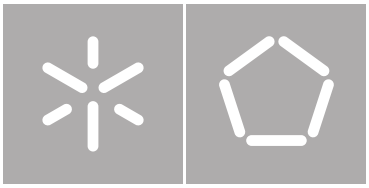
Universidade do Minho
Escola de Engenharia

Artur Miguel Matos Mariano

Scheduling (ir)regular applications on heterogeneous platforms

Setembro de 2012



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Artur Miguel Matos Mariano

Scheduling (ir)regular applications on heterogeneous platforms

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Alberto José Proença
João Garcia Barbosa

Setembro de 2012

Acknowledgments

Ao meu orientador, Alberto Proença, pelo rigor que desde sempre lhe conheci e tão bem imprimiu na orientação que me concedeu, quer durante a construção da minha tese quer na escrita desta dissertação. Ao meu co-orientador, João Barbosa, agradeço a oportunidade.

Ao professor Luís Paulo Santos endereço um particular agradecimento, pela incontestável disponibilidade que mesmo sem lhe competir me demonstrou. Aos docentes que me marcaram durante a minha jornada na Universidade do Minho, que agora finda, onde incluo os professores Alberto Proença, Luís Paulo Santos, José Nuno Oliveira, António Pina, Rui Ralha, Pedro Nuno Sousa e José Bernardo Barros.

Ao LabCG que me acolheu no último ano lectivo e a todo o pessoal que o integra, pelo ambiente fantástico. Ao Ricardo, ao Nuno, ao Waldir, e ao Jaime o meu obrigado. Um especial agradecimento ao Roberto, pela disponibilidade e pelas discussões que mantivemos durante o ano. Aos meus amigos João, Vences, Tiago, Jorge e Faber.

À Universidade do Minho, pela formação de excelência. À Fundação para a Ciência e Tecnologia (FCT) e aos projectos por ela financiados que me concederam duas bolsas de investigação, permitindo-me assim, entre outras coisas, financiar a minha pós-graduação.

To several people from the University of Texas at Austin, including my advisors during my stay at UTexas, Dr. Andreas Gerstlauer and Dr. Derek Chiou. To some other people from UTexas as well, including Dr. Gregory Pogue, Dr. David Gibson and Luís Rodrigues, which were generous in sharing some of their precious time with me. To Kevin, from Michigan.

Mais que tudo, gostaria de agradecer à minha família, aos meus pais, irmão e avós, uma vez que me motivam em cada dia da minha vida. Ao meu falecido tio Joaquim Corte, dedico esta tese.

Em memória de Joaquim Mariano Corte, 1946-2011
(In memory of Joaquim Mariano Corte, 1946-2011)

Reflexion

On June 7 of 1494, John II King of Portugal, the kings of Aragon and Austurias and the queen of Castile signed the treaty of Tordesillas, which divided "the newly discovered lands and the ones yet to be discovered" between the crowns of Portugal and Spain. Also due to this agreement, the Portuguese empire, the first global empire in history, expanded considerably and established itself as the world leading kingdom in economics and military power, during the fifteenth and until the beginning of the sixteenth centuries.

Coat of Arms of the Kingdom of Portugal (1139–1910)

The Portuguese discovered lands and maritime trades that defined the world map as it is known nowadays. The figure below shows the Portuguese discoveries and explorations, but it cannot show the effort they went into discovered (and in some cases conquest). To exploit the unknown is nowadays called **research**, which is precisely what was uniquely done by the Portuguese people more than 500 years ago. This makes of the Portuguese one of the bravest people in history, known worldwide as remarkable sailors and explorers.

Portuguese Discoveries and Explorations

(figures retrieved from Wikipedia.org)

Resumo

As plataformas computacionais actuais tornaram-se cada vez mais heterogéneas e paralelas nos últimos anos, como consequência de integrarem aceleradores cujas arquitecturas são paralelas e distintas do CPU. Como resultado, várias *frameworks* foram desenvolvidas para programar estas plataformas, com o objectivo de aumentar os níveis de produtividade de programação. Neste sentido, a *framework* GAMA está a ser desenvolvida pelo grupo de investigação envolvido nesta tese, tendo como objectivo correr eficientemente algoritmos regulares e irregulares em plataformas heterogéneas.

Um aspecto chave no contexto de *frameworks* congéneres ao GAMA é o escalonamento. As soluções que compõem o estado da arte de escalonamento em plataformas heterogéneas são eficientes para aplicações regulares, mas ineficientes para aplicações irregulares. O escalonamento destas é particularmente complexo devido à imprevisibilidade e às diferenças no tempo de computação das tarefas computacionais que as compõem.

Esta dissertação propõe o *design* e validação de um modelo de escalonamento e respectiva implementação, que endereça tanto aplicações regulares como irregulares. O mecanismo de escalonamento desenvolvido é validado na *framework* GAMA, executando algoritmos científicos relevantes, que incluem a SAXPY, a Transformada Rápida de Fourier e dois algoritmos de resolução do problema n-Corpos. O mecanismo proposto é validado quanto à sua eficiência em encontrar boas decisões de escalonamento e quanto à eficiência e escalabilidade do GAMA, quando fazendo uso do mesmo.

Os resultados obtidos mostram que o modelo de escalonamento proposto é capaz de executar em plataformas heterogéneas com alto grau de eficiência, uma vez que encontra boas decisões de escalonamento na generalidade dos casos testados. Além de atingir a decisão de escalonamento que melhor representa o real poder computacional dos dispositivos na plataforma, também permite ao GAMA atingir mais de 100% de eficiência tal como definida em [3], executando um importante algoritmo científico irregular.

Integrando o modelo de escalonamento desenvolvido, o GAMA superou ainda bibliotecas eficientes para CPU e GPU na execução do SAXPY, um importante algoritmo científico. Foi também provada a escalabilidade do GAMA sob o modelo desenvolvido, que aproveitou da melhor forma os recursos computacionais disponíveis, em testes para um CPU-chip de 4 núcleos e dois GPUs.

Abstract

Current computational platforms have become continuously more and more heterogeneous and parallel over the last years, as a consequence of incorporating accelerators whose architectures are parallel and different from the CPU. As a result, several frameworks were developed to aid to program these platforms mainly targeting better productivity ratios. In this context, GAMA framework is being developed by the research group involved in this work, targeting both regular and irregular algorithms to efficiently run in heterogeneous platforms.

Scheduling is a key issue of GAMA-like frameworks. The state of the art solutions of scheduling on heterogeneous platforms are efficient for regular applications but lack adequate mechanisms for irregular ones. The scheduling of irregular applications is particularly complex due to the unpredictability and the differences on the execution time of their composing computational tasks.

This dissertation work comprises the design and validation of a dynamic scheduler's model and implementation, to simultaneously address regular and irregular algorithms. The devised scheduling mechanism is validated within the GAMA framework, when running relevant scientific algorithms, which include the SAXPY, the Fast Fourier Transform and two n-Body solvers. The proposed mechanism is validated regarding its efficiency in finding good scheduling decisions and the efficiency and scalability of GAMA, when using it.

The results show that the model of the devised dynamic scheduler is capable of working in heterogeneous systems with high efficiency and finding good scheduling decisions in the general tested cases. It achieves not only the scheduling decision that represents the real capacity of the devices in the platform, but also enables GAMA to achieve more than 100% of efficiency as defined in [3], when running a relevant scientific irregular algorithm.

Under the designed scheduling model, GAMA was also able to beat CPU and GPU efficient libraries of SAXPY, an important scientific algorithm. It was also proved GAMA's scalability under the devised dynamic scheduler, which properly leveraged the platform computational resources, in trials with one central quad-core CPU-chip and two GPU accelerators.

Contents

Chapter 1

Introduction

This chapter introduces the dissertation work, contextualizing the state of the art of heterogeneous platforms in high performance computing. It briefly presents the development and importance of both CPU-chips and GPU accelerators, and introduces the scheduling issue in heterogeneous environments. Section 1.3 presents the scientific motivation and the contributions of this thesis, whereas section 1.4 overviews the reminder of the dissertation.

1.1 Context

Heterogeneity is increasingly prevailing across High Performance Computing (HPC) platforms, which now include accelerators as co-processors to complement the general purpose central processing unit (CPU) chip. This success is justified by two main factors: (i) the wider spectrum of computing capabilities on heterogeneous systems, more suitable to applications, sets of computational tasks with different computational requirements and (ii) good watt/performance and dollar/performance ratios.

The ever increasing programming capabilities of application specific co-processors, such as graphic processing unit (GPU) boards, are a major contributor to this heterogeneity. From the mid-nineties to the early XXI century, GPUs drastically moved from special-purpose non-programmable boards to powerful general-purpose programmable devices, simultaneously enabling high levels of performance and power efficiency. This success was so significant that these boards influenced HPC hardware, as nowadays accelerated processing unit (APU) chips can prove.

Heterogeneity's success became particularly noticeable in the TOP500's[1] November 2010 list where the chinese *Milky Way No.1* GPU-based *Tianhe-1A* cluster, was considered the fastest supercomputer in the world. The latest TOP500's list, published in June 2012, includes 55 GPU-based clusters. Graphics boards emerged the general-purpose GPU (GPGPU)

[1] TOP500.org aims to deliver the list of the most powerful supercomputers in the world, twice in a year: both in June and November.

era, which motivated efficient implementations of data-parallel applications both in GPUs [54] and CPU+GPU platforms [72, 70].

Common accelerators include not only GPUs, but also both field programmable gate array (FPGA) devices and digital signal processors (DSP) units. Their architectures are not only different from the typical CPU but from one another. To extract the maximum performance out of these devices, time intensive hand-tuning may be required, along with a thorough knowledge of the their architecture. Additionally, these devices are tendentiously parallel, and only a small community of programmers are familiar with parallel programming.

While it is hard to efficiently exploit accelerators separately or as a CPU co-processor, it is even harder to exploit platforms with several accelerators, especially if they differ from one another. These devices have different computing and programming models, as well as different architectures. This is especially relevant when determining the amount of computational work to assign to each computing unit (CU[2]) and managing data transfers. In this context, frameworks were released to hide the burden of the required expertize to program these platforms.

This is not an isolated phenomenon in the history of computer science; the multi-core technology changed the way code must be written to get full advantage of the available parallelism in the chip, only possible on a multi-threaded fashion. To comply with this requirement, frameworks and libraries such as OpenMP [17], Intel Threading Building Blocks (TBB) [61] and Cilk [59] were released or adapted to relieve the end user from the required expertize to effectively program multi-core chips. By ensuring efficient implementations, these frameworks usually raise the level of both performance and productivity[3].

In the past few years, especially between 2008 and 2009, relevant frameworks were designed to address systems populated by computing units including CPU-chips and accelerators, the so-called **het**erogeneous on-board **plat**form**s** (HetPlats). Focused in data-parallel applications, these frameworks have been designed with high emphasis on their embedded schedulers, considered a key player in effectively exploiting the hardware platforms. These frameworks include StarPU [3], Qilin [47], Harmony [24], Merge [46] and more recently MDR [56]; section 3.2 overviews the key issues on these frameworks.

These frameworks usually represent the computational work by tasks, which are then assigned to the available computational resources. For regular workloads, frameworks that tackle multi-core chips and symmetric multiprocessor (SMP) devices are focused on both

[2]In this dissertation the term computing unit (CU) refers to a chip or an entire board, whereas a processing element (PE) refers to a composing part (e.g. a CPU-core or a GPU Stream Multiprocessor (SM)).

[3]In this context, productivity is defined as the ratio between the achieved performance and the man-hours spent in the implementation, similarly to economics, where it is calculated by the ratio between the developed product and time spent on it production.

fairly and equally divide the computational work by the available resources, whereas frameworks addressing heterogeneous platforms must also consider the performance differences of each computing unit (CU) during the scheduling phase.

Scheduling may be defined as the assigning of computational tasks to CUs and the definition of their execution order [63], with respect to a specific target. These targets usually include to minimize the execution time of one application, also called its time-to-solution (TTS), to maximize the throughput and utilization of the system or to minimize the power consumption levels of the hardware, among others. This thesis work addresses the former.

Scheduling is also a key concept in operating systems, which aims to balance the workload across the available resources. In cluster and grid environments, the job scheduling middleware usually aims to keep resources busy as much as possible [11]. Schedulers embedded either on frameworks or on applications, on the other hand, usually aim to run applications in order to maximize or minimize cost functions, where the execution time is particularly common.

Schedulers can be classified as *static*, *dynamic* or *adaptive*. *Static* schedulers assign the workload to computational devices either at compile or launch-time on a single time. *Dynamic* schedulers start the workload assigning process in run-time, performing several assignments. *Adaptive* schedulers are dynamic, but they may change previously taken assignment decisions, usually for the purpose of correcting load imbalance on the system.

Directed acyclic graphs (DAGs), also called task-graphs, are a common and natural way to represent applications, since they can represent applications with arbitrary task and dependence structures [63]. In a DAG, each node represents a computation and each edge the communication cost between the incident nodes. DAGs have been used to adequately represent regular applications, but they have limitations to model parallelism in irregular algorithms [57].

In its general form, the scheduling is an NP-hard problem [69, 26], i.e., its decision problem is NP-complete and no optimal solutions can be find in polynomial time. Due to this inherent difficulty, scheduler decisions are usually based on heuristics, which produce reasonable or even near-optimal decisions, which can potentially be inaccurate. Schedulers overcome these limitations by usually embedding load-balancing schemes, such as work-stealing and task-donation schemes [10, 66, 25, 68].

Inaccuracy in scheduling decisions may also occur due to other factors. Schedulers must incur in low overheads, otherwise the potential performance gains will be lost. As decisions must be quickly taken, schedulers do not usually base their decisions in complex (and slow) algorithms to schedule. In heterogeneous platforms, schedulers must also consider (i) the different computational capabilities of each CU, thus taking different times to run a task and (ii) the distributed memory paradigm on these platforms, where data-movement is an

expensive operation.

Since the effective mapping between tasks and CUs is crucial to achieve good levels of performance [31], (i) is a major issue on scheduling decisions. To consider the different computational capabilities of each CU, schedulers usually embody (per-task) performance models to estimate the execution time of a pair (*task, device*). These estimations are not only useful to decide which CU to assign a task, but also to estimate the relative performance differences among the available CUs, for each task. This matter is covered in detail in section 3.1.

Performance models are well behaved for regular applications and tasks, lacking accuracy for irregular applications. Current definitions for irregular applications differ from one another, but common definitions state that irregular applications either (i) access pointer-based data structures such as trees and graphs [45] or (ii) have data access patterns not known before execution time [51]; although related, (i) and (ii) are not exactly the same.

1.2 Technological Background

1.2.1 Hardware's Perspective

CPU-chips

In 1965 Gordon Moore predicted that CPU-chips would double their number of transistors roughly every 1.5 years [49]. Although this was predicted to happen for approximately ten years, this rule became valid for half a century and is expected to remain valid at least up to 2015. This has enabled the processor clock frequencies to double every year and a half, which enabled software to run faster from one generation to another, just due to hardware developments.

However, in 2005, the clock frequencies started to stall, due to thermal dissipation, and manufacturers started to increase the number of cores within the CPU-chip, while keeping a steady clock frequency. This fact marked the beginning of the multi-core era and the increasing importance of parallel computing.

CPU-chips are general-purpose computing devices, formed by multiple highly-complex cores, each a processing element with local and fast memories, a small part of the memory hierarchy. High performance cores are highly pipelined with replicated functional units to support instruction level parallelism (ILP). CPU-cores also hide long external RAM latencies through complex multi-level caches, whose access latencies are considerable slower.

Additionally, these devices also have vector and super-scalar capabilities. The first enable vector processing, where the same instruction operate on multiple vectorized data sets, the so-called Single Instruction Multiple Data (SIMD) instructions. The latter allows faster CPU

throughput, by executing more than one instruction in a clock cycle, which are dispatched to the replicated functional units on the processor.

GPU devices as accelerators

GPUs, driven by computer graphics, have evolved from non-programmable specific-purpose devices to programmable general-purpose devices. From 1994 to 2001 these devices progressed from the simplest and specific pixel-drawing functions to implementing the full 3D pipeline: transforms, lighting, rasterization, texturing, depth testing, and display. Due to the industry's demand for flexibility, to implement customized shaders, programmable units were added and increased programmability enabled the GPUs to be adopted by the HPC community in general, the so-called general-purpose GPU (GPGPU) era.

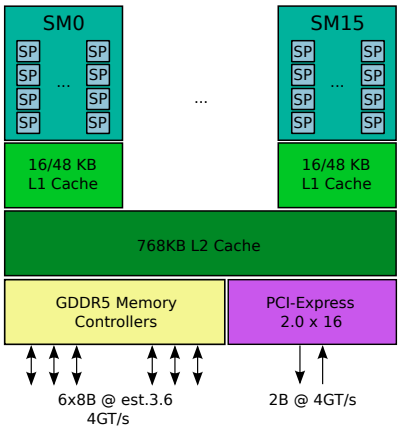


Figure 1: NVIDIA's Fermi architecture.

In particular, NVIDIA, a market leader sided by ATI, released the Fermi architecture in March 2010, formed by up to 512 Stream Processors (SP), clustered in up to 16 Stream Multiprocessors (SM) of 32 elements each. The SMs share a L2 cache, whereas SPs share a scratch-pad one another, within each SM (figure 1). A GDDR5 memory controller is also included in the device, which also supports DMA to system memory, through the PCIe bus. This massive data-parallel device is focused on delivering the higher possible throughput, simultaneously working at relatively low power consumptions.

