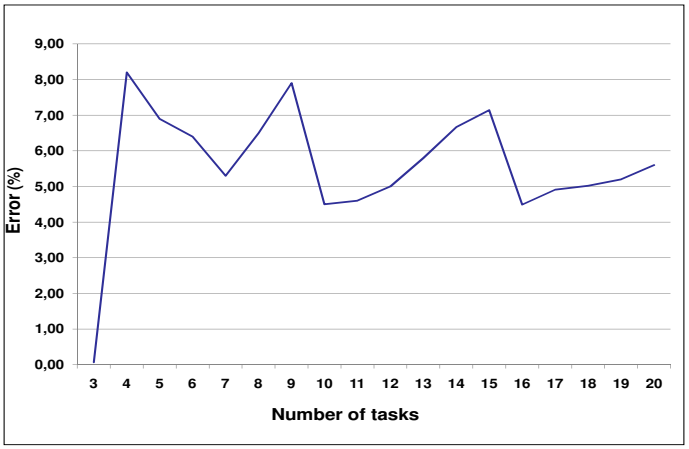


Figure 6.6: FAP heuristic accuracy

Tasks	Overhead (ms)	Gain (%)
12	~0	35.58
24	~0	38.29
36	15	29.63

Table 6.3: Overhead of the dynamic scheduling using Algorithm 2 and its gain in comparison to scheduling all tasks to the GPU

To complement the evaluation of all heuristics proposed on the FAP phase, Table 6.4 shows an effectiveness comparison of the scheduling techniques:

- Optimal assignment (exhaustive search).
- All tasks assigned to the GPU.
- The assignment executing the FAP using Algorithms 1 and 2.
- The assignment executing the FAP using Algorithms 3 and 4.

The *overhead* represents basically the cost of the heuristics themselves, the *solve time* is the execution time of the task on the PU (including data transfers to/from the PU), and the *total time* is the solve time plus the overhead. In this case, the table shows that Algorithms 3 and 4 are the most efficient heuristics to be used in the FAP.

It is important to mention that the used performance measurement precision was milliseconds (ms). This explain the apparently null overhead of Algorithms 2 and 4. The unique absolute zero overhead is given when all tasks are statically assigned to the GPU. When such precision is set to microseconds an overhead different from zero may be measured. However, as this thesis focuses on high-level tasks and the microseconds precision can be negligible in terms of CPU consumption, milliseconds precision suffice for this case study.

Moreover, a comparison was performed of all heuristics and the two assignment phases, with an experiment composed of 24 tasks arriving at the FAP and 42 at the RAP.

	Optimal	All on GPU	FAP-Alg.2	FAP-Alg.4
Overhead (ms)	5012	0	~0	~0
Solve time (ms)	6130	7660	6833	6145
Total time (ms)	11142	7660	6833	6145
Error (%)	0	24.96	11.47	0.25

Table 6.4: Comparison of the scheduling techniques for 24 tasks: *overhead* is the time to perform the scheduling; *solve time* is the execution time to compute the tasks; *total time* is the overhead plus solve time; and the *error* represents how worst is the total time of the techniques in comparison to the fastest solve time, which is the *optimal* solution without its overhead.

After the first scheduling, the dynamic allocation is performed as a FIFO design. The timing database is used as support to perform the dynamic allocation, since it is initialized with samples of costs based on the benchmark analysis. This strategy is simple and has the goal to reduce the overhead of the scheduling approach, which can be increased for a larger number of tasks and processing units.

Using the timing database and assuming that the tasks will execute in a certain periodicity and will deal with similar problem domain sizes, an allocation for a new task is performed consulting its recent execution. However, if there is no entry for a task using the specific current domain size, the method performs a database search with the goal to identify a *similar* domain size and assumes its cost as the base for the allocation. After the task execution, a new entry is then inserted with the real cost gathered using the assigned PU. The estimated costs for the other PUs are the same of the *similar* task used as a base for the scheduling.

Thus, introducing the RAP, Table 6.5 presents the performance of all described heuristics for scheduling tasks over one CPU and one GPU. For direct comparing purposes of the heuristics' efficiency, the experiment used the same algorithm at the FAP and at the RAP, i.e., the table presents the results of using each heuristic on both phases. The experiment was performed using two variations: fixed tasks, where all tasks already assigned but not executed are not resubmitted for a new scheduling; and with reconfiguration, as depicted by the RAP, where tasks can change their assignment to promote a scheduling gain.

It is important to note that using the heuristic of Algorithm 2, a worst result was achieved using the reconfiguration feature at the RAP. The reason for this issue is due to the algorithm's complexity. The approach took more time to evaluate the RAP assignment with a larger number of tasks. And due to this overhead, it was identified that the PUs became idle and ready to process a new task before the assignment decision was finished.

On the other hand, this problem did not occur with Algorithm 4, which delivered the fastest assignment and achieved the best total execution time for the tasks. An interesting point is, in this case, that the reconfiguration technique did not promoted an expressive final gain (16037ms against 16084ms without reconfiguration of tasks).

Finally, Algorithm 5 resulted in a slightly worse scheduling than Algorithm 4, but still considerably better if compared to all tasks being statically assigned to the GPU. This dynamic load-balancing heuristic takes advantages from the database infrastructure that

Scheduling	Performance	With Fixed tasks (ms)	With Reconfiguration (ms)
Algorithm 2	overhead	15	1341
	idle time GPU	~0	967
	idle time CPU	~0	47
	total time	16693	17425
Algorithm 4	overhead	~0	~0
	idle time GPU	~0	~0
	idle time CPU	~0	~0
	total time	16084	16037
Algorithm 5	overhead	~0	~0
	idle time GPU	~0	~0
	idle time CPU	~0	~0
	total time	16927	16988
All on GPU	total time	20499	20499

Table 6.5: Comparison of all techniques for 24 tasks in the FAP plus 42 tasks arriving in the RAP: Algorithm 4 produced the best execution times for one CPU and one GPU, while Algorithm 2 did not perform well with reconfiguration. Algorithm 5 represents the generalization for several PUs and achieved a better performance in comparison to scheduling all tasks to the GPU.

contains recent execution times of the tasks. This way, the dynamic scheduling is context-aware performed. The main advantage of this heuristic is its flexibility to consider more than two PUs, point that is a severe drawback presented by the other techniques.

Based on these findings, the proposed system uses Algorithm 4 at the FAP and Algorithm 5 at the RAP focusing on an execution platform of one CPU and one GPU. For an execution platform composed of more that two processing units, Algorithm 5 should still present superior behavior to deal with the dynamic load-balancing scenario and to be the core of the Sm@rtConfig scheduling module even if it promotes a slightly worse scheduling than Algorithm 4.

6.3 Chapter Remarks: gaining performance with the proposed system

Based on the *solvers' performance analysis* with and without the proposed method for data access on the GPU, it is clear that there are some scenarios where the CPU provides better performance, partially based on the amount of data to be processed. In some cases, the GPU has better computational times. This is not only valid for the three iterative solvers presented as case study, but also for several other tasks that work with dynamic data.

As an example that this characteristic of having better performances on a specific processing unit depending on data not known a priori also persists for other tasks, consider a so-called Multigrid solver (KELLER, 2009), that can be programmed effectively on GPUs. Just for illustration, Figure 6.7 shows the presence of a frontier, in terms of control volumes (CVs), that delimits the processing unit where the solver achieves better

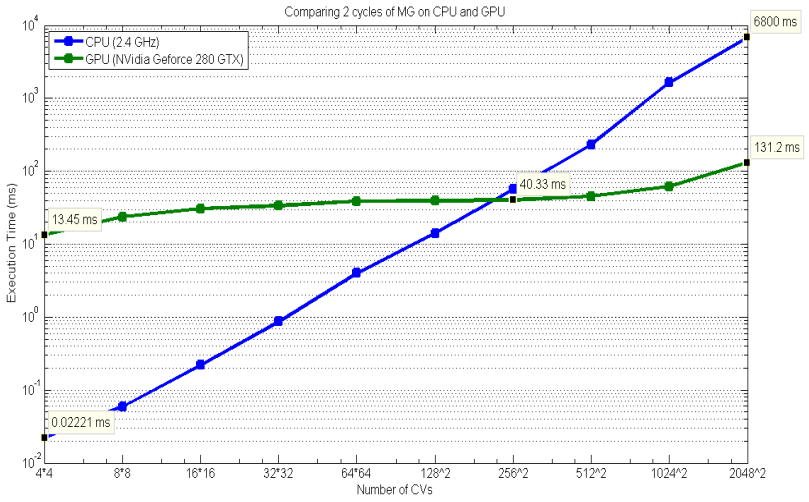


Figure 6.7: Performance comparison of the Multigrid (MG) solver over the CPU and the GPU (KELLER, 2009)

performance.

Additionally, this chapter has shown that one GPU achieves better performance than two GPUs in the cases that the CFD simulation works with small problem size domains.

Based on the performance results of the solvers, a simple scheduling over the CPU and the GPU for the tuned CFD applications could be performed using the *break-even points* as the main decision parameter. However, more elaborated dynamic scheduling techniques for desktop platforms composed by CPU and co-processors could considerably improve the current static and programming time scheduling of high-level tasks used by OpenCL or CUDA.

The set of five context-aware heuristics was evaluated for the scheduling of the high-level solvers. The experiments explored the compromise between reducing the execution time of the multiple tasks, due to appropriate dynamic scheduling in two phases (FAP and RAP), and the cost of computing such scheduling applied on the platform composed of CPU and GPU. The novel heuristic of Algorithm 4, which takes into account the performance differences of the tasks over the CPU and the GPU, achieved the best compromise. However, its design is focused to work with two processing units.

To overcome this drawback, the proposed Algorithm 5 was structured to be sufficiently flexible to work with several processing units. In the experiments, the heuristic achieved a similar compromise as Algorithm 4, being suitable for a generic approach of the Sm@rtConfig runtime system.

Finally, the scheduling system experiments were carried out using the solvers as high-level tasks, but it can naturally be generalized for any type of high-level task or algorithm.

7 CONCLUSION

This thesis presents a set of strategies and mechanisms that, when fully incorporated into a standard programming API oriented to a heterogeneous desktop execution platform, the OpenCL, promotes an efficient, platform independent, and context-aware assignment of high-level tasks and algorithms. The OpenCL can incorporate the strategies by offering a dynamic scheduling feature for high-level tasks using the developed classes at Sm@rtConfig.

The main goal, thereby, is to enable an efficient utilization of the heterogeneous multi and many-core desktop co-processors, like the CPU and the GPU, shared among a multitude of applications' tasks. By allowing a context-aware management of the high-level algorithms, the proposed system is capable to assign those tasks to the computational resources based on their recent execution history, verifying the tasks' performance not only in an idle scenario, but taking into account real and current platform conditions.

As an overview, the contributions reached in this thesis cover the following goals:

- to provide efficient load-balancing over the processing units of a desktop, considering real scenarios of execution;
- to provide a strategy that aggregates allocation and scheduling capabilities to tasks that must be executed by heterogeneous desktop systems;
- and to provide acceleration methods using the CPU and the GPU for a real-time CFD simulation research project.

In order to achieve the proposed goals, three research fields were combined: high performance computing, distributed systems, and computational engineering. Focusing on the performance improvement of virtual engineering applications, more specifically on iterative solvers for systems of linear equations used by real-time CFD applications, a study of the performance evaluation of the solvers over an asymmetric desktop platform composed of CPU and GPUs was performed. Based on experimental results, it was observed that the performance is directly influenced by the domain size (number of unknowns). For a GPU approach, the characteristics of the solvers are important to be analyzed. Even more important is the management of memory accesses in order to obtain an optimal gain on the calculations and the size of data chunk transferred to the GPU. Break-even points, where a different PU obtains a better time performance, were discussed as well as the speedup using multiple GPUs.