In general, there is little benefit in running more threads on a CPU-chip than there are physical cores. GPUs, on the other hand, require thousands of threads to achieve several goals, including to hide the latency loads and texture fetches from DRAM. While CPUs rely on the memory hierarchy to hide long external RAM latencies, the GPU relies on switching threads whenever these stall, waiting for a load or texture fetch to complete: that can keep processors busy despite the long memory latency seen by individual threads [55].

Another major feature on GPUs is their ability to efficiently schedule threads by hardware. The Fermi architecture includes a GigaThread global scheduler, and two schedulers per SM. The GigaThread global scheduler distributes thread blocks to SM thread schedulers. Each SM then schedules threads in groups of 32 parallel threads called *warps*. As warps execute independently and no dependencies have to be checked, this is considered a highly efficient process.

Despite their programming and computation model, general purpose GPUs achieved a widely acceptance and success on the HPC field, as the 37 GPU-based clusters in the November 2011 TOP500 list proved. In particular, the chinese *Milky way No.1* GPU-based, Tianhe-1A, became the most powerful cluster in the world, in October 2010, a position kept up to June 2011. Currently, and according to the latest TOP500 list, it is the fifth faster computer in the world, whereas also GPU-based *Jaguar* and *Nebulae* are sixth and tenth, respectively.

Other Computing Units

The set of accelerators in HPC is very likely to grow, as recently Texas Instruments (TI) C66x DSPs series have proved. These microprocessors became suitable to general purpose computing by being endowed with floating point capacity to support the 4G wireless standard [38]. Especially designed for power-efficiency, they can provide more than 500 Giga floating point operations per second (GFLOPS) of performance while consuming 50 Watts of power.

Similarly to TI's DSPs, ARM processors, now leading the mobile industry, are likely candidates to be soon adapted by the HPC community[4]. Also developed to deliver power economy, these chips are now being designed also targeting high performance levels, motivated by the increasing complexity of both mobile applications and operating systems, such as Android or Symbian [40]. Currently limited to a 32-bit architecture, ARMv8 has already announced as a 64-bit ARM architecture.

The 50+ core Many Integrated Core (MIC) is Intel's answer to NVIDIA and AMD's GPU challenge. MIC is being used in experimental trials as a x86 accelerator, connected to the CPU through a PCI bus[5]. In spite of the community's familiarity with the x86 architecture and the possibility of coding MIC using C, C++, Fortran and OpenMP, the facility of MIC's programming and its sustained performance is yet to be proven.

[4] e.g. the ARM-based Montblanc project will replace the MareNostrum in the Barcelona Supercomputing Center (BSC).

[5] MIC is the basis of a 10-petaflop capable supercomputer, the *Stampede*, announced by Texas Advanced Computer Center (TACC) in September 2011. The Stampede's heart is based on 50+ core Knights Corner (KNC) processor chips, packaged in a PCIe form factor.

1.2.2 Software's Perspective

pThreads

To fully exploit multi core features, the application algorithm and code may require a full re-design. Applications instantiated in processes can thus create multiple threads, since a single thread will eventually leave unused resources at some point. A thread is the smallest unit the operating system can schedule and one of the most important concepts in the history of computer science.

POSIX Threads, usually referred to as pThreads, is a POSIX standard for threads. Implementations are available for several Unix-like POSIX-conformant operating systems, such as GNU/Linux. The API provides thread management and synchronization primitives. Although this threading library is still on the top of the preferences of some computer scientists, the industry demanded higher-level libraries to explore parallelism, since with pThreads developers are forced to deal with concurrency issues, such as deadlocks and race-conditions.

OpenMP, TBB & Cilk

OpenMP [17], Intel Threading Building Blocks (TBB) [61] and Cilk [59] have emerged to take advantage of the the multi-core technology from a higher abstraction level than direct threading APIs, as pThreads.

OpenMP is the most popular library across the industry to parallelize shared memory applications. This has been arguably due to its simplicity, since OpenMP is based on preprocessor directives. In OpenMP-based applications, the master thread creates a specified number of slave threads, which concurrently compute a task in a data-structure, while the operating system handles their allocation to the available processors. Task-sharing is done through the work-sharing scheme, which was popularized by OpenMP [34].

Other libraries, such as Cilk and TBB, employ work-stealing as part of their work-balancing mechanism. Work-stealing is described as a technique to implement runtime schedulers. A scheduler employs a specified number of threads, called workers. Each worker then maintains a local queue to store and retrieve tasks. When the local queue becomes empty at some synchronization point, its respective worker will try to steal a task from another busy worker. Each busy worker manages its private queue through synchronization-free pushes and pops.

Intel's TBB [61] is a task-based threading library, which produces scalable parallel programming from standard C++. TBB is completely transversal across platforms and operating systems, only requiring a C++ compiler. The runtime system, automatically handles load balancing and cache optimization, while simultaneously offering parallel constructors and synchronization mechanisms to the programmer. The programmer specifies tasks, so the system

can map such tasks on threads in an efficient manner. Unfortunately to expert programmers, TBB does not provide manual control over task locality.

Cilk/Cilk++ is a runtime system for multi-thread parallel programming in C/C++, designed by the MIT Laboratory for Computer Science and developed by Intel [59]. In this system, each processor maintains a stack of remaining work. When a procedure call is found, the current procedure's state is recorded on the stack as an activation record, and the new procedure is executed. The last activation record is only executed as soon as the remaining procedures have been fully executed. As Cilk employs a work-stealing based scheduler, each stack can be dequeued by other processors that have executed all their work.

Chapel

Chapel is a new parallel, very high-level programming language developed by Cray [16]. It is focused on creating abstractions to data, by separating the algorithmic expression and data structure implementation details. In particular, Chapel creates the concept of "domain", which contain the size and the location of data, used to perform intra and inter-domain operations. Domains and associated operations are then mapped across the available PEs, responsible to perform the algorithmic operations over them.

Message Passing Interface

Message Passing Interface (MPI) is a message-passing API, designed by both academia and industry researchers, to tackle distributed memory, parallel systems. It relies on processes running on each CU, communicating through the point-to-point and group message passing API, either in Fortran 77 or in C.

The parallel, scalable and large-scale applications designed with MPI must explicitly split the data among processes, where each process independently takes its execution. Successive applications have been designed using MPI to leverage several loosely coupled CUs at the same time, while using a shared-memory parallel library as OpenMP or TBB to parallelize each execution flow within each CU.

CUDA

The Compute Unified Device Architecture (CUDA) computing model was launched by NVIDIA, in 2007, aiming to provide an universal environment to address devices with similar architectures to NVIDIA GPU devices. With a specific Instruction Set Architecture (ISA), the CUDA computing model enables users to program the resources (i.e. SPs and SMs) that compose the GPU-board.

CUDA model extends C and in a CUDA application, some parts can run on the CPU, the host, while some functions run on the GPU side, the device. GPU functions, called *ker-*

nels, are executed in parallel within the GPU, by a specified number of CUDA threads. The CUDA model defines a highly-semantical hierarchy of threads. Kernel calls generate a grid of blocks of threads. Both grids and blocks can be conceptually organized in 1-3 dimensions.

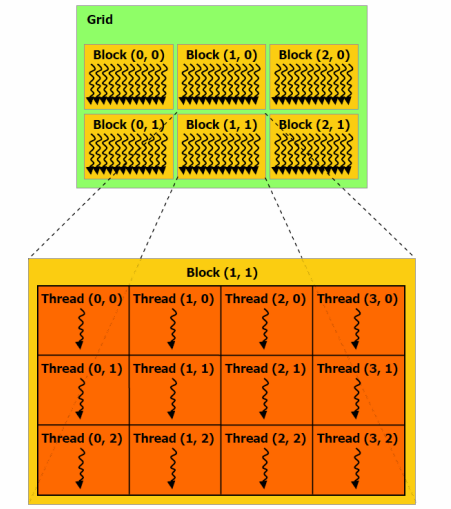


Figure 2: CUDA's hierarchy of threads.

Figure 2 shows a bi-dimensional grid and block thread hierarchy organization. Each block and each thread is associated with bi-dimensional coordinates, which are later associated with specific computation. This hierarchy fashion provides a natural way to connect threads with vectors and matrices. Within the CUDA memory model, each thread has local memory and each block has shared memory, visible by all the threads within it. The complete range of launched threads can view and access both the global and constant memories.

The GPU massively parallel, single-instruction multiple-data/thread (SIMD/SIMT) architecture has achieved impressive performances for regular, data-parallel applications. This has led to GPUs wide acceptance on the HPC community, where a significant amount of problems are data-parallel. Nevertheless, a significant part of (client-side) applications are irregular [57], and some results have been recently presented, reporting good GPU performances in irregular applications [14, 48, 58], possible at cost of extensive hand-tuning.

1.3 Motivation, Goals & Scientific Contribution

Motivation

In the recent past years heterogeneous platforms have prevailed in the HPC field, with high acceptance by the scientific community in general. As much as predictable, computational systems will remain heterogeneous both within each chip [20] and as whole platforms [41]. Moreover, the variety of accelerators on HPC may grow, as the recent Intel's MIC and TI

C66x DSPs boards have proved.

Although heterogeneous computational platforms have been well accepted by the scientific community, due to their high, energy-efficient performance levels, programmers are facing tough times to reach the promised efficiency levels of single accelerators and heterogeneous platforms in general, only possible at the cost of algorithm's re-design and/or dense code tuning, with a thorough knowledge of the underlying architecture [14, 58, 48].

This current programming workflow does not favor high levels of productivity, decreased by the long periods of implementation and tuning. Although some frameworks were released to address heterogeneous platforms, by automatically and efficiently orchestrate data management and scheduling decisions, they lack adequate solutions for irregular applications, which compose a substantial slice of both scientific and technological applications in general [57].

These limitations motivated the design of a new framework to address irregular applications on heterogeneous platforms. The scheduling of both regular and irregular applications on these heterogeneous platforms was also identified as a key issue in such frameworks, because good performance levels are proved to be strictly related with the effective mapping between the application and the available computing resources.

Goals

The main goals of this thesis are twofold: (i) to identify the proper mechanisms to address the key issues that arise on the scheduling of regular and irregular applications on heterogeneous platforms, especially in CPU+GPU setups and (ii) to design, implement and validate a scheduler to effectively employ the mechanisms identified in (i), within the context of the GAMA framework, under development in the research group related with this thesis, and presented in section 2.1. This problem is described in detail in section 2.2.

As a result of the validation of the proposed scheduling mechanism, some other side questions are expected to be answered. In particular, it is expectable this dissertation (i) to characterize the class of algorithms that suits HetPlats whose accelerators are GPUs, (ii) to identify the relation between the accelerator's usage and the application's workload size, and (iii) to identify and characterize some issues with the scalability of applications in HetPlats.

Scientific Contribution

With the thesis under this dissertation, it is expectable to produce and deliver a conceptual scheduling model to simultaneously and effectively address both regular and irregular applications on heterogeneous platforms. The model is expectable to be validated on platforms with central computational units and accelerators, tightly connected one another through high-latency channels, and both regular and irregular algorithms.

1.4 Dissertation Structure

This dissertation contains six chapters, whose summary is presented below:

- *Introduction*: introduced the dissertation work, contextualizing the state of the art of heterogeneous platforms in high performance computing. It briefly presents the development and importance of both CPU-chips and GPU accelerators, and introduces the scheduling issue in heterogeneous environments. Section 1.3 presents the scientific motivation and the contributions of this thesis, whereas section 1.4 overviews the reminder of the dissertation.

- *The problem statement*: briefly presents the GAMA framework, and aims to identify and characterize the problem under study. It also frames the scheduling of irregular algorithms on heterogeneous platforms in a general context, while analyzing the key issues that arise on this type of scheduling.

- *Scheduling on HetPlats: State of The Art*: presents the state of the art of scheduling on heterogeneous platforms. This state of the art is mainly focused on two types of research work: (i) schedulers embedded in frameworks that address heterogeneous platforms (providing a high-level unified API and execution model to the users) and (ii) individual studies on scheduling over heterogeneous platforms. As scheduling decisions are usually supported by a performance model, the initial section addresses these models.

- *An (Ir)regularity-aware Scheduler for HetPlats*: presents the conceptual model and the implementation of the proposed scheduling mechanism. It addresses both regular and irregular applications on heterogeneous platforms, which contain CPU-chips and GPU devices as accelerators.

- *Validation*: validates the proposed scheduling mechanism on the GAMA framework, when scheduling some case study algorithms. It describes the case studies and the target platform, and presents the obtained results and their discussion. The proposed scheduling mechanism is compared with the best possible static scheduling and with commercial libraries that provide the implemented algorithms. GAMA's efficiency is measured, both under dynamic and static scheduling.

- *Conclusions and future work*: concludes the dissertation, presenting an overview of the obtained results, related both with the proposed scheduling model, the performance of GAMA and the experience on heterogeneous platforms. Some guidelines of future work are suggested.

Chapter 2

The Problem Statement

This chapter briefly presents the GAMA framework, and aims to identify and characterize the problem under study. It also frames the scheduling of irregular algorithms on heterogeneous platforms in a general context, while analyzing the key issues that arise on this type of scheduling.

2.1 The GAMA Framework

The *GPU And Multi-core Aware* (GAMA) framework is an ongoing collaboration project between the University of Minho and the University of Texas at Austin. It mainly focuses to bridge the gap between different execution and programming models on a heterogeneous system formed by CPU-chips tightly coupled to accelerators. Theoretically designed to address a wide range of these devices, it currently supports x86 processors and CUDA-enabled devices (with compute capability 2.0 or higher).

Within the GAMA framework, applications are described as set of *jobs*, associated to a computational *kernel*, a data domain - where the computational kernel is applied - and a dicing description, that details how the data domain is diced to create smaller jobs. From the scheduler's point of view, jobs are instantiated as *tasks*, which are assigned to the available CUs on the platform.

Jobs may have cross dependences, which are expressed by the programmer in the form of synchronization barriers, ensuring that no task of a job $k+1$ is executed before of the end of all the tasks that belong to the k job. The run-time scheduler is responsible for ensuring synchronization, although it does not ensures any order of execution in applications without synchronization descriptions.

Memory System

GAMA employs a memory system and an address space on top of the devices memories within the system, which unifies the distributed address spaces and creates a distributed

shared memory (DSM) system. This memory system is based on the release consistency model, which triggers implicit release-and-acquire operations on the data accessed by each thread.

GAMA's memory system provides both private and shared memory, i.e., memory that can be visible and accessible by one or more threads at each time. Data is dynamically allocated and deallocated, following similar strategies to Hoard and xMalloc [8, 37], to favor both speed and scalability.

In the near future, it is planned GAMA to employ a software cache mechanism, enabling each device to maintain a piece of local and fast memory. This is expectable to reduce the memory access latencies for re-used data, which are currently accentuated since accelerators access to memory through high latency PCI-Express channels.

Job Definition and Execution

Each job is described by a class specialization in C++, in which the programmer mainly defines the *execute* and *dice* methods. In the first, the programmer defines the computation kernel executed on the data domain associated with such job. The second defines the data partition methodology to apply when dicing the data domain of that particular job.

2.2 Scheduling (Ir)regular Applications on HetPlats

Current GAMA scheduler assigns the application workload to the available CUs in a static and ungrounded fashion. It is intended to endow the GAMA framework with a scheduler that takes grounded assigning decisions, to minimize the execution time of each regular or irregular application on the running platform.

The scheduling of applications on heterogeneous platforms is still a challenging problem. In particular, the more irregular the application and the more heterogeneous the platform, the wider the spectrum of arising issues. Four types of scheduling can be identified when classifying both the platform and the application respectively as either homogeneous or heterogeneous and as either regular or irregular. These scheduling types are shown in Table 1, whose complexity grows from the top to the bottom and from the left to the right.

(Regular & Homogeneous)	(Regular & Heterogeneous)
(Irregular & Homogeneous)	(Irregular & Heterogeneous)

Table 1: Types of scheduling in the form (application & platform).

The scheduling of regular applications on homogeneous platforms is the most common

and simultaneously the most studied combination. Single core platforms and/or platforms formed by a single CU are usually homogeneous and these remained the most common platforms for several years. While scheduling one application on a single core may be a very low-level process (e.g. instruction level parallelism) and out of the scope of this thesis, the scheduling of regular applications on multi-core CPU-chips is based on assigning (relatively) equal amounts of workload to every core using programming models such as OpenMP and TBB.

By default, OpenMP schedules parallel loops statically where an equal number of iterations is given to each worker thread (e.g. $^{I}/_{T}$ for I iterations and T running threads). OpenMP's dynamic scheduler, on the other hand, is based on the work-sharing scheme to distribute the workload among the available cores. Once a thread finishes a block of loop iterations, it retrieves another block from the top of the work queue. As a result, threads executing in equivalent cores will very likely execute the same amount of workload.

The scheduling of irregular applications on homogeneous platforms has also been previously studied. The majority of this research can be found in implementations of irregular applications on OpenMP-like[1] programming models, targeting (homogeneous) multi-core chips [32, 35, 22]. Although several authors claimed that OpenMP has not the proper features to natively address this problem, some OpenMP extensions were proposed and validated to address this issue (e.g. the ability to cancel threads in a parallel region [65]).

Most of the work in the scheduling of regular applications on heterogeneous applications can be found on the state of the art frameworks designed to address heterogeneous platforms, described in detail in Chapter 3. Some of these proved that both static and dynamic scheduling approaches are able to achieve major levels of performance in some of the best known regular data-parallel algorithms.

The scheduling on these frameworks is usually based on per-task performance models [3], which train the scheduler with information about the execution time of each application's task, either built transparently [2] or through reference runs [47]. This information mainly aims to provide the relative differences of performance of every CU within the system, which are later used by the scheduler to take assigning decisions.

The scheduling of irregular applications on heterogeneous platforms has not been adequately addressed and reported yet. As the complexity in Table 1 grows from the top to the bottom and from the left to the right, this is the most complex type of scheduling to solve. Some of the key issues that arise on this problem and that are intended to be addressed in this thesis are:

[1] According to Tim Mattson, an OpenMP designer, OpenMP was not designed for irregular applications, but several studies achieved high performance levels in irregular algorithms on multi-cores, using OpenMP-like models.

(i) *Differences among CUs.* On heterogeneous platforms there may exist computing units better tailored to perform specific sub-computations on applications, due to multiplicity of architecture features within the system. As a consequence, each CU will very likely perform these sub-computations (e.g. a task) differently, with respect to performance. It is thus essential to identify these *relative differences* among architectures, to achieve good levels of performance. In particular, this is a key problem in the scheduling regular applications on HetPlats, since good performance levels have been achieved by statically distributing the workload with basis on these *relative differences* [47, 52, 31].

(ii) *Irregularity.* Irregular algorithms are composed of tasks whose amount of computation and respective execution time is not known in advance. On data-parallel applications, formed by sets of tasks, this is equivalent to say that the execution times of such tasks might differ, even when executing on the same CU. Thus, the *relative differences* as shown in (i) may not provide enough information to effectively distribute the workload between the available CUs: defining the number of tasks to assign based only on these differences does not necessarily lead to a balanced distribution of computational load. Considering irregularity into a scheduling equation is far from simple, though, since it is very hard (or even impossible) to predict the execution time of irregular applications, which depend on factors such as run-time values and the input set.

(iii) *Load imbalance.* As it is very hard to define a good static policy taking both (i) and (ii) into account, the system may become imbalanced when statically assigning the workload. One approach to correct the workload imbalance is to follow a dynamic scheduling scheme, i.e. to schedule the workload multiple times, which may amortize the imbalance in run-time. This is done by considering the workload assignment to every device at every run-time assigning.

(iv) *Data movement cost.* Accelerators reintroduced the distributed memory paradigm and moving the workload among CUs is an expensive operation when the associated data resides at the local memory of a particular accelerator. This forces to move data between the accelerator and the main memory (or even a second accelerator), incurring in a non-negligible overhead, making work balancing methods too expensive. In GAMA, the task management is done exclusively on the CPU side, but due to data prefetching, each task's data domain is copied to the local memory (if the case) of the device which will execute it. Correcting assignment decisions should thus be avoided and assignment decisions should be made with high certainty degrees.

(v) *Data placement.* As current accelerators have local memories, the data of each task may either be copied to such memories and accessed locally or be remotely accessed, through direct memory access (DMA). As result, assigning a task which accesses data stored in the local memory of an accelerator to another accelerator may incur in two memory copies (back and forth), which may severely affect performance. Thus and in general, an efficient scheduling policy would prefer to assign tasks to the device which already has the respective data.

Summary and conclusions:

After presenting the GAMA framework, this chapter identified and presented the four scheduling types, with respect the algorithm's regularity and the platform's heterogeneity. The scheduling of irregular applications on heterogeneous platforms, also under study in this thesis, is the most complex type, due to the hardware differences and software unpredictability. Several problems that arise from this kind of scheduling were identified and described.

The work in this thesis aims to design, implement and validate a mechanism to effectively schedule both regular and irregular applications on heterogeneous platforms, minimizing their execution time. It is thus expected this mechanism (i) to be aware of the differences of the platform's CUs, (ii) to be aware of the application's irregularity, (iii) to correct potential load imbalance, (iv) to minimize data movements and (v) to take data's placement into account. This mechanism will be validated in the GAMA framework, when running both regular and irregular algorithms.

In Chapter 3 the state of the art of GAMA-like frameworks is described, with particular emphasis on their scheduling mechanisms. Chapter 4 presents the thesis, the model and the implementation of the scheduler mechanism proposed in this dissertation, whereas its validation is shown in Chapter 5, where the scheduler is compared with a baseline static scheduler and its performance on GAMA is compared with commercial libraries.

Chapter 3

Scheduling on HetPlats: State of The Art

This chapter presents the state of the art of scheduling on heterogeneous platforms. This state of the art is mainly focused on two types of research work: (i) schedulers embedded in frameworks that address heterogeneous platforms (providing a high-level unified API and execution model to the users) and (ii) individual studies on scheduling over heterogeneous platforms. As scheduling decisions are usually supported by a performance model, the initial section addresses these models.

As programmers spend considerable amounts of time implementing applications on heterogeneous platforms, frameworks were designed to take this burden, raising programmers productivity levels by lowering the implementation periods. In irregular applications this is even more noticed, because good scheduling and data management decisions depend on run-time values. Without these frameworks, programmers would be forced to design complex run-time systems to effectively take advantage of the platforms.

The embedded scheduler and the data management system (DMS) are crucial players in these frameworks. The former assigns workload to CUs, taking efficient decisions regarding a specific goal, such as performance. The latter moves data among the system memory banks (main and local memories), and manages prefetching and caching mechanisms. While this thesis is focused on scheduling, a parallel research activity focuses on the DMS of the GAMA framework.

Embedded schedulers are composed of several modules, including a load-balancer scheme, which corrects its decisions when inaccurate (e.g. work stealing), a work-management system, which is responsible to maintain the association of computational work and devices (e.g. a queuing system) and an assigner system, which is responsible to take assignment decisions, by mapping workload to CUs or PEs, according to some policy. This module may also include software and/or hardware modeling, usually called a performance model.

Section 3.1 covers the performance modeling theme, by classifying the current types of embedded schedulers based on their performance models. Section 3.2 presents the state of the art frameworks to address heterogeneous platforms, at the perspective of their scheduling and performance modeling systems. Section 3.3 presents schedulers for heterogeneous platforms not embedded in frameworks, which complements the previous section.

3.1 Performance Modeling

To take grounded scheduling decisions schedulers must estimate how tailored each task is to every CU or PE within the system, regarding a specific goal (e.g. performance, the focus on this thesis). This capacity, provided by a performance model, enables the scheduler to efficiently perform the mapping between computational tasks and the available devices [31].

Heterogeneous systems are populated by different devices, with different architectures and computing capacities. They offer a wide spectrum of computational capabilities, more able to comply with the requirements of applications, sets of tasks, potentially with different computational requirements. This is the reason why high levels of performance are strictly dependent on an accurate mapping of computational tasks on devices.

A performance model estimates the "suitability" level of a tuple, typically a $(task, device)$[1] pair. It is usually represented by a mathematical function *f(x)*, where its domain would be the set of tuples *(t,d)* for every task t and device d on the system and its co-domain is typically a "suitability value", either on $\mathbb{N}$ or $\mathbb{R}$. Performance modeling is defined in this thesis as the act of building or using a performance model, whereas its refinement is called calibration.

The goal of performance modeling, also defined as the gain of understanding of a computer system's performance on various applications [5], can be defined as the estimation of the suitability of various applications (or tasks) on a computer system (or device) [64]. Even reversing its definition, it is still true that the resulting model can be used to project performance to other application/system combinations, as defined in [5].

The mathematical function that represents the performance model may be formed by several weighted components. The more components are included in the function, the more accurate the function but the slower the model might be to execute. The range of possible components to include in the function is very wide (the wider the more heterogeneous the platform) and the performance model may be executed several times in run-time, a trade-off between the complexity and the accuracy of the model. As performance prediction has been proven to be so much time-consuming [15, 5, 31], schedulers usually resort to simplified performance models [15, 1].

[1] More formally, performance models usually input the profile of both devices and tasks.

As a consequence of resorting to these less accurate performance models, schedulers may take less accurate, scheduling decisions. The error in each scheduling decision can be quantified as the difference between the optimal scheduling decision and the one taken by the scheduler. This introduced error is usually mitigated and controlled by a load balancing mechanism, which include popular schemes such as work stealing and donation [10, 68]. As these schemes introduce overhead to identify and solve the load imbalance, the error cannot be eliminated, but mitigated.

The more calibrated the performance model, the less the load-balancing scheme needs to act. Although load-balancing schemes can mitigate the scheduling errors by successfully moving tasks between computing units on homogeneous systems, they are highly conditioned on heterogeneous systems. Accelerators re-introduced the distributed memory paradigm, and data-movement might be potentially expensive on these systems, in opposition to shared-memory systems, in which this cost may be negligible. This problem does also affect NUMA-based systems, and studies over OpenMP and TBB frameworks opted by either mitigating or preventing the work-stealing/donation [71, 13, 50].

Schedulers can build (and access) performance models either in compile or run-time (also called as off/on-line). In either case, these performance models may be dynamically calibrated during the application's life-time [2]. Performance models built off-line are designed considering the underlying architectures on the system and the code to execute [31]. Performance models built on-line, on the other hand, are usually based on dynamic learning, i.e., by increasing awareness during the application's lifetime [2, 47, 39, 15]. Performance models may also be parameterized, with information provided by programmers.

Current schedulers may be classified based on their performance models. This thesis classifies as **instrumented** schedulers those that resort to dynamically (on-line) built performance models, because these are built based on run-time sensed values. On the other hand, schedulers that resort to compile-time (off-line) information are classified as **predictors**. These do not need the application to start to estimate the suitability of tasks and devices, which is done by matching the architectural properties of the system with the code to execute. Performance model's calibration is inevitably an on-line process.

Instrumented schedulers resorts to instrumentation to measure the performance of tasks, which is then recorded in the performance model. This is an empirical operation, where the measurement of each task's performance is highly dependent on the occupancy of the platform. Predictor schedulers are based on matching code properties (possibly extracted by a formal analysis of the code) with the hardware ones, usually by an analytical formula. They are usually based on lexical and syntactical analysis of the code, which can be made at compile-time, where compilers are very likely to perform these operations.

An important part of formal performance modeling can be found in performance model-

ing for GPUs, which has been made at the cost of dense analytical functions. One can find several studies in the literature, not only for performance modeling but also for other kinds of modeling for GPUs, as power modeling [53, 36]. Analytical GPU performance models consider a wide set of hardware constraints and features, well modeled in terms of latencies [43, 4, 36].

Presenting estimations with up to 13% of error, the latter work is designed resorting to variables that are gathered by the GPU PTX emulator Ocelot [42]. On the other hand, the authors in [4], have identified the major micro-architecture features for NVIDIA GPUs and built a predictor, which formally analyzes the code and builds a work flow graph (WFG). The combination of the several identified factors, through efficient symbolic evaluation, provide the final approximation to the execution time of the kernel.

3.2 Frameworks to Address Heterogeneous Platforms

An important slice of the state of the art in scheduling on heterogeneous platforms is related to frameworks that emerged on the last years to address these on-board systems. Besides the GAMA framework, current frameworks mainly target regular applications. They have been designed with high emphasis on their embedded schedulers, due to their major role in effectively exploiting the systems. Those frameworks, initially published between 2008 and 2009, include StarPU [3], Qilin [47], Harmony [24], Merge [46] and MDR [56].

3.2.1 StarPU

According to StarPU's authors, approaches to run applications on regular cores with parts offloaded on accelerators are not sufficient to take full advantage of the hardware resources. The real challenge is to dynamically schedule an application over the entire system, across the available PEs [1]. StarPU's scheduler is mainly focused on minimizing the cost of transfers between processing units and on using the data transfer cost prediction to improve the task scheduler decisions.

StarPU's work team published two papers on its scheduler [1, 2], a major player in their framework. The former is focused on the devised performance modeling mechanism, whereas the latter is an extension to efficiently deal with multi-accelerator hardware configurations where data transfers are a key issue.

The authors argued that finding an explicit performance model for a kernel's execution time is a tough task, due to the required extensive knowledge about the kernel and the underlying architectures on the system. As a consequence, it is proposed an empirical, history-based performance model, resulting in an instrumented scheduler.

Based on the Heterogeneous Earliest Finish Time (HEFT) heuristic [67], StarPU's scheduler reported super-linear speedups on a LU decomposition, when supported by the presented history-based scheme. The performance modeling is based on three steps: (i) the measurement of each task's duration, a particular tough task on devices that overlap DMA transfers with computation, (ii) the identification of the tasks, based on the data layout, and (iii) feeding and accessing the model, based on a hash table per architecture.

A relevant part of this model is the "carefully chosen hash-function" that allows to access the performance model on a very efficient fashion. Although transparent to the programmer, this method is not applicable to irregular applications, since slightly different input sizes cannot be predicted based on one another. While introducing the performance model's design, the authors argued that analytical models are very hard to build even on homogeneous modern systems. They also argue that empirical models are most suitable, since they are realistic and can be calibrated, either at runtime using linear regression models or offline for non-linear models.

The latter paper is focused on the StarPU's scheduler, an extension to efficiently deal with multi-accelerator hardware configurations where data transfers are a key issue. It also presents the implementation of data-prefetching on GPUs and a new scheduling policy that reduces the memory occupancy of the memory buses. This paper detailed the scheduler's extension, namely to take data-transfers into account. It also introduced the asynchronous data request management capability, since CUs and PEs should ideally not stall, waiting for input data.

Since StarPU keeps track of each replicated data on the system, the scheduler can conclude whether accessing some data requires a transfer or not. A rough estimation of the data transfer cost is based on both the latency and bandwidth between each pair or nodes. The HEFT policy is thus extended, so that the scheduler can take into account the data-transfer time of each task, along with the estimation of the task's execution time, provided by the performance model. This is reported as significantly important to boost the performance of a stencil kernel and LU decomposition implementations.

However, the StarPU framework addresses only regular applications. Its scheduler and performance model are severely affected by irregular applications, since no assumptions can be made of the size of new input data-sets, even for the same algorithm, and they must be re-written to cover this class of applications.

3.2.2 Qilin

Qilin is described as a (experimental) heterogeneous programming system. The authors described a new type of empirical performance modeling, adaptive mapping, with reported speedups of 9.3x over the best serial implementation, by "judiciously" distributing work over

the CPU and the GPU. This implementation has been reported as 69% and 33% faster than using only the CPU and GPU, respectively, for a set of well-known benchmarks [47].

The adaptive mapping technique is reasoned on the likely unstable optimal mapping regarding different applications, different input problem sizes, and different hardware configurations. The authors have used the well-known matrix-multiplication operation as case study, generating static partitions for three different input data-sets, that ranged from CPU-only to GPU-only in multiples of 10%, as similar studies have also reported (e.g. [31]). Such method allows to validate a good approximation to the task/data partition.

The adaptive mapping is based on both training and reference runs. On the former, the framework splits the application's input size N into N_1 and N_2. Afterwards, these are broken into smaller chunks (of different sizes), which are executed on the CPU and on the GPU. Their execution times are measured and used to feed a curve fitting process to determine the curves of each task, for both the CPU and the GPU. A reference run is then predicted through these curves, stored in a database, and the maximum time of these predictions is the estimated at run-time.

The authors empirically noticed that low computation-to-communication ratio renders the GPU less effective, which implicitly takes the bandwidth to the device into account. This metric, although relevant on these environments, has always been implicitly handled through the ratio between the data-transfer and computation on Qilin and on other similar work [31].

Unlike some similar-purpose research, the Qilin programming system relies on DAGs to express the dependencies of the application and on dynamic compilation, which builds the DAG of the application and decide the (task,device) mapping either using programmers restrictions or using the adaptive mapping technique. Before code generation, where the available data on the device is taken into account, the Qilin system optimizes the built DAG to reduce operation coalescing and unnecessary temporary arrays.

Qilin is not designed to address irregular applications, similarly to the reminder related work. While DAGs are not appropriate to model parallelism in irregular algorithms [57], the adaptive mapping technique is not suitable to these either, since the execution time of irregular applications cannot be statically predicted on a single reference or training run. Moreover, the adaptive mapping technique is neither applicable nor scalable for several and different accelerators.

3.2.3 Harmony

Harmony is focused on providing semantics to simplify the parallelism management, dynamic scheduling of compute intensive kernels to heterogeneous computing resources, and online monitoring-driven performance optimization for heterogeneous many core systems, with high

focus on binary compatibility across the platform. The Harmony's short paper provides little information on its scheduler. It does, however, acknowledge the importance of dynamic mapping and states that *a priori* prediction would improve the quality of the schedule [24].

Harmony's dynamic scheduler is based on mapping kernels to PEs and variables to memory spaces as the program is being executed [23]. The scheduling operation lasts while the window of kernels fetched from the program, continuously updated, is not empty. The scheduler includes a performance model to predict the execution of kernels based on the used variables, the information about its PTX assembly code, and the history of previous execution of the same or similar kernels.

This performance model allows to predict the execution of a kernel on any PE in the system, either as an absolute value or as a confidence interval. The other components of the Harmony runtime may use the performance model as its default behavior or query it again after a higher confidence prediction is obtained. The performance model is built by recording the execution times of kernels along with the machine parameters of the PE it is executed on, the size of the input data set and other values. These parameters are then used to feed a polynomial regression model to create a suitability function.

The kernel resorts to a dependence graph to get the set of fetched kernels that have not been scheduled yet. The scheduler is only responsible to define the PE where a kernel should run, based on the list scheduling algorithm [29], with priority for critical kernels[2]. The scheduling decision may be re-computed in case of misspeculation, which removes kernels from the scheduling list, and load imbalance, where a list becomes empty for a PE whereas others have excess kernels.