Besides, a GPU tuned library of iterative solvers was also developed to improve the real-time CFD case study performance. Focusing that goal, a performance comparison was done for different solvers in a heterogeneous platform, a GPU data access strategy description for iterative solvers that enabled memory coalescing was developed, and different scenarios were analyzed where each solver obtained better execution times.

To accomplish the other proposed goals for load-balancing and dynamic scheduling, core of this work, a runtime system was developed to allow an optimal use of resources offered by asymmetric processing units. The system provides compliance with multiple concurrent CFD applications to be executed on a desktop platform and with dynamic changes in the problem domain size (unknowns).

The presented *context-aware runtime and performance tuning system* is based on a compromise between reducing the execution time of the applications' tasks - due to appropriate dynamic scheduling - and the cost of computing such scheduling applied on a platform composed of CPU and GPUs. A model for a *first scheduling* based on an offline performance benchmark is combined with a *runtime model* that keeps track of real execution times of the tasks with the goal to extend the scheduling process of the OpenCL API.

The system was validated using a CPU-GPU platform for computing iterative SLEs' solvers focusing on the number of unknowns as the main parameter for assignment decision. Based on the performance evaluation of the solvers used by real-time CFD applications, the need for dynamic scheduling strategies in a heterogeneous desktop execution platform was verified. By scheduling tasks to the CPU and to the GPU, it was achieved an execution time gain of 21.77% in comparison to the static assignment of all tasks to the GPU with a scheduling error of only 0.25% compared to exhaustive search.

The core module of the system works with scheduling heuristics oriented to one CPU and one GPU. The generalization for multiple PUs was also described, making use of a greedy strategy for more than two processing units. The tasks are to be sorted based on their performance and assigned trying to minimize the total execution cost. The termination of currently assigned tasks on the PUs can be predicted and, based on that, the runtime phase computes the time that the PUs will become idle. This way, it can assign a new task to the PU that minimizes the time for becoming idle plus the estimated execution time of this new task.

Using the proposed system, the programmer do not have to be concerned about assigning an algorithm to the best processing unit, specially GPU or CPU, or deal with execution conditions not known a priori. The proposed strategies can now be integrated to the OpenCL in order to provide an automatic dynamic kernel assignment feature. Additionally, the proposed strategies can be applied as a support to provide real-time geometry-modifying of models on concurrent CFD simulations.

In summary, exploring efficiently the utilization of asymmetric devices on the desktop is an actual topic being investigated by the scientific community. The proposed system showed one possible way to tackle the important opened problems introduced on the beginning of this dissertation. Besides, more appropriate solutions can improve the techniques developed in this thesis. The performed tests have enabled the identification of some points that can be further explored. Some features are still being improved as part of ongoing work and others may serve as motivation for scientific investigations in a future research.

7.1 Future Research

Part of ongoing work is a study of regression methods that are to be used when a task with a specific domain size is not presented on the database. Additionally, the developed strategies work with high-level algorithms, assuming they are independent. Thus, tasks from different applications arrive solely or in batches in a certain order,representing the application workflow. The locality of tasks is considered, but it is not the goal to keep a locality of data over the processing units for dependent tasks. However, the framework is prepared to deal with dependencies with minimal improvement because the performances of the tasks are stored for every PU as well as the costs for transferring the data from/to the CPU to/from the GPU. In a batch of dependent tasks, for example, the framework can calculate if it is better to place a task in the same PU from its predecessor taking into account data transfer timing.

An aspect orientation approach is also part of ongoing work as a way of implementing the developed scheduling methods and strategies that deal with Non-Functional Requirements (NFR), like timing measurements. The objective is the integration with a framework for timing aspects, named Distributed Embedded Real-time Aspects Framework (DERAF), which provides aspects with high-level semantics to specify the handling of crosscutting NFRs within Unified Modeling Language (UML) models (WEHRMEISTER, 2009). Partial results regarding this exploration can be found on the Appendix B.

Future research directions lead to a further analysis of how the CFD application is benefited from such tuning of solvers and dynamic scheduling using a hybrid approach. Dynamically adapting the solvers' convergence in order to continuously assure an acceptable relationship between the real-time requirement and the precision of the solution is an important issue to attack on future research.

Regarding the platform, an increased number of GPUs can be used as well as, more challenging, other types of PUs (given specific tasks' implementation or drivers). An important improvement is to extend the solvers to explore the modern CPUs using a hardware specific library, like the Linear Algebra Package - LAPACK [1]. This way, the execution times of the solvers over the CPU and the GPU could be more fairly made.

A work towards a general framework for creating common auto-tuned and data-parallel algorithms is highly significant. The identification of tasks' common characteristics, like a classification of matrices formats and their numerical methods to solve SLEs, are crucial to develop a set of numerical algorithms with a variety of computational patterns for the wide area of virtual engineering. As an example, results indicated that the introduced core functions presented in this thesis are easily modified to produce a highly efficient Multigrid algorithm that uses the GPU for the computationally intensive parts of the algorithm.

The full integration with the OpenCL API is also part of future research. Additionally, appropriate statistic and probability models, including multivariate regression methods and prediction of new observations may improve to concept to predict the future allocation of tasks based on their recent past (MONTGOMERY; RUNGER, 2011).

Another important possibility to be investigated is the use of this thesis in order to meet possible requirements of energy consumption. As an example, the cost of the tasks could be represented as energy consumption and not execution times. An hybrid model, based

[1] http://www.netlib.org/lapack/

on OpenCL, that can offer better performances and better energy consumption seems to be an important contribution towards future smart systems. This issue also leads to an exploration of tasks' migration, i.e., when a task migrate its execution to other PU during execution, using the OpenCL (TAKIZAWA et al., 2011).

7.2 Closing Remarks

Finally, the exposed potential for the generalization of the developed Sm@rtConfig System – aiming the dynamic workload balancing over the processors – indicates the assumption that the findings of this research could be generalized and may probably apply to a variety of other tasks, processors, and problems, like, for example: computational biology (CHEN; SCHMIDT, 2005; SOUSA; MELO; BOUKERCHE, 2010), rendering (HUI; XIAOYONG; SHULING, 2009), the Cell BE processor (BELLENS et al., 2006), the FPGA (TSOI; LUK, 2010), embedded systems (DE BOSSCHERE et al., 2007), among others.

REFERENCES

AGULLO, E.; AUGONNET, C.; DONGARRA, J.; FAVERGE, M.; LTAIEF, H.; THIBAULT, S.; TOMOV, S. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2011., 2011. **Anais...** Los Alamitos, CA, USA: IEEE Computer Society, 2011.

AHMADINIA, A.; BOBDA, C.; KOCH, D.; MAJER, M.; TEICH, J. Task scheduling for heterogeneous reconfigurable computers. In: SBCCI '04: PROCEEDINGS OF THE 17TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN, 2004, Pernambuco, Brazil. **Anais...** [S.l.: s.n.], 2004. p.22–27.

AMD. AMD CodeAnalyst. Disponível em: <http://developer.amd.com/cpu/codeanalyst/Pages/default.aspx>. Acesso em: 01 dez. 2010.

AMD. ATI Stream SDK with OpenCL. Disponível em: <http://developer.amd.com/gpu/AMDAPPSDK>. Acesso em: 01 dez. 2010.

AMD. The AMD Fusion. Disponível em: <http://fusion.amd.com/>. Acesso em: 09 fev. 2011.

AMDAHL, G. M. Validity of the single-processor approach to achieving large scale computing capabilities. In: AFIPS CONFERENCE, 1967. **Anais...** [S.l.: s.n.], 1967. p.483–485.

AMENT, M.; KNITTEL, G.; WEISKOPF, D.; STRASSER, W. A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. **Parallel, Distributed, and Network-Based Processing, Euromicro Conference on**, [S.l.], v.0, p.583–592, 2010.

AUGONNET, C.; CLET-ORTEGA, J.; THIBAULT, S.; NAMYST, R. Data-Aware Task Scheduling on Multi-accelerator Based Platforms. **Parallel and Distributed Systems, International Conference on**, [S.l.], v.0, p.291–298, 2010.

AUGONNET, C.; THIBAULT, S.; NAMYST, R.; WACRENIER, P.-A. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. In: INTERNATIONAL EURO-PAR CONFERENCE, LECTURE NOTES IN COMPUTER SCIENCE, 15., 2009. **Proceedings...** [S.l.]: Springer, 2009. p.863–874. (Lecture Notes in Computer Science, v.5704).

AZIMI, M.; CHERUKURI, N.; JAYASIMHA, D. N.; KUMAR, A.; KUNDU, P.; PARK, S.; SCHOINAS, I.; VAIDYA, A. S. Integration Challenges and Tradeoffs for Tera-scale Architectures. **Intel Technology Journal**, [S.l.], v.11, n.3, p.173–182, 2007.

BARRETT, R.; BERRY, M.; CHAN, T. F.; DEMMEL, J.; DONATO, J.; DONGARRA, J.; EIJKHOUT, V.; POZO, R.; ROMINE, C.; VORST, H. V. der. **Templates for the Solution of Linear Systems**: building blocks for iterative methods, 2nd edition. Philadelphia, PA: SIAM, 1994.

BELL, N.; GARLAND, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: SC '09: PROCEEDINGS OF THE CONFERENCE ON HIGH PERFORMANCE COMPUTING NETWORKING, STORAGE AND ANALYSIS, 2009, Portland, Oregon. **Anais...** [S.l.: s.n.], 2009. p.1–11.

BELLENS, P.; PEREZ, J. M.; BADIA, R. M.; LABARTA, J. CellSs: a programming model for the cell be architecture. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2006., 2006, Tampa, Florida. **Proceedings...** [S.l.: s.n.], 2006. (SC '06).

BINOTTO, A.; FREITAS, E. de; PEREIRA, C.; STORK, A.; LARSSON, T. Real-time task reconfiguration support applied to an UAV-based surveillance system. In: COMPUTER SCIENCE AND INFORMATION TECHNOLOGY, INTERNATIONAL MULTICONFERENCE ON, 2008. **Anais...** Wisla, Poland: s.n.], 2008. p.581 –588.

BINOTTO, A. P. D.; BRUNETTI, G.; PEREIRA, C. E.; SANTOS, P. Research in Interactive Design: proceedings of virtual concept 2006. In: FISCHER, X.; COUTELLIER, D. (Ed.). . [S.l.]: Springer-Verlag, 2006. v.2, p.10.

BINOTTO, A. P. D.; COMBA, J. L. D.; FREITAS, C. M. D. Real-Time Volume Rendering of Time-Varying Data Using a Fragment-Shader Compression Approach. In: IEEE SYMPOSIUM ON PARALLEL AND LARGE-DATA VISUALIZATION AND GRAPHICS, 2003., 2003. **Proceedings...** [S.l.: s.n.], 2003. p.10–. (PVG '03).

BINOTTO, A. P. D.; DANIEL, C.; WEBER, D.; KUIJPER, A.; STORK, A.; PEREIRA, C.; FELLNER, D. Iterative SLE Solvers over a CPU-GPU Platform. In: HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS, 12TH IEEE INTERNATIONAL CONFERENCE ON, 2010. **Anais...** [S.l.: s.n.], 2010. p.305–313.

BINOTTO, A. P. D.; FREITAS, E. P.; GöTZ, M.; PEREIRA, C. E.; STORK, A.; LARSSON, T. Dynamic Self-Rescheduling of Tasks over a Heterogeneous Platform. In: INTERNATIONAL CONFERENCE ON RECONFIGURABLE COMPUTING AND FPGAS, 2008., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.253–258.