Harmony's performance model, whose authors refer to as "performance predictor", is reported to spend >50% of the execution time of the whole Harmony's execution model, whereas the dependence graph takes ∼20%. Other scheduling tasks, such as kernel dispatch take less than 5% of the overall execution time, while the kernel scheduling operation achieves negligible percentages (1%). The Harmony's memory manager consumes ∼20% in this time breakdown. Consequently, its performance model is significantly more time consuming than the other tasks on the system.

3.2.4 Merge

The Merge framework relies on the map and reduce constructs as an efficient set of semantics for describing the potential concurrency in an algorithm. The MapReduce pattern is also responsible to keep the processors balanced with respect to their load, beyond providing transparency in parallelizing the code.

[2]Criticality in this context is defined as the sum of execution times of kernels that directly or indirectly depend on the current kernel.

To tackle portability, this framework uses EXOCHI to create an interface between the code and the accelerators, which makes this framework easily extensible [46]. This approach to load-balance and scheduling is different from the previous one and there is not much information regarding how this framework executes the associated performance modeling.

3.2.5 MDR

The model driven framework (MDR) is designed based on several performance models, which influence run-time decisions, including mapping and scheduling tasks to CUs and copying data between memory spaces [56]. It thus models task execution, while orchestrates the data-movement within the platform. The workloads are represented as parallel-operator directed acyclic graphs (PO-DAGs). The scheduling decisions are based on four identified criteria: suitability, locality, availability and criticality (SLAC).

During the run-time execution the MDR framework exploits coarse-grain parallelism across CUs, and fine-grained parallelism across PEs. These grains of parallelism correspond to inter-node and intra-node on the PO-DAG, respectively. To exploit intra-node, fine-grained parallelism, the MDR framework resorts to TBB and CUDA, for multi-core CPU-chips and GPU boards. Empirical performance models are used to estimate the execution time of each kernel, whereas the communication modeling is based on analytical models.

In their paper, the MDR's authors detailed each criteria in the SLAC set:

- Suitability is considered a first-order effect, and is reported as based on the execution time of each kernel on a CU; a kernel is thus better suited to PE_i than a PE_j if the expected run-time on PE_j is less than the one expected on PE_j.

- Locality is based on the data locality/placement on the system, since each kernel's execution time is not purely determined by its execution time but also the time required to move the associated data between different memory spaces.

- Availability of a PE is based on the estimated time for a PE to become free; as pointed out by the authors and also already considered, the availability allows one to address two scenarios: when a CU is free but a better scheduling decision would be to wait for another specific CU to schedule the task to it, and when it is a better decision to schedule a task for a CU_j even when its suitability is better on a CU_i.

- Critically is based on the impact of the execution of one kernel in the overall execution time application.

Computational tasks on a critical path particularly increase the execution time of one application, when scheduled for a different CU than their preferable one. When using a DAG, the critical path is related with the graph's structure, in addition to computation and

communication costs. It is also argued by MDR's authors that although in some cases critical nodes (kernels) can be statically determined, in other cases it depends on run-time values.

The MDR framework employs a non-blocking pipeline, based on four phases: (i) *sequencing* the tasks ready to execute, (ii) *assigning* those tasks to the proper CUs, (iii) *executing* them on the respective CU, where the memory allocation and data coherence procedures are taken and (iv) *cleaning-up*, where the memory is "aggressively" reclaimed. MDR employs a MSI-based data coherence layer, where data is lazily allocated and eagerly reclaimed. The PO-DAG representation allows MDR to avoid unnecessary allocations.

MDR's scheduler is particularly interesting for including empirical performance models to predict the execution time of each kernel, whereas communication is modeled analytically. Employed history-based performance models, to estimate the execution time of a kernel, are built and calibrated in run-time, similarly to other works [2], in which each computation has an unique signature. The communication model is composed of two parts, namely the *object-level* and the *byte-level* part, which estimates the time to move x contiguous bytes between two memories.

The MDR framework thus identifies four criteria - SLAC - as critical to achieve good performance levels. The framework was tested across 3 different platforms, representative of typical environments of net-books, laptops and servers. The validation was made resorting to selected benchmarks from the Rodinia [18] package, and the employed scheduler was tested against 5 baseline schedulers, including GPU-only and round-robin, concluding that MDR achieved performances at worst 3% and at best 4.2% faster than the best possible baseline.

3.3 Other Studies

Additional studies were published addressing the problem of scheduling in heterogeneous platforms, as an extension to the studies presented above. The majority of them have focused on a particular perspective over the overall problem, including performance/execution-time, device contention and data locality and transfer-time. Thus, this section is structured by these subjects, with particular a focus on performance to match the scope of this thesis.

3.3.1 Execution Time Awareness

The majority of the state of the art schedulers for on-board heterogeneous platforms is focused on minimizing the execution-time of an application, whereas some are focused on other metrics, such as power consumption, data locality and throughput. These schedulers can also be classified either as predictors or instrumented, depending on the use or either formal or empirical performance models.

The first researchers to address the scheduling on heterogeneous platforms have designed

history-based schedulers on a very similar fashion to StarPU's performance modeling and other state of the art frameworks [39]. The authors presented a two-level estimation "estimate-hist" scheduler, which executes a task t on all PE p, afterwards assigning such task to free PEs capable of executing it in less than a θ threshold. If the queues of each PE become full, the scheduler estimates the waiting time of each queue, scheduling t to the less delayed.

According to the introduced terminology, this type of scheduler is instrumented, and it calibrates its accuracy during the application's life-time. The assessment of each queue's delay, schedule each task to the fastest queue, is a well-known process used in the HEFT policy. The transfer time of each task, oppositely to StarPU, is not considered in this scheduling scheme. Although no load-balancing scheme was designed to mitigate the scheduling errors, the authors reported it would increase the effectiveness of the scheduler. Nevertheless, the scheduler achieved speedups up to 40%, using the GPU-only execution as reference.

However pioneer, this work has many aspects to improve. The library is based on a scheduler shared among all the processes (a process per device) which limits the scalability, as recognized by the authors. The two-level estimation algorithm decreases accuracy for large number of tasks, which indicates that the initial estimation is rough and it would be calibrated during the application's life-time, with the execution-time of equal tasks. The suitability of the tasks with the devices on the system is not assessed, which significantly degrades performance, especially in the first-come-first-serve (FCFS) algorithm, due to the assignment of tasks to the CPU.

Similar work presented a fair scheduler extended by a load-balancing scheme, especially aiming to increase the overall throughput [15]. The authors reported the importance of dynamic scheduling, since even GPU-tailored applications would increase the overall throughput of a set of applications if they would be scheduled to the CPU. It is argued that this scheme should be supported by a simple historical run-time information, and the scheme gets better as the database gets built.

Its thesis is based on the importance of knowing which device will allow an application (in this scope as a task) to run faster, especially because previous research claimed speedups up to hundred times using GPUs [62, 19]. Later on, this work describes that the overall throughput can be seriously hurt due to the incapability of GPUs to time slice kernels. Similarly to some of the previous works, this work relied on historical run-time support to tackle this problem. As stated in [1], this work focuses on fast access to the database, which resorts to a hash-map of execution times per application, represented by a minimal data-structure.

Although the authors claimed the scheduler would be able to predict the execution time of an application, this scheduler should be considered instrumented. Its *modus operandi* is based on a database to store and look up (average) execution times of tasks already executed in the system, with the same input size. For different input sizes, the scheduler resorts to

a linear least-squares interpolation, using the new input size and the previous run times as parameters. It is shown in the paper that, for 150 trials of a few benchmarks, predicted execution times fall within the interval of the ones previously measured. The scheduler is reported to introduce 1% of overhead.

The authors reported speedups ranging from 29% to 39%, although used references may be consider poor (GPU-only and a scheduler that assigns each application to its preferred device). This work focused on running several applications in the platform instead of executing one application as efficiently as possible, a perspective considerably close to operating systems. Nevertheless, most of the presented concepts may be used to design a effective scheme for a single application, except the overall system utilization, which may not be imperative.

To build history databases, instrumented schedulers must run every task t in every PE p. However, in applications which are more effective when executed only in the GPU, it is usually true that scheduling even small parts of the workload to the CPU or another accelerator will lead to exponential performance degradation [31]. Some schedulers avoid this by running a test-run (in which results are not committed), but that may not be possible or practical in every real situation.

The second class of schedulers - predictors - may avoid this, by following a formal approach to build their performance model. They are usually designed based on two approaches: (i) by formally analyzing the code, extracting its features and mapping them on the ones from the the system's architecture, by an analytical equation [4, 43] and (ii) by formally analyzing the code, possibly at compile-time, extracting its features and endowing the scheduler with them, which may then be able to take better scheduling decisions and define partitions.

Either way, the analysis of the code is attractive because it is embodied on the compiler, which is very likely to perform the syntactical and lexical analysis of the code. However, this approach has handicaps: (i) it may lack portability, for schemes with a constant model and (ii) the final instructions to execute may be optimizations of the high-level code, thus making high-level analysis potentially inaccurate.

An approach to statically partition data-parallel tasks was recently presented [31], where there is no profiling of the target program and no run-time overhead of dynamic schemes. By extracting code features from OpenCL programs and thus determine the partition phase, this approach relies on machine-learning to build a hierarchy of models [9], instances of support vector machines (SVMs), that maps code features to partitions. This was dealt as a function f that maps a vector of program code features c, i.e. $f(c) = p$ where p is as near as possible the optimal partitioning scheme.

Arguing that off-line profiling is too expensive [47] and noting pitfalls to dynamic techniques [60], this work designed a framework implemented in Clang to build an abstract syntax

tree. The formal analysis is then performed on this tree, thus extracting the code features. Namely, it extracts the number of floating point instructions or the number of memory accesses, due to their impact on the code's performance on the GPU. The values of these features were normalized, enabling the prediction of similar tasks with different input sizes.

This performance modeling approach has two levels of prediction, where the first stage filters tasks that are either CPU or GPU-tailored, since the scheduling of task to the CPU in GPU-only problems (and vice versa) has proved to substantially hurt the overall performance. The second prediction level is also supported by a SVM model, but now combined with a radial basis function, to account for the increased complexity of the problem. Taking a dynamic run-time approach as a baseline, this work reported speedups of 1.57 times.

3.3.2 Device Contention Awareness

Another perspective to design scheduling systems is to focus on contention, i.e., the occupancy of a device. The contention of each device has been measured as the execution delay of its remaining assigned work [67, 1]. It is usually unfeasible to determine the processed ratio of the current kernel in execution, which is also taken into account when possible.

Researchers from the University of Virginia investigated a dynamic scheduling approach for two different devices using OpenCL [12]. Relying on meta-information available at run-time to estimate the most efficient use of heterogeneous resources, this work aimed to study the consequences of the data-locality (analyzed in the next section), the contention of each PE and their strict speed.

The authors reported difficulties to study the impact of the cache occupancy on the overall performance, and found no practical ways to measure the contention of each PE, regarding the kernel in execution at that instant. The Unix *top* command has proved to cause high overhead while GPU had no proper ways to measure its own occupancy. This was usually overcome by designing queue-based schedulers, where each queue's contention is practically measured, resorting to an adequate performance model [3].

3.3.3 Data Awareness

Heterogeneous multi-device platforms are typically distributed memory systems, where data-transfer is not a negligible operation, in contrast with UMA shared-memory systems. Consequently, schedulers must take data into account during scheduling decisions. In particular, the scheduler must consider: the data-placement, i.e., the place where data resides - either in global memory, in a device's local memory or even in a device's cache [12], and the data-transfer cost, for each device on the system [1], two issues related one another.

Data placement and transfer cost is crucial on scheduling over distributed memory systems, for two main reasons. First, current accelerators typically require to keep in local

memory the data associated with the tasks they execute, copied either explicitly or implicitly, as the Unified Virtual Addressing (UVA) does. As these transfers have non-negligible duration, the scheduler must consider the data placement when trying to balance the workload assignment as a function of the execution time [1].

The second reason is strictly related with load balancing mechanisms, such as work stealing and task donation. As schedulers take erroneous decisions, as mentioned before, the system incurs in a potential load imbalance, usually corrected by these load balancing schemes. Based on moving work between devices, these schemes have been effectively implemented in shared memory systems, but its effectiveness may be decreased or even null for distributed memory system. Significant studies on this topic were not found in published literature.

3.4 Overview

In a nutshell, scheduling is a NP-hard problem, affected by the limited amount of time the decisions must be taken, by the heterogeneity of the platform and by the algorithm irregularity. All approaches presented in this chapter follow scheduling solutions based on heuristics and empirical samples, which still provide good scheduling decisions.

The state of the art of scheduling on heterogeneous platforms is relatively short and new, where relevant papers were only published after 2008. As platforms populated by different commodity devices are especially attractive due to their wide spectrum of capabilities - more able to satisfy applications in general - the mapping between tasks and CUs is crucial to achieve good levels of performance. The quality of this mapping is usually improved with performance modeling, which estimates the suitability of computational tasks and CUs.

As these scheduling mechanisms are very likely to repeatedly perform those estimations, they employ relatively simple and fast performance models to reduce performance overheads. As a consequence, scheduling decisions can be inaccurate, requiring the scheduler to embed work-balancing mechanisms: stealing and donation can be identified as the most popular ones. However, as platforms with accelerators follow the distributed memory paradigm, data-movement is now an expensive operation and work-balancing mechanisms should be traded-off.

Performance models are built either off or on-line. Schedulers which employ the former are classified in this dissertation as predictors whereas instrumented ones follow the latter. Predictors rely on analytical functions, possibly fed by formal code analysis. Instrumented approaches, on the other hand, are naturally dynamic and rely on monitoring/sensoring methods to build the associated performance models. State of the art frameworks to address heterogeneous platforms resort to instrumented approaches, arguing that these provide realistic and efficient solutions.

The state of the art of scheduling on heterogeneous platforms lacks adequate solutions for irregular applications and workloads, where most of the efforts are still focused on systems with a general-purpose CPU and a single accelerator, namely a GPU. This is a major topic of interest, since both chips and platforms are becoming more heterogeneous, whereas a significant slice of applications, either scientific or technological, are irregular.

Summary and conclusions:

This chapter introduced the definition and general concepts of a performance model, which estimates the execution time of a task t when assigned to a device d. Two types of schedulers were identified, with bases on their performance model mechanism. The majority of the presented studies reported that high performance levels are directly related with the efficient mapping of tasks on the available CUs, when scheduling applications on HetPlats.

The presented state of the art of scheduling on heterogeneous platforms shown that these are still mainly focused on regular applications. Most reported studies were either related with frameworks to address HetPlats or with sparse studies on scheduling. The presented scheduling schemes are mostly based on performance modeling such as reference runs and historical run-time data, which proved to be efficient for regular algorithms.

Next chapters propose a new approach of scheduling both regular and irregular applications on HetPlats. Chapter 4 presents the roots of the approach, along with its model and its implementation. Chapter 5 presents its validation with the relevant scientific algorithms, which include SAXPY, the Fourier Transform, and two n-Body solvers, chosen due to their variety of characteristics. The SAXPY, the Fourier Transform and the brute force n-Body solver are regular, whereas the intelligent n-Body solver is irregular.

Chapter 4

An (Ir)regularity-aware Scheduler for HetPlats

This chapter presents the conceptual model and the implementation of the proposed scheduling mechanism. It addresses both regular and irregular applications on heterogeneous platforms, which contain CPU-chips and GPU devices as accelerators.

Scheduling mechanisms based on *per-task* performance models or on the processor speeds to run regular applications have proved to be efficient and capable of achieving super-linear efficiency, while keeping the load balanced [3, 52, 47]. However, considering the same scheme for irregular applications might be insufficient, since performance models lack accuracy for irregular algorithms, which may lead to load imbalance.

The efficient scheduling of irregular applications is often only solved with dynamic scheduling, capable of balancing the load on the system during the life-time of the application [6, 44, 27]. Load imbalance is particularly noticeable in this class of applications since good scheduling decisions are hard or even infeasible to find in irregular workloads. Dynamic assignment also enables the dynamic scheduler to balance the load according to the algorithm's behavior.

The proposed scheduling mechanism is thus based (i) on an empirical per-task performance model to effectively schedule regular applications and (ii) on dynamically scheduling the workload, as a response to the ineffectiveness of performance models to schedule irregular applications. The load imbalance on the system, caused by inaccurate scheduling decisions that arise from (i), is corrected by scheduling the workload in chunks, sets of tasks that belong to the same job, whenever the scheduler is signaled by a worker thread.

The proposed model also benefits from dynamic scheduling to correct load imbalance without any relation with irregularity. Because empirical performance models are based on values gathered in run-time and platforms may have indeterminable behavior, these are also potential causes of load imbalance. As Figure 3 shows, the dynamic assignment is done with

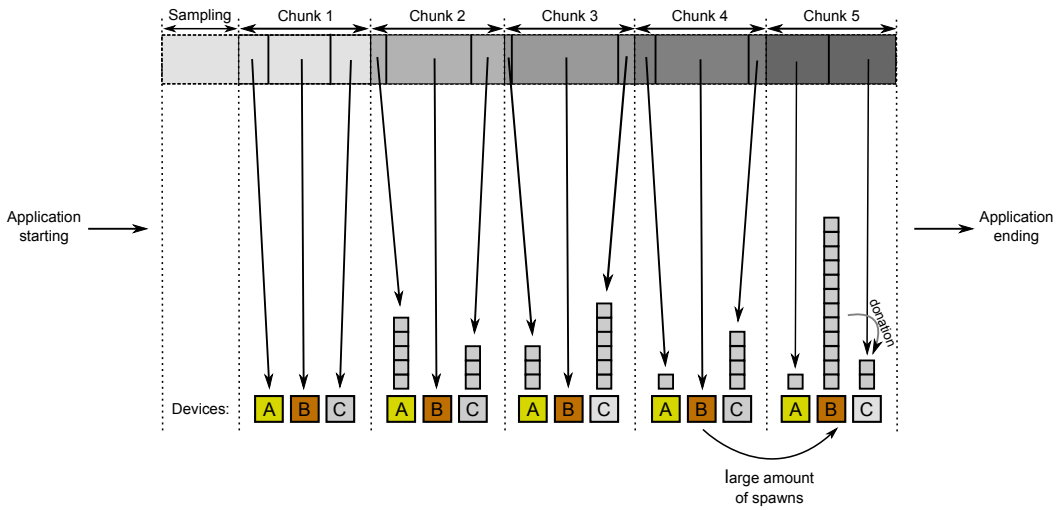


Figure 3: The application's workload is continuously scheduled in chunks, during the execution of the application. The load is assigned to the devices on the system according to (i) their workload at each instant and (ii) their tailorness to the job under assignment.

regard to the load of each device.

Even though dynamic scheduling can maintain the system balanced, it incurs in additional overhead and may hurt spacial locality when compared with static scheduling. In order to mitigate these handicaps, it is required an efficient implementation of the dynamic scheduler and proper data partition. With regard to the former, a solution might be to hide scheduling latencies with computation on the devices, whereas good data partition is associated with efficient dicing schemes. In GAMA, the latter is entirely up to the programmer.

4.1 Conceptual Model

This section presents the conceptual model of the proposed scheduling mechanism, starting by describing the entities within the model, including their job on the run-time system and their interactions with one another. Later on, it is described the performance modeling workflow and the run-time execution model of the proposed scheduler, represented through a Finite State Machine (FSM).

4.1.1 Model's Structure: Entities and their Interaction

The model of the proposed scheduling mechanism is presented in Figure 4. Worth to mention here three key entities: the scheduler, the performance model and the worker threads (workers), which represent the CUs on the platform. One application is composed of one or more jobs, which have a certain number of associated tasks, created by the GAMA's dicing

mechanism.

The scheduler maps the application's tasks into the available workers, according to one scheduling policy and to the estimations given by the performance model. The performance model estimates the execution time of a pair (*task,device*), considering both the hardware features and the execution history. The workers act as proxies of the CUs within the platform, whose job is to execute the computational work.

GAMA represents the computational work by tasks, tuples (k,d) where a kernel k is applied to a data-domain d. These are stored on data-structures that compose a work manager module (e.g. a queuing system), accessible both by the scheduler and by the workers. The scheduler assigns workload to the workers by moving tasks on those data-structures, according to a given scheduling policy.

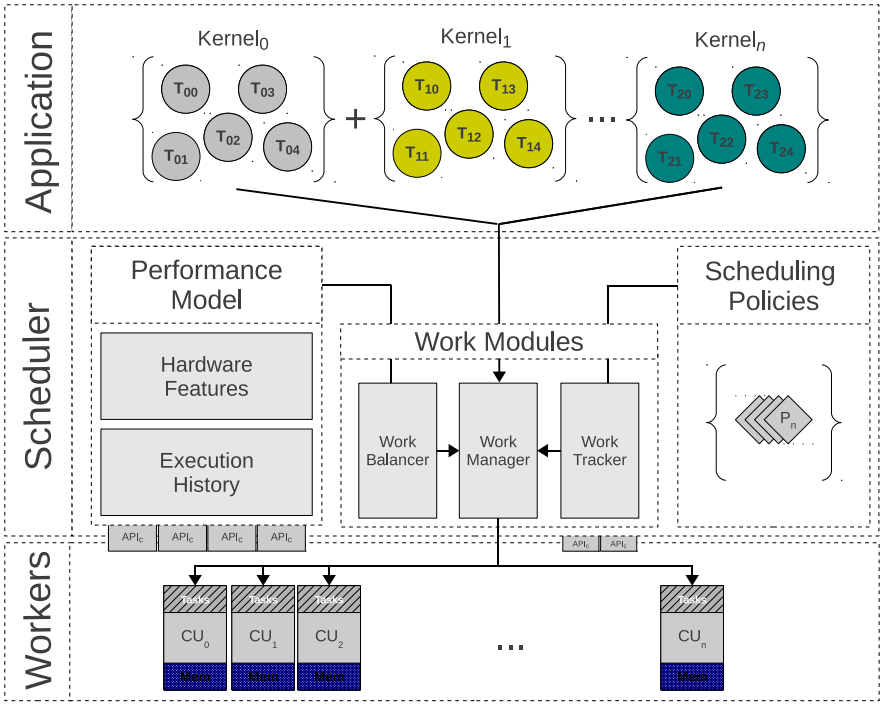


Figure 4: Scheduler's conceptual model and related entities.

The Work Tracker module keeps track of each task on the system, which may be already assigned to a given worker w_i or may still to be assigned. With this module, the scheduler determines how many tasks are assigned to each worker, and at which job it belongs. The Work Balancer module is composed of stealing/donation schemes, which are activated when running the adaptive mode of the scheduler, whose implementation was pushed to future work.

The workers interact with the scheduler and the performance model through specific Application Program Interface (API) calls, both (i) to request additional computational work, on a demand-driven fashion and (ii) to inform that a task of a certain job j has been executed. The workers also communicate with the performance model to supply the execution times of the computed tasks, when explicitly requested by the scheduler.

4.1.2 Assignment Policy

The default assignment policy is based on the execution time estimation of each task both to evaluate the state of the system at each moment and to assign computational work. As a consequence, the scheduler must know the estimated execution time of each task under assignment and the estimated execution time of the workload assigned to each worker, which possibly belongs to different jobs. These estimations are provided by the performance model.

The assignment policy follows a greedy approach, choosing the best possible assigning decision whenever it is executed. At every run, the scheduling decision is calculated so that the remaining tasks on every device take the closest time to execute. This is a variant of the HEFT scheduling algorithm [67], which ranks the tasks based on both their computation and communication costs, assigning every unscheduled task t_i to the processor p_j that minimizes the EFT value of the task t_i.

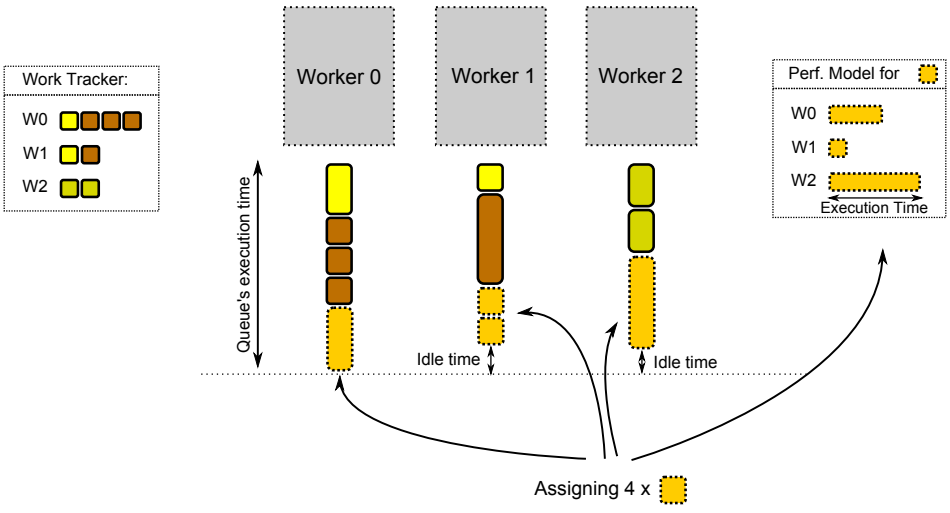


Figure 5: Illustration of the default assigning policy's behavior. Rectangles represent tasks, each color represent a different job (■, ■, ■ and ■ represent tasks from different jobs) and the width of each rectangle represents the estimated execution time of that task.

In contrast to the original HEFT, the proposed scheduling policy dynamically assigns chunks of $\frac{N}{x}$ tasks in every of the x runs, for an application with N tasks. Tasks are scheduled in order to equalize and minimize, as much as possible, the estimated execution time of

the workload assigned to each worker. This method, illustrated in Figure 5, is based on minimizing the idle time of each worker, thus favoring the application's time-to-solution (TTS).

As shown by Figure 5, the scheduler assigns the workload dynamically, in chunks of tasks (4 in this particular case), among the available workers, in order to the execution time of the workload assigned to each one of them to be as closest as possible.

In the context of heterogeneous on-board platforms, the communication cost of one task - referenced in the original HEFT - is mainly related with data transfer latencies. This is especially relevant when tasks whose data is allocated on the local memory of a worker are scheduled to a different worker. Currently, because all memory is allocated on a pinned fashion, this issue is not relevant. However, GAMA will employ a software cache mechanism in the future, and experiments should be made to evaluate the relevance of this problem.

4.1.3 Performance Modeling

The proposed scheduling model is based on an empirical, per-task performance model, created and fed whenever an application is executed on GAMA. The performance modeling process, called *Sampling*, is based on tagging part of the workload as *sampling tasks*, whose execution times will feed the performance model. Before executing a task, the workers verify its label, recording its execution time and communicating it to the performance model, in case of sampling tasks.

The scheduler initiates its execution with an empty performance model, considering all the workers as equally efficient, thus statically assigning them the same number of tasks. The performance model is updated and calibrated in run-time, so the scheduler can progressively refine its decisions. During the execution of the application's first job, the scheduler assumes that the first worker to update the performance model is the most tailored to that task.

The dynamic scheduling process starts after the performance model's first update. Although ideally the scheduler would start assigning the workload after the sampling process becomes complete, this can considerably degrade performance, especially for applications composed of tasks whose execution times differ substantially for different devices. In order to mitigate the performance model's inaccuracy for irregular applications, the scheduler chooses non-consecutive, random tasks to participate in the sampling process.

4.1.4 Run-time Execution Analysis

The proposed scheduling model may run statically (when the number of chunks is set to 1), dynamically and adaptively. While the adaptive mode, which embed work stealing/donation mechanisms, was pushed to future work and the static mode is a particular instance of the dynamic mode, this section analyzes the latter, which is represented by a FSM of 5 states,

as shown in Figure 6.

The scheduler starts in state **P** (Paused), which leaves as soon as it receives a worker's request (of work). The scheduler moves to state **A** (Assigning) to assign a chunk of workload to the workers, according to the selected assigning policy. In case of no work to assign is available, the scheduler returns to state **P**. Otherwise, the scheduler assigns a chunk of work and updates both the Work Manager, in state $\mathbf{U}_{m1}$ (Updating) and the Work Tracker, in state $\mathbf{U}_{m2}$, respectively.

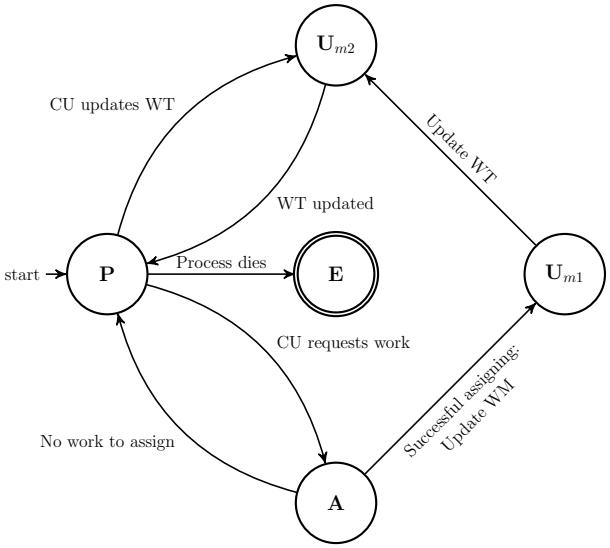


Figure 6: Scheduler's run-time execution FSM. **P** = Paused, **A** = Assigning, $\mathbf{U}_{m1}$ = Updating Module 1 (Work Tracker), $\mathbf{U}_{m2}$ = Updating Module 2 (Work Manager) and **E** = End.

After updating both the Work Manager and the Work Tracker, in states $\mathbf{U}_{m1}$ and $\mathbf{U}_{m2}$ respectively, the scheduler returns to state **P**, where it waits for new requests of work. In this state, the scheduler can also receive a request to update the Work Tracker, which comes from a device that completes a task, then returning to the state **P** again ($\mathbf{P} \rightarrow \mathbf{U}_{m2}$ and $\mathbf{U}_{m2} \rightarrow \mathbf{P}$). This behavior remains cyclic until the application ends. Once the application's workload is complete, the scheduler moves to the final state **E** (End), when GAMA's process dies.

4.2 Implementation

4.2.1 Performance Model

The performance model estimates the execution time of a $(task, device)$ tuple with basis on the median of the five most recent measures. The performance model is calibrated until the application's termination, unless a convergence of $\leq$10% of standard deviation is achieved. The performance model is able to interpolate the execution time of tasks whose size was not

used in the sampling process, through a Spline feature.

The performance modeling base process (Sampling) is based on analyzing a workload sample (20% by default) and on recording the execution times of the associated tasks. The scheduler labels each task as *sampling*, for which the workers record the associated execution time: x86 workers use a function from sys/time.h, whereas the GPU workers determine the end of a kernel with callbacks, a feature available with CUDA 5.0.

The performance model is implemented as a 2-level chained hash-table to mitigate accessing overheads, as shown in Figure 7. The first hash-table stores the task classes (e.g. "saxpy"), each of which containing a pointer to a second hash-table, which stores the sizes of the data domains used in the sampling process, for that particular class of task. The scheduler creates a performance model for each worker within the system, to avoid concurrency and performance degradation.

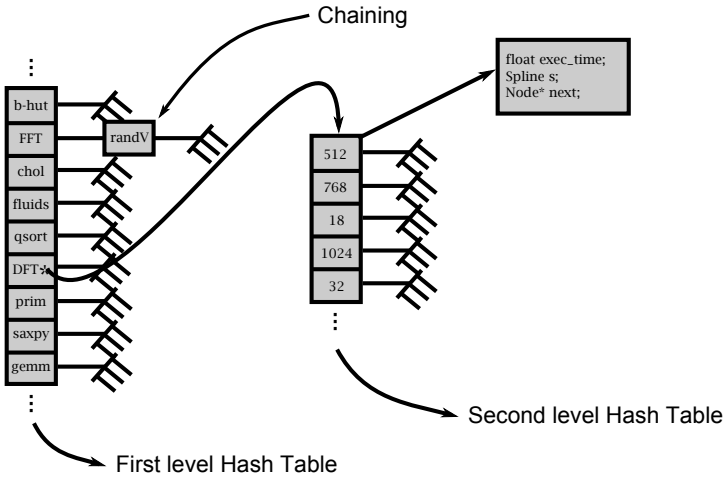


Figure 7: Performance model's implementation: a dual-level hash table.

4.2.2 The Scheduler

The proposed scheduling mechanism was implemented on the GAMA framework, whose Work Manager module is based on a hierarchic queuing system. This module was adapted to fit on the dynamic scheduling mechanism, employing two types of queues: a *higher level queue* (HLQ) and n *local queues* (LQ)s, each accessible by one of the n workers on the system, as Figure 8 shows.

In GAMA, each job is diced according to the programmer's specification. The resultant tasks are stored in the HLQ, from where the scheduler retrieves tasks to place on the n LQs, according to the selected policy. Each worker then executes the available work by retrieving tasks from its own LQ and instantiating them according to its own computing model. If the

size of the LQ becomes lower than a fixed parameter, the worker awakes the scheduler, which assigns a new chunk of tasks[1] and sleeps again.

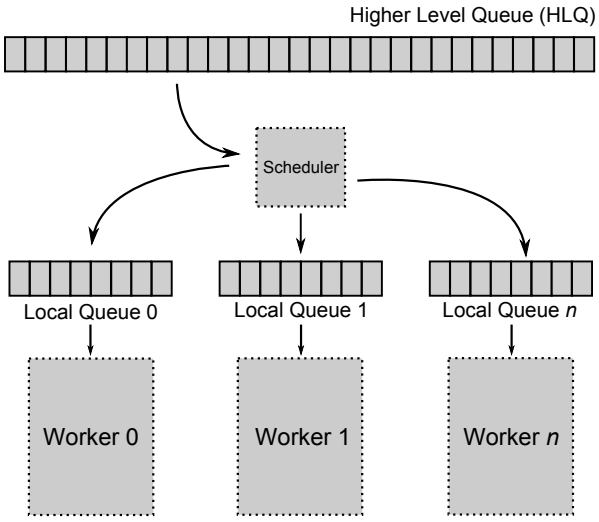


Figure 8: Work Manager module, implemented as a hierarchical queuing system.

The scheduler is implemented on a thread that sleeps after assigning a chunk of tasks, whenever requested. In order to apply the scheduling policy, the scheduler accesses the *Work Tracker* and the Performance Model to estimate the execution time of the load associated with each device on the system. The *Work Tracker* is implemented as a table, similar to Table 2, filled in this case according the workload distribution in Figure 5.

The *Work Tracker* maintains the number of both assigned and executed tasks for all jobs in all the workers, enabling the scheduler to determine the number of tasks (and of what job) that a worker has on its LQ, along with their execution time. This avoids the scheduler to analyze the local queues of every worker, a concurrent operation that may lead to unexpected behavior.