BINOTTO, A. P. D.; FREITAS, E. P.; PEREIRA, C. E.; LARSSON, T. Towards Dynamic Task Scheduling and Reconfiguration using an Aspect Oriented Approach applied on Real-Time concerns of Industrial Systems. In: IFAC SYMPOSIUM ON INFORMATION CONTROL PROBLEMS IN MANUFACTURING - INCON 2009, 13., 2009. **Anais...** Moscow, Russia: s.n.], 2009. p.1406 –1411.

BINOTTO, A. P. D.; FREITAS, E. P.; WEHRMEISTER, M. A.; PEREIRA, C. E.; STORK, A.; LARSSON, T. Towards Task Dynamic Reconfiguration over Asymmetric Computing Platforms for UAVs Surveillance Systems. **Scalable Computing: Practice and Experience**, [S.l.], v.10, n.3, p.277–289, 2009.

BINOTTO, A. P. D.; PEREIRA, C. E.; FELLNER, D. W. Towards dynamic reconfigurable load-balancing for hybrid desktop platforms. In: PARALLEL DISTRIBUTED PROCESSING, WORKSHOPS AND PHD FORUM (IPDPSW), 2010 IEEE INTERNATIONAL SYMPOSIUM ON, 2010. **Anais...** [S.l.: s.n.], 2010. p.1 –4.

BINOTTO, A. P. D.; PEREIRA, C. E.; GIERLINGER, T.; SANTOS, P. Research in Interactive Design: proceedings of virtual concept 2006. In: FISCHER, X.; COUTELLIER, D. (Ed.). . [S.l.]: Springer-Verlag, 2006. v.2, p.3.

BINOTTO, A. P.; PEDRAS, B. M.; GOETZ, M.; KUIJPER, A.; PEREIRA, C. E.; STORK, A.; FELLNER, D. W. Effective Dynamic Scheduling on Heterogeneous Multi-/Manycore Desktop Platforms. In: COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING WORKSHOPS, INTERNATIONAL SYMPOSIUM ON, 2010. **Anais...** [S.l.: s.n.], 2010. p.37–42.

BLUMOFE, R.; JOERG, C.; KUSZMAUL, B.; LEISERSIN, C.; RANDALL, K.; ZHOU, Y. Cilk: an efficient multithreaded runtime system. **ACM Sigplan Notices**, [S.l.], v.30, n.8, p.207–216, 1995.

BOLZ, J.; FARMER, I.; GRINSPUN, E.; SCHRöDER, P. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: SIGGRAPH '05: ACM SIGGRAPH 2005 COURSES, 2005, Los Angeles, California. **Anais...** [S.l.: s.n.], 2005. p.171.

BRANDENBURG, B. B.; CALANDRINO, J. M.; ANDERSON, J. H. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: a case study. In: REAL-TIME SYSTEMS SYMPOSIUM, 2008., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.157–169.

BRIDSON, R. **Fluid Simulation for Computer Graphics**. [S.l.]: A K Peters, 2008.

BRODTKORB, A. R. **Scientific Computing on Heterogeneous Architectures**. 2010.PhD Thesis — Faculty of Mathematics and Natural Sciences, University of Oslo, 2010.

BUATOIS, L.; CAUMON, G.; LÉVY, B. Concurrent Number Cruncher: an efficient sparse linear solver on the gpu. In: HIGH PERFORMANCE COMPUTATION CONFERENCE - HPCC'07, 2007. **Anais...** Houston, USA: s.n.], 2007. p.358–371.

BUCK, I.; FOLEY, T.; HORN, D.; SUGERMAN, J.; FATAHALIAN, K.; HOUSTON, M.; HANRAHAN, P. Brook for GPUs: stream computing on graphics hardware. **ACM Transactions on Graphics**, [S.l.], v.23, n.3, p.777–786, 2004.

BURNS, A.; WELLINGS, A. **Real-time Systems and Programming Languages**. New York, USA: Addison-Wesley, 1997.

CEDERMAN, D.; TSIGAS, P. On dynamic load balancing on graphics processors. In: GH '08: PROCEEDINGS OF THE 23RD ACM SIGGRAPH/EUROGRAPHICS SYMPOSIUM ON GRAPHICS HARDWARE, 2008, Sarajevo, Bosnia and Herzegovina. **Anais...** Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2008. p.57–64.

CEVAHIR, A.; NUKADA, A.; MATSUOKA, S. Fast Conjugate Gradients with Multiple GPUs. In: ICCS '09: PROCEEDINGS OF THE 9TH INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE, 2009, Baton Rouge, LA. **Anais...** [S.l.: s.n.], 2009. p.893–903.

CHE, S.; LI, J.; SHEAFFER, J. W.; SKADRON, K.; LACH, J. Accelerating Compute-Intensive Applications with GPUs and FPGAs. In: SYMPOSIUM ON APPLICATION SPECIFIC PROCESSORS, 2008., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.101–107.

CHEN, C.; SCHMIDT, B. An adaptive grid implementation of DNA sequence alignment. **Future Generation Computer Systems**, [S.l.], v.21, p.988–1003, July 2005.

CHORIN, A. J. A numerical method for solving incompressible viscous flow problems. **Journal of Computational Physics**, [S.l.], v.135, p.118–125, August 1997.

CHRISTEN, M.; SCHENK, O.; BURKHART, H. PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2011., 2011. **Anais...** Los Alamitos, CA, USA: IEEE Computer Society, 2011.

COHEN, J. M.; TARIQ, S.; GREEN, S. Interactive fluid-particle simulation using translating Eulerian grids. In: I3D '10: PROCEEDINGS OF THE 2010 ACM SIGGRAPH SYMPOSIUM ON INTERACTIVE 3D GRAPHICS AND GAMES, 2010, Washington, D.C.. **Anais...** [S.l.: s.n.], 2010. p.15–22.

COMMISSION, E. FP7 Information and Communication Technologies. Disponível em: <http://cordis.europa.eu/fp7/ict/>. Acesso em: 01 dez. 2010.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. Cambridge: The MIT Press, 2009.

CRANE, K.; LLAMAS, I.; TARIQ, S. In: **GPU Gems 3**. [S.l.: s.n.].

DATTA, K.; MURPHY, M.; VOLKOV, V.; WILLIAMS, S.; CARTER, J.; OLIKER, L.; PATTERSON, D.; SHALF, J.; YELICK, K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008., 2008, Austin, Texas. **Proceedings...** [S.l.: s.n.], 2008. p.4:1–4:12. (SC '08).

DATTA, K.; WILLIAMS, S.; VOLKOV, V.; CARTER, J.; OLIKER, L.; SHALF, J.; YELICK, K. Auto-tuning the 27-point stencil for multicore. In: IN PROCEEDINGS OF THE FOURTH INTERNATIONAL WORKSHOP ON AUTOMATIC PERFORMANCE TUNING, 2009. **Anais...** [S.l.: s.n.], 2009.

DE BOSSCHERE, K.; LUK, W.; MARTORELL, X.; NAVARRO, N.; O'BOYLE, M.; PNEVMATIKATOS, D.; RAMIREZ, A.; SAINRAT, P.; SEZNEC, A.; STENSTRÖM, P.; TEMAM, O. High-Performance Embedded Architecture and Compilation Roadmap. In: TRANSACTIONS ON HIPEAC I, LECTURE NOTES IN COMPUTER SCIENCE 4050, 2007. **Anais...** [S.l.: s.n.], 2007. p.5–29.

DIAMOS, G. F.; YALAMANCHILI, S. Harmony: an execution model and runtime for heterogeneous many core systems. In: HPDC '08: PROCEEDINGS OF THE 17TH INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 2008, Boston, MA, USA. **Anais...** [S.l.: s.n.], 2008. p.197–200.

FEDKIW, R.; STAM, J.; JENSEN, H. W. Visual simulation of smoke. In: SIGGRAPH '01: PROCEEDINGS OF THE 28TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 2001. **Anais...** [S.l.: s.n.], 2001. p.15–22.

FERZIGER, J. H.; PERIC, M. **Computational Methods for Fluid Dynamics**. Heidelberg, Germany: Springer-Verlag, 2002.

FREITAS, E. de; BINOTTO, A.; PEREIRA, C.; STORK, A.; LARSSON, T. Dynamic Reconfiguration of Tasks Applied to an UAV System Using Aspect Orientation. In: PARALLEL AND DISTRIBUTED PROCESSING WITH APPLICATIONS, 2008, 2008 IEEE INTERNATIONAL SYMPOSIUM ON, 2008. **Anais...** [S.l.: s.n.], 2008. p.292 –300.

FREITAS, E. P.; BINOTTO, A. P. D.; PEREIRA, C. E.; STORK, A.; LARSSON, T. Dynamic Activity and Task Allocation Supporting UAV Teams in Surveillance Systems. In: IFAC WORKSHOP ON REAL-TIME PROGRAMMING, 30., 2009. **Anais...** Mragowo, Poland: s.n.], 2009. p.51 –58.

FREITAS, E. P. d.; BINOTTO, A. P. D.; PEREIRA, C. E.; STORK, A.; LARSSON, T. Dynamic Reconfigurable Task Schedule Support towards a Reflective Middleware for Sensor Network. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WITH APPLICATIONS, 2008., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.886–891.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability**: a guide to the theory of np-completeness. New York, NY, USA: W. H. Freeman & Co., 1990.

GHARAIBEH, A.; RIPEANU, M. Size Matters: space/time tradeoffs to improve gpgpu applications performance. In: ACM/IEEE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2010., 2010. **Proceedings...** [S.l.: s.n.], 2010. p.1–12. (SC '10).

GÖDDEKE, D. **Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters**. 2010. Tese (Doutorado) — Fakultät für Mathematik, Technische Universität Dortmund, 2010.

GÖDDEKE, D.; WOBKER, H.; STRZODKA, R.; MOHD-YUSPF, J.; MCCORMICK, P.; TUREK, S. Co-Processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FeastGPU. **Journal of Computational Science and Engineering**, [S.l.], v.4, n.4, p.254–269, 2009.

GÖTZ, M. **Run-time Reconfigurable RTOS for Reconfigurable Systems-on-chip**. 2007. Tese (Doutorado) — Fakultät für Informatik, Universität Paderborn, 2007.

GÖTZ, M.; DITTMANN, F.; XIE, T. Dynamic Relocation of Hybrid Tasks: a complete design flow. In: RECONFIGURABLE COMMUNICATION-CENTRIC SOCS, 2007. **Anais...** [S.l.: s.n.], 2007. p.31 –38.

GPGPU. General Purpose using GPU. Disponível em: <http://gpgpu.org/>. Acesso em: 01 dez. 2010.

HILL, M. D.; MARTY, M. R. Amdahl's Law in the Multicore Era. **IEEE Computer**, Los Alamitos, CA, USA, v.41, p.33–38, 2008.

HONG, S.; KIM, H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. **SIGARCH Comput. Archit. News**, [S.l.], v.37, n.3, p.152–163, 2009.

HOUSTON, M. **A Portable Runtime Interface for Multi-level Memory Hierarchies**. 2008.PhD Thesis — Department of Computer Science, Stanford University, 2008.

HUI, C.; XIAOYONG, L.; SHULING, D. A dynamic load balancing algorithm for sort-first rendering clusters. In: , 2009. **Anais...** [S.l.: s.n.], 2009. p.515–519.

HWU, W.-m.; KEUTZER, K.; MATTSON, T. G. The Concurrency Challenge. **IEEE Design and Test of Computers**, [S.l.], v.25, p.312–320, 2008.

IEEE. IEEE Standard Glossary. Disponível em: <http://standards.ieee.org/findstds/standard/software_and_systems_engineering.html>. Acesso em: 11 mar. 2011.

INTEL. Intel Core2Quad Processors. Disponível em: <http://www.intel.com/products/processor/core2quad/>. Acesso em: 01 dez. 2010.

INTEL. Intel Hyper-Threading Technology. Disponível em: <http://www.intel.com/technology/platform-technology/hyper-threading/index.htm>. Acesso em: 01 dez. 2010.