With the default assignment policy, the scheduler assigns a chunk by determining the number of tasks to assign to each worker with basis on the result of a linear equation system. It is formulated according to the execution time estimation of both each LQ and the execution time estimation of the task under assignment, on each worker.

The estimated execution times of both the task t under assignment and each LQ are given by the performance model. For an application with m jobs, the execution time estimation of

[1] According to the default scheduling policy, the worker that awakes the scheduler is not relevant, since the policy evaluates all the workers as candidates of receiving work. The worker to awake the scheduler may, in general, receive more tasks than the others, because its queue is generally smaller than the other ones.

Worker 1	Job 1		Job 2		...	Job n-1		Job n	
	4	3	18	15		0	0	1	0
Worker 2	Job 1		Job 2		...	Job n-1		Job n	
	8	7	15	14		0	0	2	0
Worker n	Job 1		Job 2		...	Job n-1		Job n	
	4	4	0	0		2	0	1	0

Table 2: Work Tracker module, implemented as a table that stores the number of assigned and executed tasks, respectively, for each worker and all the application jobs.

each worker's LQ, referred to as TLQ_w, considers the number or remaining tasks of each job on that queue, multiplied by the estimation of their execution time on that particular worker w, as equation 4.1 shows:

$$TLQ_w = \sum_{j=1}^{m} \Lambda_{wt}(j, w) \times \epsilon_{pm}(j, w) \tag{4.1}$$

where $\Lambda_{wt}(j, w)$ represents the amount of tasks, according to the Work Tracker, associated with job j, on the local queue of the worker w and $\epsilon_{pm}(j, w)$ represents the performance model's estimation to the execution time of tasks belonging to job j on the worker w (for simplicity purpose, the size of each task is not considered in this formulation). For instance, assuming the Work Tracker data as in Table 2, the estimation of the execution time for the local queue of the first worker is given by:

$$(4 - 3) \times \epsilon_{pm}(j_1, w_1) + (18 - 15) \times \epsilon_{pm}(j_2, w_1) + (1 - 0) \times \epsilon_{pm}(j_n, w_1).$$

The number of tasks to assign to each worker at each moment, when scheduling a chunk of T tasks belonging to job j, is computed through a system of n linear equations and n unknowns, n being the number of workers. It is based on balancing and minimizing the execution time of all the LQs in the system. The equation system is given by:

$$\begin{cases} TLQ_0 + t_0 \times \epsilon_{pm}(j, 0) = TLQ_1 + t_1 \times \epsilon_{pm}(j, 1) \\ TLQ_1 + t_1 \times \epsilon_{pm}(j, 1) = TLQ_2 + t_2 \times \epsilon_{pm}(j, 2) \\ ... \\ TLQ_{n-1} + t_{n-1} \times \epsilon_{pm}(j, n\text{-}1) = TLQ_n + t_n \times \epsilon_{pm}(j, n) \\ t_0 + t_1 + t_2 + ... + t_n = T \end{cases} \tag{4.2}$$

where t_w represents the number of tasks to assign to the worker w. This system is always possible and determined. Its solution may include negative values, which are set to 0, and positive values are proportionally adjusted to the number of tasks to assign, T. Negative

values represent, in the context of this problem, workers whose load is excessive when compared with the other ones. Expect for the last chunk, the scheduling model relies on the next chunk to correct such potential load imbalance. In future work, stealing schemes might be implemented as part of the adaptive mode of the proposed model, which can relieve some load of excessively loaded workers, at the scheduling of every chunk. This capability will be especially relevant in the last chunk of work and in dynamic task spawning, a feature that GAMA might include in the near future.

The system resorts to the Lapack++[2] (lapackpp) library to solve the linear equation system. The scheduler converts the equation system into a matrix, used to feed the method LaLinearSolve. As the problem is formulated by a linear equation system, the solution can contain negative values, which is not suitable for this particular problem. In order to solve this issue, an optimization problem with variable constrains could be described instead, but results have shown that solution methods (e.g. simplex) incur in excessive overhead.

Summary and conclusions:

Performance models were successful in achieving major levels of performance when scheduling regular applications on heterogeneous platforms. The scheduling of irregular applications, on the other hand, was proved to be efficient almost only when employing dynamic scheduling, capable of balancing the workload in run-time, which becomes imbalanced since little or no assumptions can be made about the application's behavior.

It is thus proposed a novel and greedy approach, inspired on the HEFT policy, that schedules both regular and irregular applications on heterogeneous platforms, both considering performance modeling and dynamic scheduling. In particular, the proposed scheduling mechanism balances the workload in run-time, considering execution time estimations for both the tasks under assignment and the workload which have been already assigned to each worker and is still remain on their queues.

This chapter presented the thesis under this novel scheduling mechanism, along with its detailed conceptual model and the implementation of its main components, which include a performance model, a scheduling thread and a Work Tracker module. Its default scheduling policy was also presented, and its dynamic mode was analyzed, in the form of a finite state machine.

[2] http://lapackpp.sourceforge.net/

Chapter 5

Validation

This chapter validates the proposed scheduling mechanism on the GAMA framework, when scheduling some case study algorithms. It describes the case studies and the target platform, and presents the obtained results and their discussion. The proposed scheduling mechanism is compared with the best possible static scheduling and with commercial libraries that provide the implemented algorithms. GAMA's efficiency is measured, both under dynamic and static scheduling.

5.1 Case Studies

This chapter presents four case studies, to support the evaluation runs with which the proposed scheduling mechanism is tested. These include the SAXPY, a memory bound routine of linear algebra, the Fourier transform, based on a CPU-tailored implementation of the Cooley-Tukey algorithm, a naive brute force n-Body solver, and the Barnes Hut algorithm. The Barnes Hut algorithm is irregular, whereas the others are regular.

5.1.1 SAXPY

The SAXPY operation is a first level (vector-vector) routine from the Basic Linear Algebra Subprograms (BLAS) package, that computes $z_i = \alpha \times x_i + y_i$. The element z_i of the final vector is the sum of an element y_i with the scalar multiplication of a scalar α with an element x_i, where Y, X and Z are vectors, as shown in Figure 9.

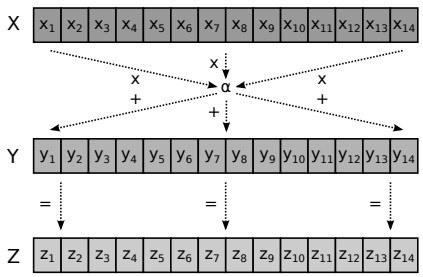


Figure 9: The SAXPY operation.

The SAXPY operation is based on a single precision scalar α and two single precision vectors X and Y, whereas DAXPY works with double precision elements. The remaining related operations, CAXPY and ZAXPY, work with complex and double precision complex elements, respectively.

5.1.2 1D Fast Fourier Transform

The Fast Fourier Transform (FFT) is an efficient algorithm to compute the Discrete Fourier Transform (DFT). The DFT is a discrete transform used in Fourier analysis, which computes a function *f'*, of an original function *f*. In the decimation in time (DIT) Fourier Transform, the original function f is called the time domain representation, whereas f' function is a function in the frequency domain. While the original and naive DFT has quadratic complexity, i.e. $\mathcal{O}(n^2)$, the FFT has $\mathcal{O}(n\ log\ n)$ asymptotic complexity.

A common FFT implementation is based on the Cooley-Tukey algorithm. Although its roots date from 1805, when Friedrich Gauss devised the algorithm, it only became widespread in 1965, due to a publication of James Cooley and John Tukey [21]. It was later re-"discovered" by Heideman and Burrus in 1984 [33]. This divide-and-conquer algorithm recursively breaks a DFT of size N into two $N/2$ DFTs. Radix-2 versions work on power of two element sets, whereas Radix-4 and 8 variants work with powers of 4 and 8, respectively.

The Cooley-Tukey algorithm can be implemented in three steps[1]: (i) the bit-reversal operation over the input data array, (ii) the calculation of the nth unit roots, traditionally referred in the literature as twiddle factors [28], and (iii) a set of butterfly operations, shown in Figure 12. While phases (i) and (ii) can be computed in parallel, phase (iii) requires both (i) and (ii) to be finished. The execution time of phase (ii) is usually discarded in measurements, since it can be amortized when computing several FFTs, of the same or lower size.

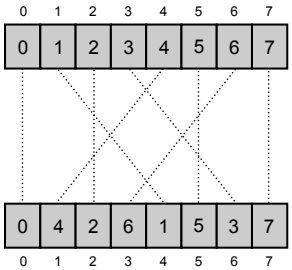


Figure 10: Bit reversal operation for $n = 8$ and $m = 3$.

The bit-reversal operation is based on permuting an array with $n = 2^m$ elements, by reversing the m binary digits of each index in the array, where n is a power of two. Figure 10

[1]A similar implementation was proposed by Weng et al., on a paper entitled "Implementing FFT using SPMD style of OpenMP", published in the 6th Int. Conf. on Networked Computing and Advanced Information Management, 2010, not cited in this thesis due to the several errors in the available print.

shows the result of the bit-reversal operation for $n = 8$ and $m = 3$. Twiddle factors are stored in an array that contains 2^{k-1} unit roots for each stage k, properly stored to favor efficient accesses, as shown in Figure 11. Each complex root is formed by a real and an imaginary parts, respectively calculated by $\cos(\frac{-2\times\pi\times i}{2^k})$ and $\sin(\frac{-2\times\pi\times i}{2^k})$.

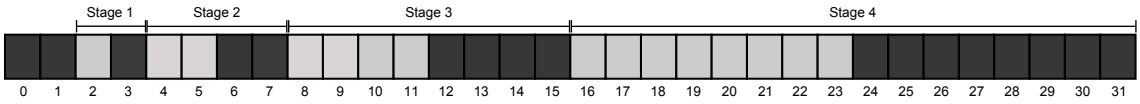


Figure 11: Twiddle factors storage: calculated roots are stored in light-gray indexes, whereas dark-gray positions represent empty positions.

Phase (iii) is computed in several stages, whose number depends on the input size. For the common breadth-first implementation, stage $i + 1$ cannot be computed before stage i is complete, although each stage can be computed in parallel. The data input set is divided in k private domains, distributed among the available k workers. These access data according to different patterns, as shown in Figure 13.

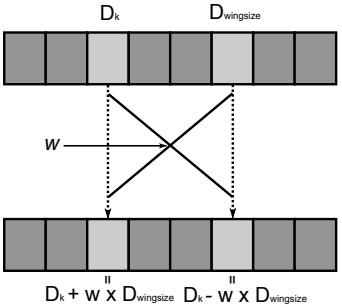


Figure 12: The butterfly operation.

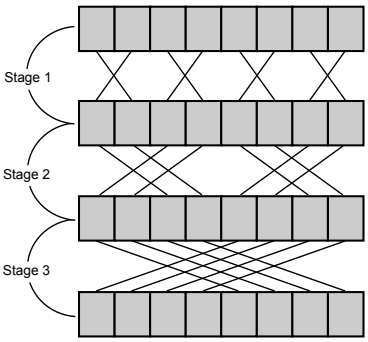


Figure 13: Data access patterns during the computation of the butterfly operations.

The devised FFT implementation in GAMA is composed of four different jobs: the bit-reversal operation and three jobs for the butterfly operations, each one with a specific data access pattern. The implementation has synchronization barriers in two points: (i) between the bit-reversal operation and the butterfly operations, and (ii) between each stage of butterfly operations. The calculation of the twiddle factors was implemented in OpenMP, since their calculation is not considered in the evaluation runs.

Except for the last stage, the implementation is based on computing $N/2$ complete butterflies per stage. In the last stage, on the other hand, the implementation is based on computing $N/2$ half butterflies twice. This procedure favors spacial locality, avoiding to access distant positions in every butterfly calculation. Since this particular scheme arises

concurrency issues, the implementation uses an auxiliary array, an out-place operation.

5.1.3 n-Body Solvers

The n-Body problem is a simulation of a dynamic system of particles. In these systems, particles exercise forces one another, changing their positions in the space, which are continually calculated until the simulation stops (e.g. a convergence is achieved). These systems are useful to study the evolution of star clusters. Several algorithms were proposed to solve this problem, whose asymptotic complexity may go up to $\mathcal{O}(n^2)$ (on brute force solvers).

A n-Body Brute Force Solver

The first devised n-Body solver computes the net forces exercised on particles in a brute force fashion. Its implementation in GAMA is based on a job to compute the exercised forces and on another job to update the position of each particle in the system. These are synchronized and continuously submitted until the simulation stops, upon an user's request. The application shows the system simulation through a simple OpenGL interface.

The Barnes-Hut Algorithm

Some n-Body solvers run with lower asymptotic complexity than the brute force solvers. In particular, the Barnes-Hut algorithm is able to perform the n-Body simulation with $\mathcal{O}(n\ log\ n)$ asymptotic complexity [7]. The algorithm divides the volume into cells in an quadtree for 2D systems and in a octree for 3D systems. This scheme enables the algorithm to treat particles from nearby cells individually and far away ones through their center of mass.

The center of mass is calculated for each cell generated by the tree. For each particle in the system, the algorithm traverses the tree verifying the distance to the center of mass of each cell. For close centers of mass, all the child cells are recursively visited and the particles inside are considered in the calculation of net forces. For far away ones, on the other hand, the calculation of the net force is approximate, by considering the cell's center of mass instead.

The algorithm runs either for a given number of iterations or until a convergence is reached. At each iteration, the algorithm is decomposed in five steps: (i) the computation of the bounding box for all the particles, (ii) the build of the octree, (iii) the calculation of the center of mass for each cell, (iv) the sort of the octree, (v) the calculation of the net forces that each particle exercises on the remaining ones and (vi) the update of the particle positions.

The devised Barnes-Hut implementation in GAMA is based on a highly optimized algorithm to work in GPUs [14]. It performs the most time-consuming part of the algorithm, step (v), whereas the remaining steps are implemented in OpenMP. As a result, in GAMA the devised implementation has a single job. In benchmarks, the application was programmed to run sixty iterations, and the reported time is the median of all sixty iterations.

5.2 Experimental Environment

The evaluation runs were performed on a computational heterogeneous platform whose devices, a CPU-chip and two GPU boards, are specified in Table 3. It runs Linux Ubuntu 11.10 (Kernel 3.0) and it works with CUDA 5.0 (developer release). The code was compiled with GCC 4.6 and NVCC 5.0, with -O2 optimizations in both compilers. Several libraries were used during the runs, including FFTW 3.3.2, Lapack++ 2.5.2, cuFFT and cuBLAS.

Device type	**CPU-chip**	**GPU board**
Number	1	2
Manufacturer	Intel	NVIDIA
Code	Core i7-960	GTX 580
Code Name	Bloomfield	GF110
Year	2009	2010
Cores	4	16 MT-SIMD
Core frequency	3.20 GHz	772 MHz
SMT	2x	48x
Vector Support	SSE 4.2	-
Compute Capability	-	2.0
L1 Cache	32KB iC + 32KB dC	64KB per SM
L2 Cache	256KB per core	768KB
L3 Cache	8MB, Shared	-
SP Peak Performance	102 GFLOPS	1581 GFLOPS
TDP	130 Watt	244 Watt
Main Memory	8GB	1.5GB

Table 3: Target hardware platform.

The evaluated runs addressed four main issues:

- The performance of the devised dynamic scheduler, and how it scored when compared to the best static scheduling solution. To evaluate this issue, the workload is statically scheduled among the CPU-chip and one GPU, in all possible variations on multiples of 10%/25%. The best achieved performance is then compared with the performance of the devised scheduler. This methodology was also the base of similar studies, to evaluate the performance of their devised schedulers [47, 31].
- The execution time and the distance to the platform's Theoretical Peak Performance (TPP), achieved by GAMA when running the case studies. Both the best static and dynamic schedulers were tested on GAMA, which was compared with efficient CPU/GPU commercial and open-source libraries.
- The scalability of the GAMA framework, running both under the devised best static and dynamic schedulers.

- The efficiency η of the GAMA framework, when running both the best static scheduling solution and the devised dynamic scheduler. The efficiency η of a GAMA-like framework expresses how well the framework takes advantage of the multiplicity of architectures on the platform, based on the computational power Ψ of the platform and each device individually. The efficiency formula, shown in equation 5.1, was defined as a way of evaluating frameworks to work on HetPlats [3].

$$\eta = \frac{\Psi_{HetPlat}}{\Psi_{D_0} + \Psi_{D_1} + ... + \Psi_{D_n}} \tag{5.1}$$

where $\Psi_{HetPlat}$ represents the computational power associated associated to the whole platform and Ψ_{D_i} represents the computational power associated to the device i, when the framework forces the whole workload to run exclusively on that device. With regard to a particular algorithm, the computational power of a device i (or a platform p) is given by the ratio between its input size and the execution time that device D_i delivers, as shown in equation 5.2.

$$\Psi_{D_i} = \frac{input\ size}{execution\ time} \tag{5.2}$$

All trials were executed and measured 25 times. This data was filtered by the k-best algorithm, for $k = 3$. The results are expected to enable one reasoning about the performance of the proposed scheduling model, along both with the efficiency η and the scalability of the GAMA framework, when supported by such model. Although productivity is a relevant issue on the context of the GAMA framework, such topic falls beyond the scope of this thesis.