INTEL. Intel Threading Building Blocks: tutorial. Disponível em: <http://threadingbuildingblocks.org/>. Acesso em: 01 dez. 2010.

INTEL. Intel Cilk Plus. Disponível em: <http://software.intel.com/en-us/articles/intel-cilk-plus/>. Acesso em: 01 dez. 2010.

INTEL. Ct: a flexible parallel programming model for tera-scale architectures. Disponível em: <http://techresearch.intel.com/articles/Tera-Scale/1514.htm>. Acesso em: 01 dez. 2010.

INTEL. Intel Array Building Blocks. Disponível em: <http://software.intel.com/en-us/articles/intel-array-building-blocks/>. Acesso em: 08 fev. 2011.

JIMéNEZ, V. J.; VILANOVA, L.; GELADO, I.; GIL, M.; FURSIN, G.; NAVARRO, N. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In: HIPEAC '09: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE EMBEDDED ARCHITECTURES AND COMPILERS, 2009, Paphos, Cyprus. **Anais...** [S.l.: s.n.], 2009. p.19–33.

JOSELLI, M.; ZAMITH, M.; CLUA, E.; MONTENEGRO, A.; CONCI, A.; LEAL-TOLEDO, R.; VALENTE, L.; FEIJó, B.; ORNELLAS, M. d.; POZZER, C. Automatic Dynamic Task Distribution between CPU and GPU for Real-Time Systems. In: IEEE INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND ENGINEERING, 2008., 2008. **Proceedings...** Washington, DC, USA: IEEE Computer Society, 2008. p.48–55.

JOST, T.; CONTASSOT-VIVIER, S.; VIALLE, S. An efficient multialgorithms sparse linear solver for GPUs. In: EUROGPU MINISYMPOSIUM OF THE INTERNATIONAL CONFERENCE ON PARALLEL COMPUTING, PARCO2009, 2009. **Anais...** [S.l.: s.n.], 2009. p.1–8.

KARIMI, K.; DICKSON, N. G.; HAMZE, F. A Performance Comparison of CUDA and OpenCL. **CoRR**, [S.l.], v.abs/1005.2581, 2010.

KELLER, K. **Efficient Multigrid Methods for Realtime Simulation**. 2009.Bachelor Thesis — Department of Computational Engineering, Technische Universität Darmstadt, 2009.

KIM, J. **Efficient rendering of large 3-D and 4-D scalar fields**. 2008. Tese (Doutorado) — Department of Electrical and Computer Engineering, University of Maryland, 2008.

KOMATSU, K.; SATO, K.; ARAI, Y.; KOYAMA, K.; TAKIZAWA, H.; KOBAYASHI, H. Evaluating Performance and Portability of OpenCL Programs. In: THE FIFTH INTERNATIONAL WORKSHOP ON AUTOMATIC PERFORMANCE TUNING, 2010. **Anais...** [S.l.: s.n.], 2010. p.15.

KRÜGER, J. **A GPU Framework for Interactive Simulation and Rendering of Fluid Effects**. 2006. Tese (Doutorado) — Fakultät für Informatik, Technische Universität München, 2006.

KRÜGER, J.; WESTERMANN, R. GPUGems 2 : programming techniques for high-performance graphics and general-purpose computation. In: PHARR, M. (Ed.). . [S.l.]: Addison-Wesley, 2005. p.703–718.

KUMAR, S.; HUGHES, C. J.; NGUYEN, A. Architectural Support for Fine-grained Parallelism on Multi-core Architectures. **Intel Technology Journal**, [S.l.], v.11, n.3, p.217–225, 2007.

LEE, S.; EIGENMANN, R. OpenMPC: extended openmp programming and tuning for gpus. In: ACM/IEEE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2010., 2010. **Proceedings...** [S.l.: s.n.], 2010. p.1–11. (SC '10).

LIN, S.; KERNIGHAN, B. W. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. **Operations Research**, [S.l.], v.21, p.498–516, 1973.

LINDERMAN, M. D.; COLLINS, J. D.; WANG, H.; MENG, T. H. Merge: a programming model for heterogeneous multi-core systems. **SIGPLAN Not.**, [S.l.], v.43, p.287–296, March 2008.

LUKSCH, P. **Increased Productivity in Computational Prototyping with the Help of Parallel and Distributed Computing**. München, Germany: Shaker Verlag GmbH, 2000.

MCCOOL, M. Scalable Programming Models for Massively Multicore Processors. **Proceedings of the IEEE**, [S.l.], v.96, n.5, p.816–831, 2008.

MCCOOL, M.; DU TOIT, S.; POPA, T.; CHAN, B.; MOULE, K. Shader algebra. **ACM Trans. Graph.**, New York, NY, USA, v.23, p.787–795, 2004.

MONTGOMERY, D. C.; RUNGER, G. C. **Applied Statistics and Probability for Engineers**. New York, USA: Wiley, 2011.

NGUYEN, A.; SATISH, N.; CHHUGANI, J.; KIM, C.; DUBEY, P. 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs. In: ACM/IEEE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2010., 2010. **Proceedings...** [S.l.: s.n.], 2010. p.1–13. (SC '10).

NVIDIA. CUDA Architecture. Disponível em: <http://www.nvidia.com/cuda>. Acesso em: 01 dez. 2010.

NVIDIA. Nvidia Geforce Series and CUDA. Disponível em: <http://www.nvidia.com>. Acesso em: 01 dez. 2010.

NVIDIA. OpenCL Programming Guide for the CUDA Architecture. Disponível em: <http://www.nvidia.com/opencl>. Acesso em: 01 dez. 2010.

OPENMP. The OpenMP API specification for parallel programming. Disponível em: <http://openmp.org/wp/>. Acesso em: 01 dez. 2010.

OWENS; JOHN, D.; LUEBKE; DAVID; GOVINDARAJU; NAGA; HARRIS; MARK; KRUGER; JENS; LEFOHN; AARON, E.; PURCELL; TIMOTHY, J. A Survey of General-Purpose Computation on Graphics Hardware. **Computer Graphics Forum**, [S.l.], v.26, n.1, p.80–113, 2007.

SCHIWIETZ, T. **Acceleration of Medical Image Algorithms Using Programmable Graphics Hardware**. 2008. Tese (Doutorado) — Fakultät für Informatik, Technishe Universität München, 2008.

SCHROUT, R. Nehalem Revolution: intel core i7 processor complete review. Disponível em: <http://www.pcper.com/article.php?aid=634>. Acesso em: 01 dez. 2010.

SCIENCE, P. C. of Advisors on; TECHNOLOGY. Designing a Digital Future: federally funded research and development in networking and information technology. Disponível em: <http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-nitrd-report-2010.pdf>. Acesso em: 11 mar. 2011.

SEILER, L.; CARMEAN, D.; SPRANGLE, E.; FORSYTH, T.; ABRASH, M.; DUBEY, P.; JUNKINS, S.; LAKE, A.; SUGERMAN, J.; CAVIN, R.; ESPASA, R.; GROCHOWSKI, E.; JUAN, T.; HANRAHAN, P. Larrabee: a many-core x86 architecture for visual computing. **ACM Transactions on Graphics**, [S.l.], v.27, n.3, p.18:1–18:15, 2008.

SHEWCHUK, J. R. **An Introduction to the Conjugate Gradient Method Without the Agonizing Pain**. 1994. Technical Report — .

SONG, F.; YARKHAN, A.; DONGARRA, J. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. In: SC'09 THE INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2009. **Anais...** Portland, OR: s.n.], 2009. p.1–10.

SOUSA, M. S.; MELO, A. C. M. A.; BOUKERCHE, A. An adaptive multi-policy grid service for biological sequence comparison. **J. Parallel Distrib. Comput.**, [S.l.], v.70, p.160–172, February 2010.

SPAFFORD, K.; MEREDITH, J.; VETTER, J. Maestro: data orchestration and tuning for opencl devices. In: EURO-PAR CONFERENCE ON PARALLEL PROCESSING: PART II, 16., 2010, Ischia, Italy. **Proceedings...** Berlin, Heidelberg: Springer-Verlag, 2010. p.275–286. (Euro-Par'10).

STAM, J. Stable fluids. In: SIGGRAPH '99: PROCEEDINGS OF THE 26TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1999. **Anais...** [S.l.: s.n.], 1999. p.121–128.

STONE, J. E.; GOHARA, D.; SHI, G. OpenCL: a parallel programming standard for heterogeneous computing systems. **Computing in Science and Engineering**, [S.l.], v.12, p.66–73, 2010.

TAKIZAWA, H.; KOYAMA, K.; SATO, K.; KOMATSU, K.; KOBAYASHI, H. CheCL:transparent checkpointing and process migration of opencl applications. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2011., 2011. **Anais...** Los Alamitos, CA, USA: IEEE Computer Society, 2011.

TAKIZAWA, H.; SATO, K.; KOBAYASHI, H. SPRAT: runtime processor selection for energy-aware computing. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 2008., 2008, Tsukuba, Japan. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2008. p.386–393.

THIBAULT, J.; SENOCAK, I. CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. In: AIAA AEROSPACE SCIENCES MEETING, 47., 2009. **Anais...** [S.l.]: American Institute of Aeronautics and Austronautics, 2009. p.1–15.

TOMOV, S.; NATH, R.; LTAIEF, H.; DONGARRA, J. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In: INTERNATIONAL WORKSHOP ON HIGH-LEVEL PARALLEL PROGRAMMING MODELS AND SUPPORTIVE ENVIRONMENTS, 15., 2010. **Anais...** Atlanta, GA: s.n.], 2010. p.1–8.

TOPCUOGLU, H.; SOCIETY, I. C.; WU, M. you; MEMBER, S. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.13, p.260–274, 2002.

TSOI, K. H.; LUK, W. Axel: a heterogeneous cluster with fpgas and gpus. In: ACM/SIGDA INTERNATIONAL SYMPOSIUM ON FIELD PROGRAMMABLE GATE ARRAYS, 18., 2010, Monterey, California, USA. **Proceedings...** [S.l.: s.n.], 2010. p.115–124. (FPGA '10).

VOLKOV, V.; DEMMEL, J. W. Benchmarking GPUs to tune dense linear algebra. In: SC '08: PROCEEDINGS OF THE 2008 ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2008, Austin, Texas. **Anais...** Piscataway, NJ, USA: IEEE Press, 2008. p.1–11.

WANG, C.; YU, H.; MA, K.-L. Application-Driven Compression for Visualizing Large-Scale Time-Varying Data. **IEEE Computer Graphics and Applications**, [S.l.], v.30, n.1, p.59–69, 2010.

WANG, P. H.; COLLINS, J. D.; CHINYA, G. N.; JIANG, H.; TIAN, X.; GIRKAR, M.; YANG, N. Y.; LUEH, G.-Y.; WANG, H. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. **SIGPLAN Notes**, New York, NY, USA, v.42, p.156–166, 2007.

WEHRMEISTER, M. A. **An aspect-oriented model-driven engineering approach for distributed embedded real-time systems**. 2009. Tese (Doutorado) — Fakultät für Informatik, Universität Paderborn, 2009.

WEHRMEISTER, M.; FREITAS, E.; PEREIRA, C.; WAGNER, F. Applying Aspect-Orientation Concepts in the Model-Driven Design of Distributed Embedded Real-Time Systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT/COMPONENT/SERRVICE-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 10., 2007. **Anais...** Los Alamitos, CA, USA: IEEE Computer Society, 2007. p.221–230.

WHITE, J. B.; DONGARRA, J. Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. In: IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 2011., 2011. **Anais...** Los Alamitos, CA, USA: IEEE Computer Society, 2011.

YEH, D.; PEH, L.-S.; BORKAR, S.; DARRINGER, J.; AGARWAL, A.; HWU, W. mei. Thousand-Core Chips. **IEEE Design and Test of Computers**, [S.l.], v.25, p.272–278, 2008.