5.3 Results

The first set of trials, shown in Section 5.3.1, aimed at testing the performance of the proposed dynamic scheduler. The experiments consisted in running the complete set of case studies on the target platform, making use of one CPU-chip and one GPU accelerator. The workload was statically assigned in multiples of 10%/25% as a way of determining the best interval of workload distributions. The performance of the dynamic scheduler was then framed with the latter, which allowed to reason about its effectiveness in distributing the workload.

5.3.1 Dynamic Scheduler's Performance

Workload distribution's effectiveness

The first set of results are related to the SAXPY case study, running in GAMA both under dynamic and static scheduling, as shown in Figure 14. By statically scheduling the workload between the CPU and the GPU, in multiples of 10%, it is possible to verify that the best workload distribution lies between assigning 60%+40% and 50%+50% of the workload to the

CPU and the GPU, respectively. The decision found by the dynamic scheduler lied in this band, and delivered the best performance among the entire set of trials.

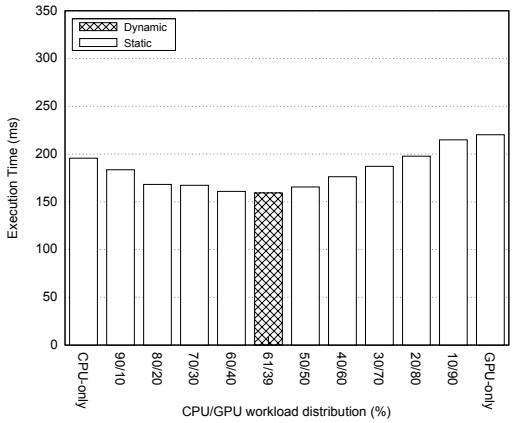


Figure 14: Dynamic and static workload distribution of the SAXPY algorithm in GAMA, for 2^{27} elements in each vector.

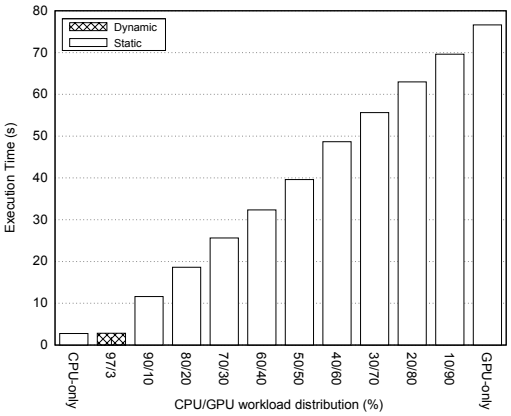


Figure 15: Dynamic and static workload distribution of the FFT algorithm in GAMA, for 2^{25} double precision elements.

Figure 15 shows the workload distribution for the FFT, according to the same methodology used for SAXPY. Scheduling the entire workload to the CPU has proved to be the most efficient solution, since the devised FFT implementation is particularly suited for the CPU. The dynamic scheduler delivered slightly worst levels of performance, since small parts of the workload were scheduled to the GPU, due to the sampling process.

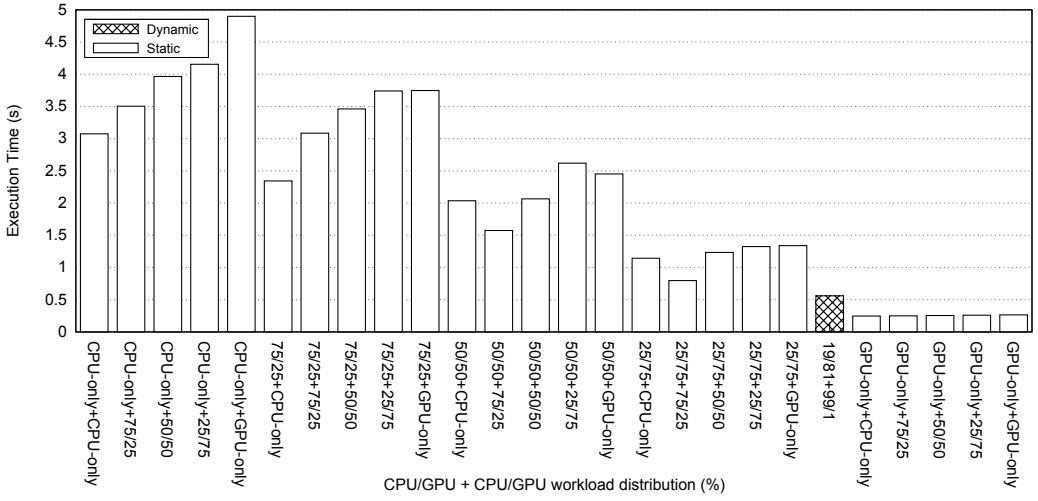


Figure 16: Dynamic and static workload distribution of the brute force n-Body implementation, for a system with 2^{15} particles.

For applications implemented in a single GAMA job, such as SAXPY, scheduling their

workload statically, in multiples of 10%, is enough to determine the interval with the best workload distributions. This methodology works fine for the FFT as well, since all of its jobs are tailored to the CPU. However, for applications whose jobs are tailored to different devices, the best scheduling interval is only found when statically scheduling each job in multiples, for all the possible permutations, as Figure 16 shows for the n-Body brute force solver.

The n-Body brute force solver application was statically scheduled in multiples of 25%, for its both composing jobs. The dynamic scheduler assigned ≈19% of the workload to the CPU and ≈81% to the GPU, for the first job. With regard to the second job, the CPU executed ≈99% of the workload, whereas the GPU executed ≈1%. This decision is relatively coherent with the suitability between the jobs and the available CUs, since the force calculation job is bounded by computation, and thus more suitable to the GPU, whereas the update job is memory-bound, and thus more suitable to the CPU.

Even though the delivered scheduling decision follows the suitability between the application's jobs and the available CUs, its performance is substantially lower than the best static decisions (e.g. CPU-only + GPU-only). The major cause of this performance degradation lies in scheduling ≈19% of the first job's workload to the CPU, when the best class of scheduling decisions consists in scheduling the entire first job's workload to the GPU.

Regular algorithms, such as the devised n-Body brute force solver, are expected to be efficiently scheduled by the proposed model, which should not fail at finding the best (class of) workload distribution(s). The inaccuracy of the scheduler for this particular case study might be likely related with implementation details, which may be adulterating the model. Poor data locality and excessive overhead in calculating scheduling decisions appeared to introduce negligible overhead, according to the results of the rest of the case studies.

Synchronization is, however, a potential cause of additional overhead. The n-Body brute force solver requires a synchronization point between every pair of jobs, and synchronization primitives are specially expensive in dynamic scheduling. Under static scheduling, the amount of workload to execute by each device is known from the beginning of the application, whereas the dynamic scheduler computes the assignment solution only between every pair of synchronization barriers, which may cause the workers to idle in the meantime.

Figure 17 shows the static and dynamic workload distributions for the net force calculation of the Barnes Hut implementation in GAMA, according to the same methodology used for SAXPY and for the FFT. The remaining kernels of the algorithm were calculated in OpenMP, which was not considered in measurements. These trials are based on running a system with 2^{15} particles.

The devised dynamic scheduler delivered the highest performance levels when compared to all the possible static workload distributions, which suggests that the proposed scheduling

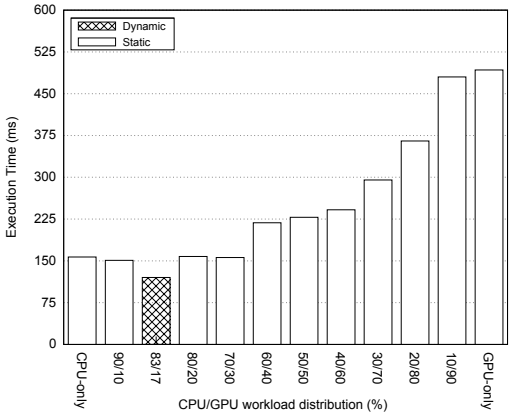


Figure 17: Execution time of both dynamic and static scheduling of the Barnes Hut implementation in GAMA, for a system composed of 2^{15} particles.

model is able to effectively work with irregular applications on HetPlats. The dynamic scheduler identified the correct impact levels of both the CPU and the GPU for this algorithm, by assigning ≈83% of the workload to the CPU and ≈17% to the GPU.

Correlation between the workload size and the GPU usage

Memory transfers are another relevant issue in (dynamic) scheduling, especially in HetPlats, were accelerators may incur in substantial memory access latencies [30]. This is also relevant in GAMA, which currently allocates pinned memory, accessed by accelerators through PCI-express channels. As a result, the accelerators usage depends on the application's workload size, which may or may not be enough to hide these latencies. Next trials aimed at testing the efficiency of the dynamic scheduler in considering memory transfer latencies on the usage of the connected accelerator.

Figures 18 and 20 show that the GPU's usage in both SAXPY and n-Body brute force solver algorithms grows with the input set. These results confirm that HetPlats incorporating GPUs as accelerators are only efficient for algorithms with high computation/data-accesses ratios and large amounts of workload, in such a way that GPU memory transfer latencies are hidden with computation. As much as expectable, the GPU usage will continue to grow for larger input sets, whose tests were excluded due to memory limitations.

In the FFT implementation, the amount of workload assigned to the GPU was approximately 1%, regardless the tested input set size, as shown in Figure 19. Even though the FFT implementation is particular suitable to the CPU, a small part of the workload was assigned to the GPU, as part of the performance modeling process.

In the Barnes Hut implementation, the dynamic scheduler assigned workload to the GPU

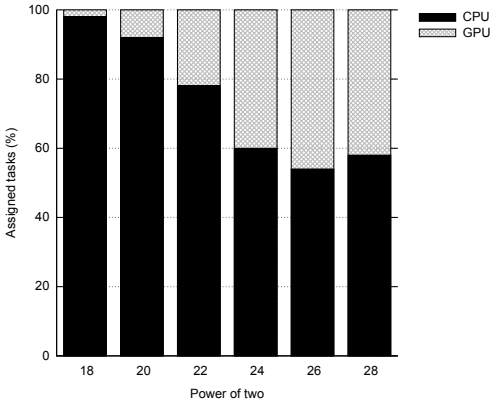


Figure 18: Workload distribution under dynamic scheduling, for SAXPY, ranging from 2^{18} to 2^{28} elements in each vector.

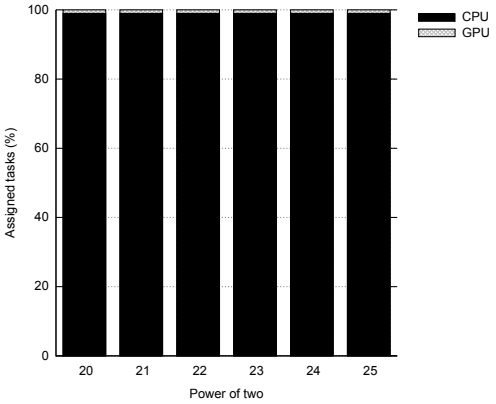


Figure 19: Workload distribution under dynamic scheduling, for FFTs ranging from 2^{20} to 2^{25} elements.

only for systems with 2^{15} or more particles, as Figure 21 shows. The larger the input size, the higher the GPU usage in the overall workload execution (as much as predictable this trend will be valid for larger input sizes). For systems with less than 2^{15} particles, the GPU received less than 1% of the workload, as part as the sampling process.

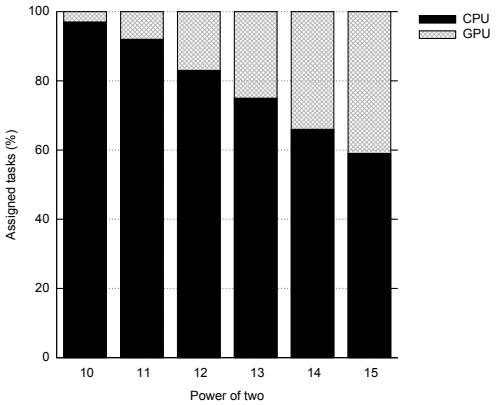


Figure 20: Workload distribution under dynamic scheduling, for the n-Body brute force implementation, for systems ranging from 2^{10} to 2^{15} particles.

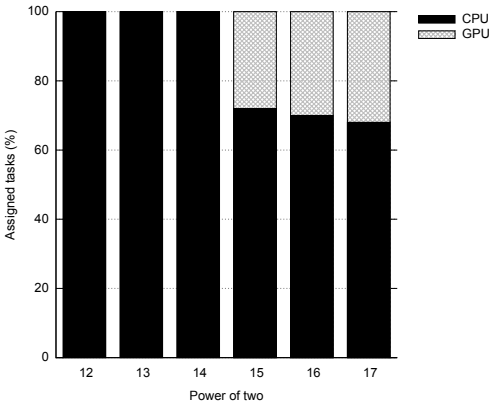


Figure 21: Workload distribution under dynamic scheduling, for the net force calculation in the Barnes Hut algorithm, for systems ranging from 2^{12} to 2^{17} particles.

Although the dynamic scheduler incurs in larger overhead when compared with the static scheduler, it delivered good levels of performance for almost every evaluation run. The required computation to get the scheduling decisions is efficiently overlapped with computation in the devices. As the results also show, the scheduler is only able to leverage the whole platform with algorithms that are not tightly tailored to one particular device.

The presented results about the performance of the proposed scheduling model evidence that, in general, (i) it was capable of achieving very good workload distributions, (ii) the model proved to be efficient for both regular and irregular applications and (iii) the memory transfer latencies, particularly accentuated in accelerators, are taken into consideration when distributing the workload.

Additionally to the overhead of computing the scheduling decisions, other factors can be identified, in the dynamic scheduler, as potential causes of performance loss:

- The performance levels of applications tailored to one particular device is hurt by the performance modeling process, which executes tasks of the same job on every system device. According to the presented results, the impact of these losses might be reduced, though. In particular, the performance of the FFT was decreased in less than 5%, when compared with the best static decision.

- The extensive use of synchronization points is particularly expensive in the dynamic scheduler, as the implementation of the n-Body problem suggests. This particular issue is likely related with implementation details, since the proposed scheduling model is not affected, in theory, by extensive use of synchronization barriers. However, the implementation of synchronization points under dynamic scheduling requires several conditions to be validated (e.g. the scheduler cannot be in flight).

The results presented in Section 5.3.2 aim to test (i) the performance of the GAMA framework (using both the best static decision and the dynamic scheduler), when compared with CPU/GPU libraries providing the same algorithm, (ii) its efficiency η, which expresses how good the framework and the associated scheduler are in leveraging the different architectures in the platform and (iii) the percentage of TPP's usage, expressed in perceptual points[2]. All the trials were executed on the target platform, making use of one CPU-chip and one GPU.

5.3.2 GAMA's Efficiency

Figure 22 (a) presents the execution time of the GAMA framework running on the CPU-chip and on one GPU, both using the dynamic and the static schedulers, in comparison with both Lapack++ (CPU-only) and cuBLAS (GPU-only). Figure 22 (b) presents the efficiency η of the GAMA framework, both under static and dynamic scheduling. Figure 22 (c) shows the delivered GFLOPS of GAMA, under the dynamic scheduler, compared both with cuBLAS and Lapack++. The workload distribution set at the static scheduler follows the best static solution achieved of 2^{27} elements, shown in Figure 14.

[2]The CPU-chip, the GPU and the platform with one CPU-chip and one GPU have TPPs of 102, 1581 and 1683 GFLOPS, respectively and according to Table 3.

As shown in Figure 22 (a), GAMA achieved the lowest execution times when supported by the dynamic scheduler. It delivered speedups of more than 2 and 1.3 times, respectively over Lapack++ and cuBLAS, in most of the input set sizes between 2^{24} and 2^{28} elements in each vector. The dynamic scheduler was also able to achieve more than 80% of efficiency for 2^{27} elements, suggesting good device cooperation and data management.

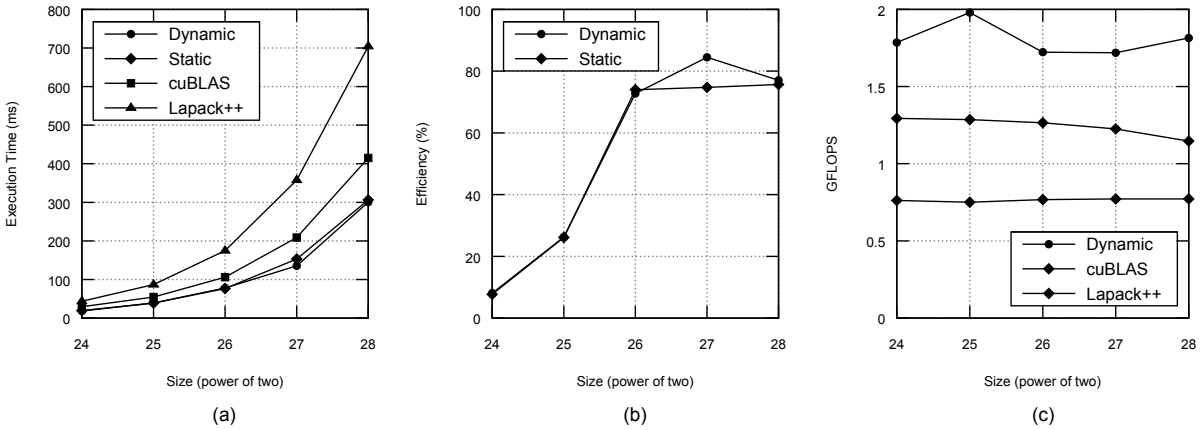


Figure 22: (a) Execution time in milliseconds for the SAXPY algorithm in GAMA, with dynamic and static schedulers, cuBLAS and Lapack++. (b) Efficiency η in percentage for GAMA, with dynamic and static schedulers. (c) GFLOPS for GAMA with dynamic scheduling, cuBLAS and Lapack++. Trials with vectors from 2^{24} to 2^{28} elements.