ZHANG, Y.; COHEN, J.; OWENS, J. D. Fast tridiagonal solvers on the GPU. In: PPOPP '10: PROCEEDINGS OF THE 15TH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2010, Bangalore, India. **Anais...** [S.l.: s.n.], 2010. p.127–136.

APPENDIX A USING THE SYSTEM IN CODE EXAMPLES

The Appendix provides the main developed codes used by the solvers in the GPU approach using CUDA and an example of the utilization of the system applied to a heat equation case study using the OpenCL.

A.1 CUDA Implementation of the Main Modules Used by the Solvers

The code for building the banded matrix:

```
#ifndef _BUILDMATRIX_KERNEL_CU_
#define _BUILDMATRIX_KERNEL_CU_

#include "cutil_math.h"
#include "solver.cuh"

#include "common_devicefunctions.cuh"

__global__
void buildBandedMatrixKernel (
    float* A0, float* A1, float* A2, float* A3,
    float* A4, float* A5, float* A6,
    bool curved,
    int* boundaries,
    float* fluidVol,
    uint3 dim)
{
    unsigned int lx = dim.x;
    unsigned int ly = dim.y;
    unsigned int lz = dim.z;
    unsigned int limit = lx*ly*lz;

    int index  = blockIdx.x * blockDim.x + threadIdx.x;

    // cut CUDA grid
    if(index >= limit)
        return;

    int3 pos = get3DPosition(index, lx, ly, lz);
    int ix = pos.x;
    int iy = pos.y;
    int iz = pos.z;

    // we do not solve for the boundary box
```

```
if( ix == 0 || iy == 0 || iz == 0 || ix == lx-1
    || iy == ly-1 || iz == lz-1 )
{
    A0[index] = 0.0f;
    A1[index] = 0.0f;
    A2[index] = 0.0f;
    A3[index] = 1.0f;
    A4[index] = 0.0f;
    A5[index] = 0.0f;
    A6[index] = 0.0f;
    return;
}

if( ix > 0 && iy > 0 && iz > 0 && ix < lx-1
   && iy < ly-1 && iz < lz-1 )
{
    if(!curved)
    {
        // obstacles == 1
        int m = boundaries[index] > 0 ? 1 : 0;
        int v = boundaries[getLinear(
                ix+1,iy+0,iz+0,lx,ly,lz)] > 0 ? 1 : 0;
        int h = boundaries[getLinear(
                ix-1,iy+0,iz+0,lx,ly,lz)] > 0 ? 1 : 0;
        int o = boundaries[getLinear(
                ix+0,iy+1,iz+0,lx,ly,lz)] > 0 ? 1 : 0;
        int u = boundaries[getLinear(
                ix+0,iy-1,iz+0,lx,ly,lz)] > 0 ? 1 : 0;
        int l = boundaries[getLinear(
                ix+0,iy+0,iz+1,lx,ly,lz)] > 0 ? 1 : 0;
        int r = boundaries[getLinear(
                ix+0,iy+0,iz-1,lx,ly,lz)] > 0 ? 1 : 0;

        // build Matrix
        A0[index] = (float) (r - 1);
        A1[index] = (float) (u - 1);
        A2[index] = (float) (h - 1);
        A3[index] = (float) (6 - (l+r+o+u+v+h));
        A4[index] = (float) (v - 1);
        A5[index] = (float) (o - 1);
        A6[index] = (float) (l - 1);

        // do not solve for obstacles
        if(m)
        {
            A0[index] =  0.0f;
            A1[index] =  0.0f;
            A2[index] =  0.0f;
            A3[index] =  1.0f;
            A4[index] =  0.0f;
            A5[index] =  0.0f;
            A6[index] =  0.0f;
        }
    }
    else     // curved boundaries
    {
        // obstacles
        float m = boundaries[index] == 1 ? 1 : 0;
```

```
            float c = fluidVol[index];
            float v = fluidVol[getLinear(ix+1,iy+0,iz+0,
                                         lx,ly,lz)];
            float h = fluidVol[getLinear(ix-1,iy+0,iz+0,
                                         lx,ly,lz)];
            float o = fluidVol[getLinear(ix+0,iy+1,iz+0,
                                         lx,ly,lz)];
            float u = fluidVol[getLinear(ix+0,iy-1,iz+0,
                                         lx,ly,lz)];
            float l = fluidVol[getLinear(ix+0,iy+0,iz+1,
                                         lx,ly,lz)];
            float r = fluidVol[getLinear(ix+0,iy+0,iz-1,
                                         lx,ly,lz)];

            // build Matrix
            A0[index] = -(r + c) / 2.0f;
            A1[index] = -(u + c) / 2.0f;
            A2[index] = -(h + c) / 2.0f;
            A3[index] =  (l+r+o+u+v+h+6.0f*c) / 2.0f;
            A4[index] = -(v + c) / 2.0f;
            A5[index] = -(o + c) / 2.0f;
            A6[index] = -(l + c) / 2.0f;

            // do not solve for obstacles
            if(m)
            {
                A0[index] =  0.0f;
                A1[index] =  0.0f;
                A2[index] =  0.0f;
                A3[index] =  1.0f;
                A4[index] =  0.0f;
                A5[index] =  0.0f;
                A6[index] =  0.0f;
            }
        }
    }
}
#endif
```

The basic reduction kernel:

```
__global__ void
reduce(float* x, float *store, int limit)
{
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int access = bid * blockDim.x + tid;

    __shared__ float xs[BLOCKSIZE];

    //Load
    if( access < limit)
    {
        xs[tid] = x[access];
    }
    else xs[tid] = 0;
    __syncthreads();

    //Reduce current block, store in x1s[0]
```

```
    int div = 2;
    int stride = BLOCKSIZE / div;
    while (div <= BLOCKSIZE)
    {
        if(access < limit && tid < BLOCKSIZE / div)
        {
            xs[tid] += xs[tid + BLOCKSIZE / div];
        }
        div *= 2;
        __syncthreads();
    }

    //Store x1s[0] in temp[bid]
    if(tid == 0)
    {
        store[bid] = xs[0];
    }
}
```

The matrix-vector multiplication kernel:

```
#ifndef _MATRIX_VECTOR_MULTIPLY_KERNEL_CU_
#define _MATRIX_VECTOR_MULTIPLY_KERNEL_CU_

#include "solver.cuh"

__global__ void
matrixVectorMultiply (
    float* A0, float* A1, float* A2, float* A3,
    float* A4, float* A5, float* A6,
    float* x1,
    float* b,
    uint3 dim)
{
    unsigned int lx = dim.x;
    unsigned int ly = dim.y;
    unsigned int lz = dim.z;

    unsigned int limit = lx * ly * lz;

    int tid = threadIdx.x;
    int bid = blockIdx.x;

    int access = bid * BLOCKSIZE + tid;

    // cut CUDA grid
    if(access >= limit)
        return;

    __shared__ float x1s[BLOCKSIZE +2];

    x1s[tid + 1] = x1[access + lx * ly];
    if(tid == 0)
    {
        x1s[tid] = x1[bid * BLOCKSIZE + - 1+ lx * ly];
        x1s[BLOCKSIZE + 1 + tid] = x1[bid * BLOCKSIZE +
            BLOCKSIZE+ lx * ly];
    }
```

```
    __syncthreads();

    float x2 = 0;

    //Left, Right
    x2 += A2[access] * x1s[tid +1 -1];
    x2 += A4[access] * x1s[tid +1 +1];
    //Up, Down
    x2 += A1[access] * x1[access - lx + lx*ly];
    x2 += A5[access] * x1[access + lx + lx*ly];
    //Front, Rear
    x2 += A0[access] * x1[access - lx*ly + lx*ly];
    x2 += A6[access] * x1[access + lx*ly + lx*ly];

    //Center
    x2 += A3[access] * x1s[tid +1];

    b[access] = x2;

}
#endif // MATRIX_VECTOR_MULTIPLY_H
```

The code for the Jacobi solver kernel:

```
#ifndef _JACOBI_KERNEL_CU_
#define _JACOBI_KERNEL_CU_

#include "solver.cuh"
#include "common_devicefunctions.cuh"

// one jacobi-step for solving A*x = rhs
__global__ void
jacobiStep (float* dA0, float* dA1, float* dA2, float* dA3,
            float* dA4, float* dA5, float* dA6,
            float* dX, float* dRhs,
            float* dX_old,
            uint3 dim)
{
    unsigned int lx = dim.x;
    unsigned int ly = dim.y;
    unsigned int lz = dim.z;

    unsigned int limit = lx * ly * lz;

    int bid = blockIdx.x;
    int tid = threadIdx.x;

    int index = bid * blockDim.x + tid;

    if (index >= limit)
        return;

    //solve only inner matrix
    int3 pos = get3DPosition(index, dim);

    // not solving for the boundary
    if(pos.x < 1 || pos.x > lx-2 || pos.y < 1 ||
        pos.y > ly-2 || pos.z < 1 || pos.z > lz-2)
```

```
        return;

    __syncthreads();

    //**** compute Jacobi Step ****

    float result = dRhs[index];

    //Left, Right
    result -= dA2[index] * dX_old[index - 1 + lx*ly];
    result -= dA4[index] * dX_old[index + 1 + lx*ly];

    //Up, Down
    result -= dA1[index] * dX_old[index - lx + lx*ly];
    result -= dA5[index] * dX_old[index + lx + lx*ly];

    //Front, Rear
    result -= dA0[index] * dX_old[index - lx*ly + lx*ly];
    result -= dA6[index] * dX_old[index + lx*ly + lx*ly];

    result /=  dA3[index];

    // X_NEW and swap
    dX[index + lx*ly] = result;
}
#endif
```

The code for the Red-black Gauss-Seidel solver kernel, emphasizing the host (CPU) management of the red and black kernels since the rest of the solution is similar to the Jacobi:

```
//Iteration loop on the Host CPU
for (i=1; i <= max; i++) {
    // Launch kernel to update red squares
    red_kernel<<<dimGrid, dimBlock>>>
    (T_old_d,an_d,as_d,ae_d,aw_d,ap_d,imx,jmx);

    // Launch kernel to update black squares
    black_kernel<<<dimGrid, dimBlock>>>
    (T_old_d,an_d,as_d,ae_d,aw_d,ap_d,imx,jmx);
}
```

The GPU red kernel of the Red-black Gauss-Seidel:

```
//thread indices (tx,ty)
int tx = blockIdx.x * BLOCK_SIZE + threadIdx.x;
int ty = blockIdx.y * BLOCK_SIZE + threadIdx.y;

//thread-grid mapping
row = (ty+1);
col = (tx+1);
```

The code for the Conjugate Gradient solver kernel:

```
#ifndef CONJUGATE_GRADIENT_KERNEL_CU
#define CONJUGATE_GRADIENT_KERNEL_CU

__device__ float r[UNKNOWNS];
__device__ float y[UNKNOWNS + 2 * CVX * CVY];
```

```
__device__ float AY[UNKNOWNS];
__device__ float temp[UNKNOWNS];

__global__ void
conjugateGradient(float* A0,float* A1,float* A2, float* A3,
                  float* A4,float* A5,float* A6,
                  float* x,
                  float* b,
                  int limit)
{

    int iterations = MAXITERATIONS;
    float alpha = 0;
    float beta = 0;
    float betaOld = 0;
    float tempScalar = 0;

    float* x_Start = x + CVX *CVY;
    float* y_Start = y + CVX *CVY;

    //Initialize
    initX(x, CVX * CVY);
    initX(x+ UNKNOWNS + CVX *CVY, CVX *CVY);
    initX(y, CVX * CVY);
    initX(y+ UNKNOWNS + CVX *CVY, CVX *CVY);

    // r = b - A*x
    matrixVectorMultiply(A0,A1,A2,A3,A4,A5,A6,x,AY,limit);
    __syncthreads();
    vectorVectorAdd(b,AY,r,SUB,limit);
    __syncthreads();

    // y = r
    vecCpy(y_Start,r,limit);
    __syncthreads();

    // beta = r*r
    vectorVectorMult(r,r,beta,MULT,limit);
    __syncthreads();

    //Iterate
    #pragma unroll
    for(int k = 0 ; k < iterations; k++)
    {
        betaOld = beta;

        //Compute AY = A*y
        matrixVectorMultiply(A0,A1,A2,A3,A4,A5,A6,y,AY,limit);
        __syncthreads();

        // alpha = beta / (y * A*y)
        vectorVectorMult(y_Start,AY,tempScalar,MULT,limit);
        __syncthreads();
        alpha = beta / tempScalar;

        // x = x + alpha * y
        vectorScalar(y_Start , alpha, temp, MULT, limit);
        __syncthreads();
```

```
        vectorVectorAdd(x_Start,temp,x_Start,ADD,limit);
        __syncthreads();

        // r = r - alpha * A*y
        vectorScalar(AY,alpha,temp,MULT,limit);
        __syncthreads();
        vectorVectorAdd(r,temp,r,SUB,limit);
        __syncthreads();

        // beta = r * r
        vectorVectorMult(r,r,beta,MULT,limit);
        __syncthreads();

        // y = r + beta * y/beta
        vectorScalar(y_Start,betaOld,temp,DIV,limit);
        __syncthreads();
        vectorScalar(y_Start,beta,temp,MULT,limit);
        __syncthreads();
        vectorVectorAdd(r,temp,y_Start,ADD,limit);
        __syncthreads();
        }
}
#endif
```

A.2 Example of Using the Sm@rtConfig System

Bellow, a basic example of using the allocation method that represents one of the described heuristics. The method, however, need to be explicitly called on the C++ code. To be fully integrated on the OpenCL API, all tasks should be represented as OpenCL kernels and the Allocation method should be *transparent* to the programmer. This way, the Allocation method should be a new OpenCL method that assigns the instantiated kernels.

It is important to note that just the main parts of the code are emphasized bellow:

```
Caller* callGPU = new Caller(for_GPU,g,database);
callGPU->init_OpenCL(); // create contexts
Caller* callCPU = new Caller(for_CPU,c,database);
omp_set_nested(1); // each caller is an OpenMP thread

Jacobi task;
task.initialize(domain_size);
database->find(task.domain_size,&cost_cpu,&cost_gpu);
n_tasks->add_task(task)

//instantiation of other tasks
//...

Allocation* allocate =
    new Allocation(cost_cpu,cost_gpu,
                   n_tasks,
                   cost_cpu_done,cost_gpu_done);

//...
//execute tasks
for(int i = 0; i < n_tasks.size(); i++){
```

```
    // open the pre-compiled PTX file and load it
    // to substitute the OpenCL code to the tuned ones
    FILE* fp = fopen(n_tasks[i].name(), "r");
    fseek (fp , 0 , SEEK_END);
    const size_t lSize = ftell(fp);
    rewind(fp);
    unsigned char* buffer = (unsigned char*) malloc (lSize);
    fread(buffer, 1, lSize, fp); //binaries
    fclose(fp);

    //...

    program = clCreateProgramWithBinary(context,
                            1,
                            &n_tasks.device(),
                            &lSize,
                            (const unsigned char**)&buffer,
                            &status, &err);

        err = clBuildProgram(program,0,NULL,NULL,NULL,NULL);

    //...

    task = clCreateKernel(program,n_tasks[i].name(),&err);

    //...
}
```

APPENDIX B HANDLING DYNAMIC SCHEDULING OVER A CPU-GPU PLATFORM USING AN ASPECT-ORIENTED APPROACH

In order to achieve a more transparent dynamic reconfigurable load-balancing, it is being investigated the use of aspect-oriented paradigms (from software engineering). The way of implementing the presented scheduling concepts as aspects, concentrating on non-functional requirements specifications that are related with tasks' execution times, may leverage from the users the explicit control of load balancing, leaving them to concentrate on the development of functional requirements of the high-level tasks.

Distributed real-time embedded systems concepts are used as base to this ongoing work and have key non-functional requirements concerning their development. Figure B.1 presents some of these key requirements developed on the DERAF framework, which are mainly based on the study presented in (BURNS; WELLINGS, 1997) and on the IEEE glossary (IEEE, 2006).

The *real-time* concern is captured by the requirements stated in the Time classification, which is divided in Timing and Precision requirements. The first concerns with the specification of temporal limits for system activities execution, such as established deadlines and periodic activations. Requirements classified as Precision denote constraints that affect the temporal behavior of the system in a "fine-grained" way, determining whether a system has hard or soft time constraints. An example is the Freshness requirement, which denotes the time interval within which value of a sampled data is considered updated. Another key requirement is the Jitter, which directly affects the system predictability because of large variance that degrades system determinism.

The Performance requirements are tightly related to those presented in the Time classification. However, these requirements have also an important relation with those concentrated in the Distribution classification, leading to decide on putting them in a distinct category. They represent requirements usually employed to express a global need of performance, like the end-to-end response time for a certain activity performed by the system and the required throughput rate.

The goal of Distribution classification is to identify key requirements related to the distribution of distributed real-time system activities, which usually execute concurrently. For instance, these concerns address problems such as task allocation over the system processing units, as well as the communication needs and constraints. Concerns related to embedded systems generally present requirements related to memory usage, energy consumption, and required hardware area size. The Embedded classification deals with

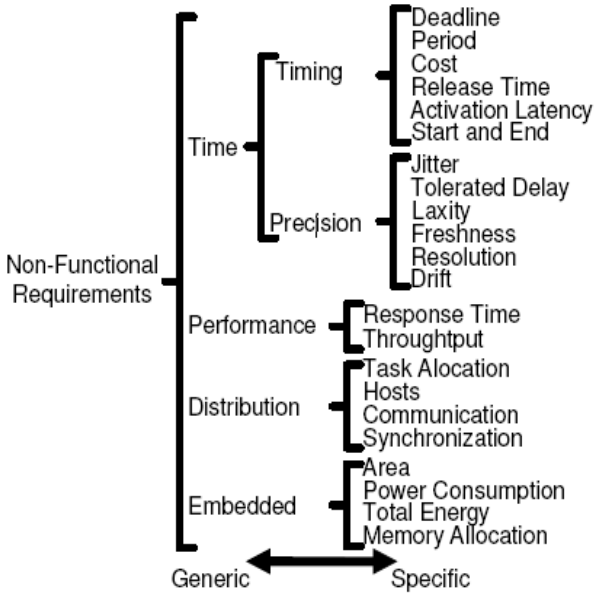


Figure B.1: NFR requirements for the DERAF distributed real-time embedded systems

the monitoring and controlling of these three issues.

In this extension of the DERAF, the interest is to provide a reconfigurable scheduling solution in runtime with the goal to meet timing requirements. It is clear that all the mechanisms related to the re-scheduling of tasks among processing units, which implements the system reconfiguration, are non-functional crosscutting concerns. This means that the reconfiguration of tasks is not a final functional behavior of any system, but it affects several elements in different ways and in different parts of the system.

This way, design aspects to handle the Time classification issues are used. These aspects model the introduction of time parameters in the system elements. Additionally, a contribution is added by proposing new aspects that extend the ideas referred on that work to handle the scheduling and reconfiguration concerns. In order to support Timing and Precision requirements, the proposal is to use aspects from the DERAF framework (WEHRMEISTER, 2009). The packages designed for these types of requirements are presented in Figure B.2. DERAF stands for Distributed Embedded Real-time Aspects Framework. It is an extendable high-level aspects framework for distributed embedded real-time systems design that provides a set of aspects to cover the needs of handling of those NFR. Interested readers are pointed also to (WEHRMEISTER et al., 2007; FREITAS et al., 2008a) for more details about DERAF.

Bellow, a short description of each package is provided:

- TimingAttributes adds timing attributes to active objects (e.g., deadline, priority, start/end time, among others), and also the corresponding initialization of these attributes.

- PeriodicTiming adds a periodic activation mechanism to active objects. This improvement requires the addition of an attribute representing the activation period and a way to control the execution frequency according to this period.

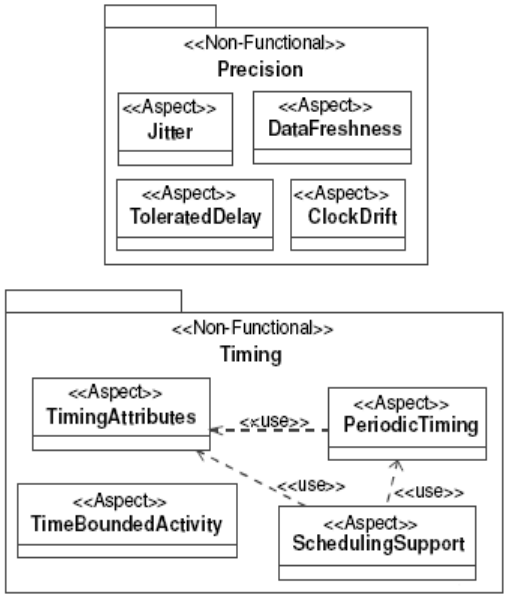


Figure B.2: Timing and Precision packages from DERAF

- SchedulingSupport: inserts a scheduling mechanism to control the execution of active objects. Additionally, this aspect handles the inclusion of active objects into the scheduling list, as well as the execution of the feasibility test to verify if the scheduling list is schedulable.
- TimeBoundedActivity temporally limits the execution of an activity, i.e., adds the mechanism to restrict the maximum execution time for an activity (e.g., limits the time which a shared resource can be locked by an active task).
- Jitter measures the start/end of an activity, calculates the variation of this metrics and whether the tolerated variance was overran.
- ToleratedDelay temporally limits the beginning of an activity execution (e.g., limits the time which an active task can wait to acquire a lock on a shared resource).
- DataFreshness associates timestamps to data, verifying their validity before their use. Every time after that "validity controlled data" are written, the timestamp must be updated. Analogously, before reading them, the timestamps must be checked.
- ClockDrift measures the time at which an activity starts and compares it with the expected beginning of this activity; it also checks if the accumulated difference exceeds the maximum tolerated clock drift.

As mentioned before, the task reconfiguration support characterizes a non-functional crosscutting concern that spread the handling mechanisms over several system elements in a non-standard way. Based on that concept, it was proposed the use of aspects to address this concern. The new proposed aspects packages are: *Reconfiguration* and *TaskAllocation*. They use the time parameters inserted by the aspects of the Timing package, described previously, and the services provided by the aspects from the Precision. Figure B.3 depicts the schema.

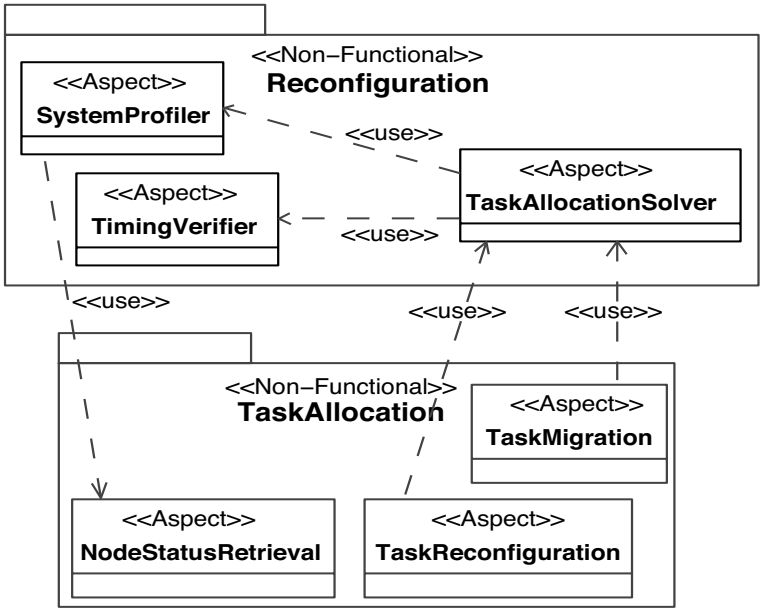


Figure B.3: Aspects for the scheduling system included in the DERAF framewrok

On the Reconfiguration package, the TimingVerifier is responsible for checking if the processing units are being able to accomplish with the timing requirements specified by the TimingAttributes, PeriodicTiming, ToleratedDelay and TimeBoundedActivity aspects. In order to perform that activity, it is intended to use the services from the aspects Jitter and ClockDrift.

A mechanism to control the meeting of timing attributes is inserted in the beginning and end of each task. This mechanism consists of measuring these times, comparing them with the requirement specified by the correspondent attribute. As an example, the accomplishment of a specified deadline can be checked by measuring the time in which the task actually ended its computation and comparing it with the time in which it was supposed to finish (using the history database). It uses the service of the Jitter aspects to gather information about the jitter related to the correspondent analyzed requirement. Taking the deadline again as an example, it measures if a non-accomplishment of a deadline is constant or if the measure varies in different executions or in changing the platform scenario. It can be used, for instance, as base information to know if the interaction with other tasks is being responsible for the variance.

The ClockDrift aspect is used by the TimingVerifier to gather information about the synchronization among the different PUs. It is useful to calculate the cost, in terms of a future task migration time. In order to illustrate the idea, consider a task that was migrated from a PU "A" to a PU "B". The difference in the clock drift between them can result in a waiting time for the result from the PU "B" that does not worth if compared with leaving the task running in the PU "A".

The second key aspect is the TaskAllocationSolver. It is responsible for deciding the scheduling and if a task will be reconfigured or not to the available PUs. It also has to consult overload of the PU candidate destinations, in order to decide if it is worthwhile to perform the reconfiguration. The TaskAllocationSolver uses the measurements available due to the work done by the TimingVerifier and SystemProfiler .

Based on these data, the reasoning about the feasibility or not of a task reconfiguration is performed by TaskReconfiguration. The reconfiguration itself and the retrieval of PUs (nodes) status are done by two other aspects from DERAF, the TaskMigration and the NodeStatusRetrieval. This way, the reasoning and the performance of the reconfiguration are decoupled, allowing that changes performed in one aspect do not affect the other. A brief summary of the TaskMigration, NodeStatusRetrieval, and TaskReconfiguration aspects is provided in the following:

- TaskMigration provides a mechanism to migrate active objects (tasks) from one PU node to another PU node. It is used by the aspects that control embedded concerns and, in the ongoing work, by the allocation solver aspect TaskAllocationSolver.
- TaskReconfiguration offers a mechanism to reallocate tasks that were previously allocated but still not executed. It is also used by the allocation solver aspect TaskAllocationSolver.
- NodeStatusRetrieval inserts a mechanism to retrieve information about processing load, send/receive message rate, and/or the PU availability (i.e., the "I'm alive" message). Before/after every execution start/end of an active object (task), the processing load is calculated. Before/after every sent/received message, the message rate is computed. Additionally, the PU availability message is sent at every "n" message or periodically with an interval of "n" time units.

APPENDIX C UM SISTEMA DE ESCALONAMENTO DINÂMICO E TUNING EM TEMPO DE EXECUÇÃO PARA PLATAFORMAS DESKTOP HETEROGÊNEAS DE MÚLTIPLOS NÚCLEOS

Atualmente, o computador pessoal (PC) moderno poder ser considerado como um *cluster* heterogênedo de um nodo, o qual processa simultâneamente inúmeras tarefas provenientes das aplicações. O PC pode ser composto por Unidades de Processamento (PUs) assimétricas, como a Unidade Central de Processamento (CPU), composta de múltiplos núcleos, a Unidade de Processamento Gráfico (GPU), composta por inúmeros núcleos e que tem sido um dos principais co-processadores que contribuiram para a computação de alto desempenho em PCs, entre outras. Neste sentido, uma plataforma de execução heterogênea é formada em um PC para efetuar cálculos intensivos em um grande número de dados. Na perspectiva desta tese, a distribuição da carga de trabalho de uma aplicação nas PUs é um fator importante para melhorar o desempenho das aplicações e explorar tal heterogeneidade. Esta questão apresenta desafios uma vez que o custo de execução de uma tarefa de alto nível em uma PU é não-determinístico e pode ser afetado por uma série de parâmetros não conhecidos a priori, como o *tamanho do domínio do problema* e a *precisão* da solução, entre outros.

Nesse escopo, esta pesquisa de doutorado apresenta um *sistema sensível ao contexto e de adaptação em tempo de execução* com base em um compromisso entre a redução do tempo de execução das aplicações - devido a um escalonamento dinâmico adequado de tarefas de alto nível - e o custo de computação do próprio escalonamento aplicados em uma plataforma composta de CPU e GPU. Esta abordagem combina um modelo para um *primeiro escalonamento* baseado em perfis de desempenho adquiridos em pré-processamento com um *modelo online*, o qual mantém o controle do tempo de execução real de novas tarefas e escalona dinâmicamente e de modo eficaz novas instâncias das tarefas de alto nível em uma plataforma de execução composta de CPU e de GPU. Para isso, é proposto um conjunto de heurísticas para escalonar tarefas em uma CPU e uma GPU e uma estratégia genérica e eficiente de escalonamento que considera várias unidades de processamento.

A abordagem proposta é aplicada em um estudo de caso utilizando uma plataforma de execução composta por CPU e GPU para computação de métodos iterativos focados na solução de Sistemas de Equações Lineares que se utilizam de um cálculo de *stencil* especialmente concebido para explorar as características das GPUs modernas. A solução utiliza o número de incógnitas como o principal parâmetro para a decisão de escalonamento.

Ao escalonar tarefas para a CPU e para a GPU, um ganho de 21,77% em desempenho é obtido em comparação com o escalonamento estático de todas as tarefas para a GPU (o qual é utilizado por modelos de programação atuais, como OpenCL e CUDA para Nvidia) com um erro de escalonamento de apenas 0,25% em relação à combinação exaustiva.

Em seguida, são ressaltados os objetivos e as contribuições desta tese.

C.1 Objetivos

Como mencionado, desktops baseados em co-processadores, como as GPUs, são hoje uma alternativa econômica para plataformas de execução que visam um melhor desempenho. Tomando um exemplo, a empresa Nvidia apresentou sua GPU GTX285 que obtém um desempenho de 1062 Gflop/s em precisão simples e 89 Gflop/s em precisão dupla (NVIDIA, 2010b).

Como conseqüência, as plataformas heterogêneas com vários tipos de unidades de processamento agem, em essência, como uma plataforma assimétrica de múltiplos núcleos, podendoe lidar com diversas aplicações e tarefas de forma simultânea, como CFD e as tarefas de solvers para sistemas de equações lineares. Isto é ainda intensificado com as CPUs de múltiplos núcleos, como a Intel Core2Quad que apresenta desempenho de cerca de 100 Gflop/s (INTEL, 2010a). Por isso, *usar de forma eficiente todos os recursos disponíveis da plataforma de execução heterogêneo* é um desafio significativo para aplicações que requerem alto desempenho.

Neste sentido, esta tese tem o objetivo de fornecer metodologias, estratégias e mecanismos que agreguem recursos de alocação e programação de tarefas a serem executadas por sistemas heterogêneos. Assim, as aplicações podem ser dinâmicamente configuradas com base na plataforma assimétrica de execução disponível, a fim de utilizar os recursos computacionais mais apropriados e diminuir o tempo de execução das tarefas.

A fim de se beneficiar do poder de processamento de todas as PUs, o principal objetivo deste trabalho é desenvolver uma estratégia para distribuir o processamento das tarefas nas PUs disponíveis. A estratégia se baseia em um *escalonamento dinâmico* em vez da atual técnica de programação estática utilizada pelo OpenCL (STONE; GOHARA; SHI, 2010) ou, mais especificamente, pelo CUDA (NVIDIA, 2010b) quando aplicado a GPUs da Nvidia (ver também o trabalho de Göddeke et al. (GÖDDEKE et al., 2009)). Esta necessidade se torna ainda mais essencial quando se lida com aplicações que apresentam restrições de tempo de execução, como a aplicação de CFD em tempo real que parcialmente motivou este trabalho.

C.2 Contribuições

O tópico "escalonamento de tarefas em plataformas heterogêneas de múltiplos núcleos" foi identificado como um recente e importante problema em aberto pelo edital de projetos de pesquisa da Comissão Europeia em TIC (Tecnologias de Informação e Comunicação), denominado Programa Quadro 7 (COMMISSION, 2010), e pelo grupo de Arquitetura de Sistemas Embarcados de Alto Desempenho que impulsiona a importância de vários problemas em aberto na área de ciência da computação (DE BOSSCHERE et al., 2007). As referências mencionam explicitamente que a disponibilidade de múlti-

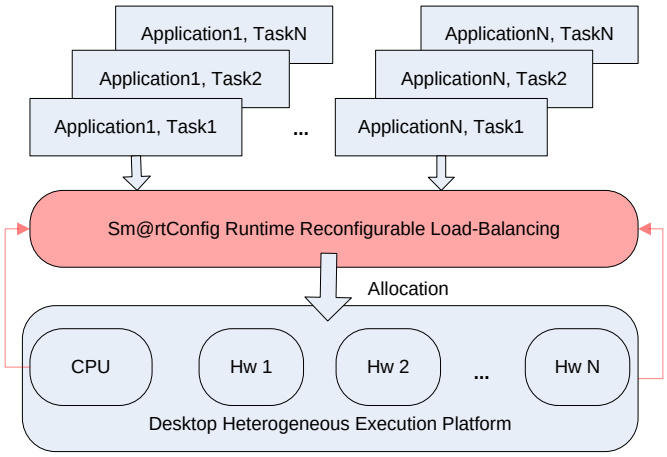


Figure C.1: Visão geral do sistema Sm@rtConfig

plos núcleos é uma tendência e vai integrar até 1000 bilhões de dispositivos até 2020. Também indica que estes dispositivos podem fornecer ordens de magnitude de melhoria de desempenho apenas com concorrência entre as tarefas e com arquiteturas heterogêneas ajustáveis de acordo com aplicações específicas. Além disso, um relatório semelhante feito pelo Conselho de Acessores de Ciência e Tecnologia do Governo dos Estados Unidos da América afirma que os ganhos de desempenho devido a melhorias nos algoritmos (explorando as características dos novos hardwares) ultrapassou os ganhos de desempenho devido à velocidade dos processadors (SCIENCE; TECHNOLOGY, 2010). O relatório também foca a necessidade de ferramentas de gerenciamento da plataforma de execução para a próxima geração de tecnologias de alto desempenho, incluindo a investigação sobre sistemas híbridos de hardware/software e sistemas de software e aplicações. Com base nessas afirmações, esta tese aborda pontualmente métodos de escalonamento em plataformas heterogêneas, especialmente compostas por CPU e GPU.

A Figura C.1 apresenta uma visão geral das contribuições desta tese, onde a plataforma proposta é chamada Sm@rtConfig. Acoplamento a descoberta de recursos de computação das PUs com as características das aplicações, uma análise é realizada para configurar a alocação de tarefas nas unidades de processamento disponíveis. Em tempo de execução, o Sm@rtConfig captura o desempenho das tarefas, alimentando o escalonador com tais dados atuais, para uma eventual nova alocação em outra PU caso esse procedimento promoca um melhor desempenho.

Este sistema apresenta uma nova estratégia para distribuir a carga de trabalho sobre CPU e GPU, sendo genérico suficiente para considerar outras PUs presentes em um desktop. Os métodos de alocação dinâmica combinam uma *primeira fase de assinalamento* de um conjunto de *tarefas em alto nível* (algoritmos, por exemplo) com uma *fase em tempo de execução* que obtém medidas de desempenho real das tarefas, alimentando o *banco de dados de desempenho*. A primeira fase é baseada em dados de de pré-processamento para a aquisição de amostras básicas de tempos de execução das tarefas em cada PU. Desta forma, após o primeiro assinalamento, o sistema considera o histórico apresentado no banco de dados para executar o assinalamento de cada tarefa adicional, maximizando o desempenho das aplicações com balanceamento de carga e sobrecarca mínima.

Em linhas gerais, as principais contribuições desta tese são:

1. O desenvolvimento de um sistema que contempla: (i) uma *primeira fase de escalonamento* de tarefas, (ii) um módulo para *aquisição de tempos de execução* das tarefas o qual alimenta uma *base de dados* que contém estes tempos de execução reais, e (iii) uma *fase em tempo de execução* que realiza o escalonamento dinâmico de novas tarefas baseado no histórico dos tempos de execução previamente armazenados;

2. O desenvolvimento de uma nova estratégia para armazenamento e recuperação de dados, usado pelas tarefas do tipo Métodos para solucionar Sistemas de Equações Lineares, utilizando a GPU com o objetivo de se alcançar coalescência de memória e de se beneficiar da memória compartilhada;

3. Análise das características dos Métodos para a solução de Sistemas de Equações Lineares e seu desempenho em uma plataforma composta por CPU e GPU, caracterizando as condições onde os métodos (tarefas) obtém melhor desempenho em termos de tempo de execução (isto é, caractrizar o *ponto de equilíbrio* que indica a melhor PU a ser utilizada);

4. Implementação e comparação de três métodos iterativos para solucionar os sistemas de equações lineares (Jacobi, Gauss-Seidel e Gradiente Conjugado) na CPU e em múltiplas GPUs, aplicado em uma simulação CFD em tempo real com modificação de geometria do modelo a ser simulado.

APPENDIX D PUBLICATIONS AND ACADEMIC ACTIVITIES

The thesis is partially based on the following publications:

1. Invited to a special section on "Advanced Software Engineering in Industrial Automation" at the IFAC journal "Control Engineering Practice" (impact factor 1.943). Submitted article: Sm@rtConfig: a Context-Aware Runtime and Tuning System for Data Intensive Engineering Applications. Expected publication in early 2012.

2. BINOTTO, A.P.D.; PEREIRA, C.E.; KUIJPER, A.; STORK, A.; FELLNER, D. An Effective Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms. In: 13th IEEE International Conference on High Performance Computing and Communications (HPCC-2011), 2011, Banff, p. 78-85.

3. BINOTTO, A.P.D.; PEDRAS, B.M.V.; GÖTZ, M.; KUIJPER, A.; STORK, A.; PEREIRA, C.E.; FELLNER, D. Effective Dynamic Scheduling on Heterogeneous Multi/Many core Desktop Platforms. In: Workshop on Applications for Multi and Many Core Architectures of the 22nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2010). Los Alamitos, CA, USA : IEEE Computer Society, 2010, Petropolis, p. 37-42.

4. BINOTTO, A.P.D.; DANIEL, C.; WEBER, D.; KUIJPER, A.; STORK, A.; PEREIRA, C.E.; FELLNER, D. Iterative SLE Solvers over a CPU-GPU Platform. In: 12th IEEE International Conference on High Performance Computing and Communications (HPCC-2010), 2010, Melbourne, p. 305-313.

5. BINOTTO, A.P.D.; PEREIRA, C.E.; FELLNER, D. Towards Dynamic Reconfigurable Load-balancing for Hybrid Desktop Platforms. In: PhD Forum of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010). New York : IEEE Computer Society Press, 2010, Sydney, p. 1-4.

6. BINOTTO, A.P.D.; FREITAS, E.P.; WEHRMEISTER, M.A.; PEREIRA, C.E.; STORK, A.; Larsson, T. Towards Task Dynamic Reconfiguration over Asymmetric Computing Platforms for UAVs Surveillance Systems. Scalable Computing. Practice and Experience, v. 10, p. 277-289, 2009.

7. BINOTTO, A.P.D.; FREITAS, E.P.; PEREIRA, C.E.; LARSSON, T. Towards Dynamic Task Scheduling and Reconfiguration using an Aspect Oriented Approach

applied on Real-time concerns of Industrial Systems. In: XIII IFAC Symposium on Information Control Problems in Manufacturing, 2009, Moscow, p. 1406-1411.

8. FREITAS, E.P.; BINOTTO, A.P.D.; PEREIRA, C.E.; STORK, A.; LARSSON, T. Dynamic Reconfiguration of Tasks applied to an UAV System using Aspect Orientation. In: IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA-08), 2008, Sydney, p.292-300.

9. FREITAS, E.P.; BINOTTO, A.P.D. ; PEREIRA, C.E.; STORK, A.; LARSSON, T. Dynamic Reconfigurable Task Schedule Support towards a Reflective Middleware for Sensor Network. In: IEEE International Workshop on Modeling, Analysis and Simulation of Sensor Network (MASSN-08), 2008, Sydney, p. 886-891.

10. BINOTTO, A.P.D.; FREITAS, E.P.; PEREIRA, C.E.; STORK, A.; LARSSON, T. Real-time Task Reconfiguration Support Applied to an UAV-based Surveillance System. In: International Multiconference on Computer Science and Information Technology, 2008, Wisla, v. 3, p.581-588.

11. BINOTTO, A.P.D.; FREITAS, E.P.; GÖTZ, M.; PEREIRA, C.E.; STORK, A.; LARSSON, T. Dynamic Self-Rescheduling of Tasks over a Heterogeneous Platform. In: Reconfig 2008 - International Conference on ReConFigurable Computing and FPGAs, 2008, Cancun, p. 253-258.

12. BINOTTO, A.P.D.; PEREIRA, C.E.; GIERLINGER, T; SANTOS, P. Enhancing Real-Time Engineering Based Simulations with the General Purpose of Graphics Hardware. In: Fischer, X. (Org.). Research in Interactive Design: proceedings of Virtual Concept 2006. Heidelberg: Springer-Verlag, 2006, pp.4.

13. BINOTTO, A.P.D.; BRUNETTI, G.; PEREIRA, C.E.; SANTOS, P. Computer Graphics Applications in Virtual Engineering. In: Fischer, X. (Org.). Rsearch in Interactive Design: proceedings of Virtual Concept 2006. Heidelberg: Springer-Verlag, 2006, pp.15.

Additionally, the thesis' author was involved in the following academic activities:

1. Reviewer of the Computers & Graphics, Special Section VR in Brazil, 2011.

2. Auxiliary reviewer of the International Workshop on Wireless Networking for Unmanned Aerial Vehicles (Wi-UAV), 2010.

3. Program Committee Member of the 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2009.

4. Reviewer of the IX Brazilian Symposium on Automatics (SBAI), 2009.

5. Auxiliary reviewer of the 13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM), 2009.

6. Auxiliary reviewer of the book Informatics in Control, Automation, and Robotics, published by Springer-Verlag, 2009.

7. Organizer of the Virtual Concept Summer School, Brazil, 2006.

APPENDIX E SUPERVISING ACTIVITIES

The following list summarizes the student bachelor thesis and the practical work supervised by the author, both on Computational Engineering at the Technische Universität Darmstadt. The results of these works were partially used as an input into this thesis.

E.1 Bachelor Thesis

1. Christian G. Daniel. On the Acceleration of CFD Simulations using a GPU Approach. 2009. Bachelor in Computational Engineering. Technische Universität Darmstadt.

E.2 Practical Work

1. Bernardo Meierfrankenfeld Villela Pedras. Dynamic Scheduling of Solvers for Systems of Linear Equations over a CPU-GPU Platform. 2010. Practical Work required for the Master in Computational Engineering. Technische Universität Darmstadt.

APPENDIX F AUTHOR'S CURRICULUM VITAE

Alécio Pedro Delazari Binotto
Curriculum Vitae

Personal Data

Name Alécio Pedro Delazari Binotto
Address Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Prédio 72, Setor 4 - Sala 119
91501-970 - Porto Alegre
Brasil
Email and Phone abinotto@inf.ufrgs.br
+55 51 3308 7020 / +55 51 8108 2267 (mobile)
Gender Male
Birth date and Place October 23rd, 1979, Belém-Pará
Citizenship Brazilian

Education

04.2006 – 06.2011 Ph.D. in Computer Science in a joint cooperation between the Federal University of Rio Grande do Sul and the Technische Universität Darmstadt
"A Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi- and Many-Core Desktop Platforms"

03.2001 – 03.2003 M.Sc. in Computer Science, Federal University of Rio Grande do Sul.
"Real-time Visualization of Dynamic Volumetric Data using the Graphics Hardware"
Grade: A unanimously

02.1997 – 12.2000 Information Systems, University Center of Pará
"Limits: an educational software for teaching Limits using the rediscovery technique"
Grade: 9.38 (in a range from 0 to 10)

Professional positions

1. Federal University of Rio Grande do Sul, Brazil

07.2011 - Post-doc Researcher

04.200–06.2011 Ph.D. Researcher, in a binational-degree agreement with the Technische Universität Darmstadt, Germany, with focus on high-performance computing on heterogeneous desktop platforms. Advisor: Prof. Dr.-Ing. Carlos Eduardo Pereira

03.2001–03.2003 Researcher on the MAPEM Project for the off-shore oil industry with focus on Real-time Visualization of CFD large data

2. i9access Tecnologia Ltda., Brazil

04.2009 - Partner, co-founder
ICT Spin-off Company focused on innovative Telemedicine and eHealth solutions
www.i9access.com.br

3. Fraunhofer Institut für Graphische und Datenverarbeitung – IGD, Germany

12.2007 – 10.2010 PhD Researcher at the Industrial Applications Department, with focus on applied research projects using concepts of scheduling and CFD simulation. Advisors: Prof.

Dr.-Ing. André Stork and Prof. Dr. techn. Dieter W. Fellner
www.igd.fraunhofer.de / www.fraunhofer.com.br

4. Center of Excellence on Advanced Technologies – CETA , Brazil

01.2004 – 11.2007 Technological Coordinator
Coordinated seven research projects (four international in the area of Telemedicine acting as local coordinator) with technology transfer focus.
www.fiergs.org.br

5. Osório Faculty – FACOS, Brazil

02.2005 – 12.2006 Undergraduate Lecturer
www.facos.edu.br

6. Brandenburg GmbH, Germany

03–10.2003 AIESEC Trainee in software development for automobile light simulation
www.brandenburg-gmbh.de

7. Pará State Bank, Brazil

03.1999–02.2001 Trainee and Junior Programmer for bank automation systems
www.banparanet.com.br

Languages

Portuguese	Native speaker
English	Fluent
German	Good knowledge
Spanish	Good knowledge

Awards

Intel	Finalist of the Intel Challenge – Brazilian Phase – for start-ups, 2011
IEEE	Winner of the IEEE Presidents' Change the World Competition, 2011
Santander	Winner of the Rio Grande do Sul State phase - Entrepreneurship Santander Prize, 2010
UFRGS	Winner of the X UFRGS Entrepreneurship Marathon – best business plan, 2009
DAAD	Ph.D. scholarship, n. A/07/70158, 2007-2010
Alßan	Ph.D. scholarship, n. E07D402961BR, 2008-2010

Other relevant information

Detailed academic CV with a complete list of publications and projects' descriptions can be found online at: http://lattes.cnpq.br/0812941211957547 .