Neither GAMA nor the used libraries passed the mark of 2 GFLOPS of performance, although the TPP of the CPU is set at 102 GLOPS and the GPU's goes beyond 1500. These results are likely related with the low ratio of computation per memory accesses in SAXPY. Regarding the percentage of used peak performance, Lapack++ is the most efficient framework, by achieving almost 1% of the CPU's TPP, against less than 0.1% (CPU+GPU) and less than 0.01% (GPU) of GAMA and cuBLAS, respectively.

Figure 23 shows experiments with the FFT algorithm, similar to those performed with SAXPY. GAMA, supported both by dynamic and static scheduling, was compared with both cuFFT and the parallel FFTW libraries, as shown in Figure 23 (a). FFTW was set to run with 4 threads, the same number of x86 workers running in GAMA. Figure 23 (b) shows the efficiency η of GAMA both using the static and the dynamic schedulers. Figure 23 (c) compares the delivered GFLOPS of GAMA, with static scheduling, cuFFT and FFTW.

As shown in Figure 23 (a), cuFFT library was considerably faster than both GAMA and FFTW, among the entire range of tested input sizes. The larger the input set size, the higher the speedups of cuFFT over GAMA and FFTW: speedups of more than $\approx$10 times were achieved for the FFT running with 2^{25} elements. Except for the input set with 2^{25} elements, GAMA overcame FFTW in every trial.

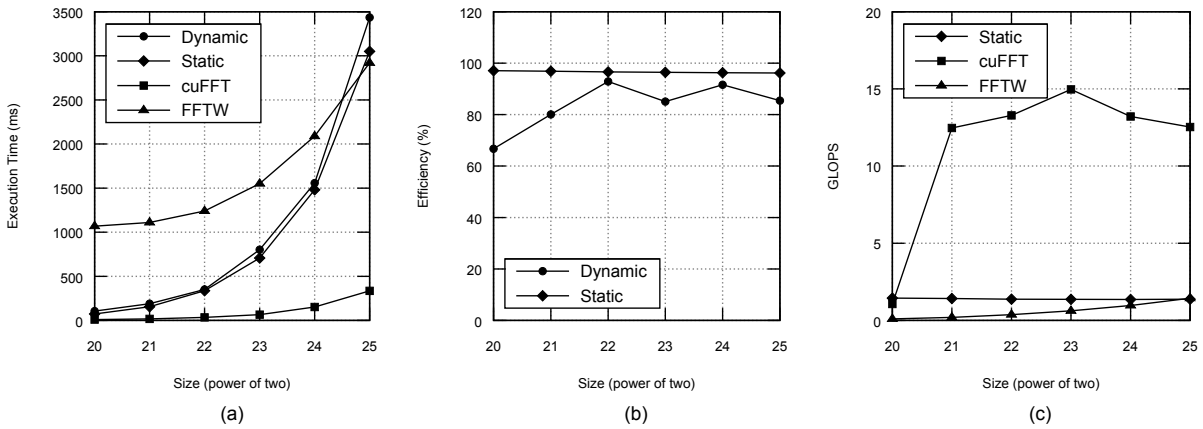


Figure 23: (a) Execution time in milliseconds for the FFT algorithm in GAMA, with dynamic and static schedulers, cuFFT and the parallel FFTW. (b) Efficiency η in percentage of GAMA, with dynamic and static scheduling. (c) GFLOPS for GAMA with static scheduling, cuFFT and FFTW. Evaluation runs with input sets ranging from 2^{20} to 2^{25} elements.

As also shown in Figure 23 (a), GAMA achieves better performance with the static than with the dynamic scheduler. The static scheduler was set at assigning the whole workload to the CPU, the best decision according to Figure 15. The dynamic scheduler, on the other hand, assigns small parts of the workload to the GPU, due to its performance modeling scheme, dropping somewhat the performance levels. Nevertheless, the dynamic scheduler was able to deliver good efficiency levels, of $\eta \geq 80\%$[3], as shown in Figure 23 (b).

With regard to the TPP's usage, FFTW overcame both GAMA and cuFFT. GAMA achieved no more than 1% of the theoretical peak performance of the platform, set at 1683 GFLOPS. cuFFT achieved 15 GLOPS of performance, as shown in Figure 23 (c), which represents 9.5% of the GPU's TPP. FFTW achieved 14.1% of the CPU's TPP, set at 102 GLOPS.

The results evidence that GAMA was seriously penalized for not leveraging the GPU during the computation of the FFT, as a result of the application's CPU-tailorness. The rest of the results show that while cuFFT achieves the best levels of performance in calculating the FFT, FFTW is more efficient than both GAMA and cuFFT in using the hardware's peak performance. These results are especially relevant when choosing computational platforms, which might be chosen with regard to power consumption or raw performance criteria.

The n-Body brute-force solver was ignored in this set of trials, as there are no external libraries providing the exact same implementation in the CPU and in the GPU. The calculation of the delivered FLOPS in the Barnes Hut algorithm was also ignored, since they depend

[3]By definition, CPU/GPU-only algorithms cannot achieve 100% of efficiency.

on the input. As a consequence, the benchmarks related with the usage percentage of the theoretical peak performance were excluded.

Both the devised GAMA and the OpenMP implementations are based on the Barnes Hut implementation presented by Martin Burtscher et al.[4], version 2.2, highly tuned to deliver major performance levels on the GPU [14]. The OpenMP version using during this set of benchmarks was implemented in the context of the GAMA framework, and used in these benchmarks since it is the fastest known Barnes Hut implementation in the CPU.

Figure 24 (a) shows the execution time of GAMA, running both with the dynamic and the static schedulers, in comparison with the OpenMP and Martin Burtscher's versions, respectively running on the CPU-chip and on the GPU. The OpenMP version was set to run with 4 threads, for a fair comparison with GAMA, which ran with four x86 workers. Figure 24 (b) compares both GAMA schedulers with regard to efficiency.

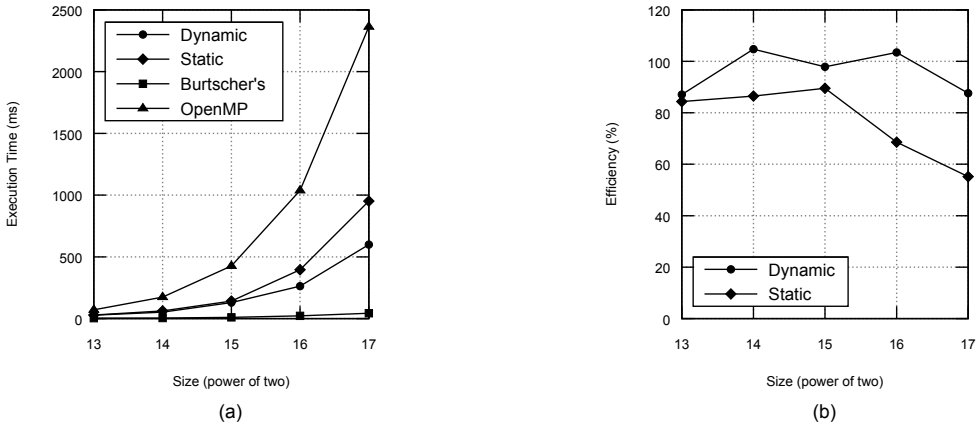


Figure 24: (a) Execution time in milliseconds of the Barnes Hut algorithm running in GAMA, with both dynamic and static scheduling, along with the devised OpenMP and Martin Burtscher's implementations. (b) Efficiency η of GAMA, when using the dynamic and the static schedulers. Evaluation runs for systems from 2^{13} to 2^{17} particles.

As Figure 24 (a) shows, Martin Burtscher's version ran substantially faster than GAMA, especially for larger input set sizes. On the other hand, the OpenMP version was considerably slower than GAMA, either when using the static or the dynamic scheduler. The latter overcame the former, which was especially noticeable in larger input set sizes. While Martin Burtscher's version was more than 12 times faster than GAMA, GAMA was ≈4 times faster than the OpenMP version, when running with the dynamic scheduler.

GAMA achieved more than 100% of efficiency twice, for systems with 2^{14} ($\eta \simeq 104\%$)

[4]Available in http://www.gpucomputing.net/?q=node/1314.

and 2^{16} ($\eta \simeq 105\%$) particles, when equipped with the devised dynamic scheduler, as shown in Figure 24 (b). These results show that both the GAMA framework and the dynamic scheduler can properly take advantage of the resources on the platform, in such a way that their cooperation is extremely effective.

Algorithms with efficiency levels higher than 100% are suitable for HetPlats, under the tested conditions, which does not necessarily mean that an HetPlat is the most suitable platform for the algorithm (e.g. Martin Burtscher's implementation on GPU was substantially faster than GAMA's version, with $\eta > 100\%$).

Section 5.3.3 presents several experiments related to GAMA's scalability, when supported by the devised dynamic scheduler. The performance of the framework was measured in three different scenarios: (i) using the CPU-chip exclusively, (ii) using the CPU-chip and one GPU accelerator and (iii) using the CPU-chip and two GPU accelerators. These trials aimed at proving the scalability of the dynamic scheduler itself and to prove its capacity in leveraging the existent computational resources in function of the platform's setup.

5.3.3 GAMA's Scalability

Figure 25 shows the scalability of SAXPY in GAMA, supported by the devised dynamic scheduler. Results show that the algorithm obtains a speedup of ≈1.19x when a GPU is considered in the computation. However, when adding an additional GPU, the algorithm does not scale, when compared with the CPU+GPU setup. These results indicate that the SAXPY algorithm is excessively memory bound, which is not adequate for platforms with GPU accelerators, since these cannot hide memory transfer latencies with computation.

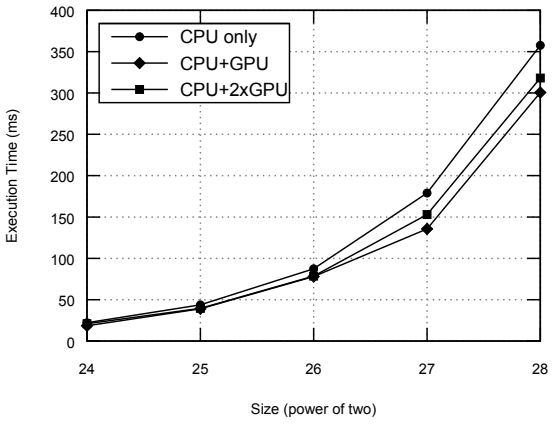


Figure 25: Scalability of SAXPY running in GAMA, using the devised dynamic scheduler.

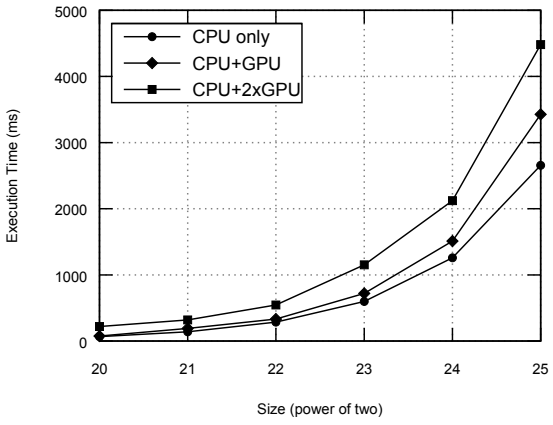


Figure 26: Scalability of FFT running in GAMA, using the devised dynamic scheduler.

The scalability of the FFT is shown in Figure 26. The FFT implementation does not scale,

regardless the number of GPUs populating the platform. When including more GPUs in the computation process, the execution time of the algorithm grows, due to the sampling process. Because of the performance modeling empirical scheme, every GPU receives workload, thus creating a performance bottleneck. These results are strictly related with the performance differences of the algorithm in both architectures.

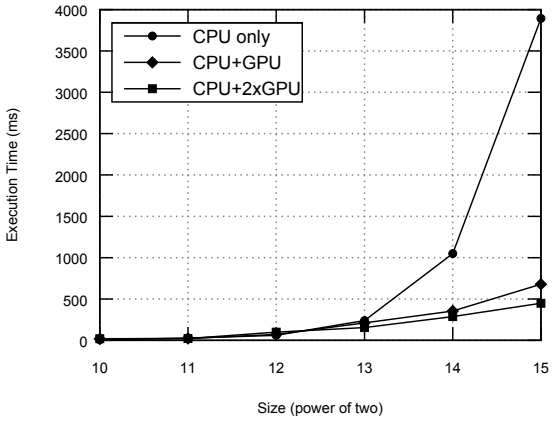


Figure 27: Scalability of the brute force n-Body solver running in GAMA, using the devised dynamic scheduler.

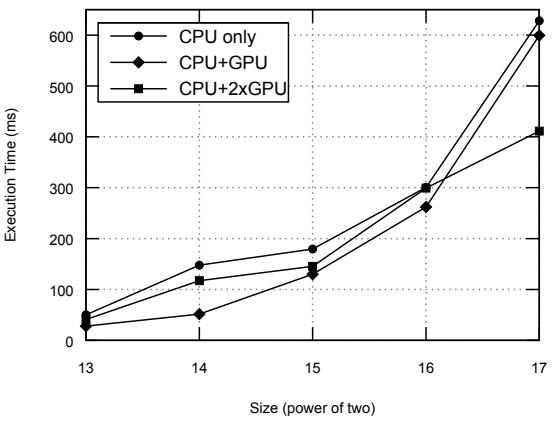


Figure 28: Scalability of Barnes Hut algorithm running in GAMA, using the devised dynamic scheduler.

Figure 27 shows the scalability of the n-Body brute force solver. The algorithm scaled when adding one GPU to accelerate the CPU-chip, which resulted in a speedup of ≈5.74 times. The algorithm continued to scale when an additional GPU was included, obtaining a speedup of ≈1.5 times over the latter, for a system with 2^{15} particles. These results show that systems with 2^{15} particles create enough computational work so the GPU memory transfers are effectively overlapped with computation.

The scalability of the Barnes Hut algorithm is presented in Figure 28. The algorithm scaled for one and two GPUs, depending on the input size. In systems with less than 2^{17} particles, the CPU+GPU setup proved to be the most efficient. However, for systems with 2^{17} particles, the algorithm scaled for two GPUs, achieving a speedup of ≈1.45 times over a CPU+GPU configuration.

In a nutshell, neither algorithms with low ratios of computation per memory accesses nor algorithms suited for a specific CU are adequate for HetPlats. The formers are particularly inefficient in platforms whose accelerators suffer high memory transfer latencies, which is generalized on accelerators connected to the platform by PCIe channels. On the other hand, algorithms with enough computation to hide these latencies are suited and scale on HetPlats, as proved by the n-Body brute force solver and the Barnes Hut algorithms.

The results also show that the proposed scheduling model is capable of efficiently considering memory access latencies, since the larger the input data sets, the higher the proportional usage of PCIe-connected devices. These trials may be extended to other accelerators in the future. Although the suitability of the target platform to memory bound algorithms, such as SAXPY, has proved to be reduced, both the GAMA framework and the devised scheduler have efficiently handled the algorithm, by beating both Lapack++ and cuBLAS.

Summary and conclusions:

This chapter validated the proposed scheduling model and mechanism on a platform composed of one quad-core CPU-chip and two NVIDIA GTX580 GPUs. Results show the model as able to achieve good performances, when scheduling both regular and irregular applications. The model's implementation failed at achieving the best class of scheduling decisions on a particular case study, which is likely related with implementation details.

The GPU's usage appears to be strictly related with the size of the algorithm's input set. The results show that the larger the algorithm's input set, the higher the GPU usage levels. Large workloads are required to hide the latencies of PCIe channels, used by GPUs to access pinned memory. This issue is automatically solved by the dynamic scheduler, able to use the available computational resources according to the platform's setup and the algorithm's workload size.

Some evaluation results were presented to compare the raw performance and the percentage of the used platform's TPP delivered by GAMA and by some fast libraries. GAMA overcame both Lapack++ and cuBLAS on the raw performance of SAXPY, but was less efficient than Lapack++ on the percentage of the used platform's TPP. cuFFT and Martin Burtscher's version of the Barnes Hut algorithm were considerably faster than GAMA in executing FFT and Barnes Hut, respectively. These results indicate than highly optimized and tuned code may be more efficient than heterogeneous platforms.

The scalability of the GAMA framework was also tested using the proposed scheduling model. The results indicate that the implementation of the dynamic scheduler does not limit the scalability of the algorithm, since the trials with both n-Body solvers showed that both algorithms scale with either one or two GPUs, depending on the workload's size. In particular, the brute force n-Body solver delivered a speedup of $\approx$1.5 times when adding a second GPU over a single accelerator setup.

Chapter 6

Conclusions & Future Work

This chapter concludes the dissertation, presenting an overview of the obtained results, related both with the proposed scheduling model, the performance of GAMA and the experience on heterogeneous platforms. Some guidelines of future work are suggested.

6.1 Conclusions

This dissertation presented a model and the respective implementation of a dynamic scheduling model to address both regular and irregular applications. Its respective scheduling mechanism assigns the workload based on an empirical performance model, which provides information about the suitability between one device and one computational task. The workload is assigned in chunks to control the workload imbalance, which may occur due to performance model's inaccuracy, especially noticeable in irregular applications.

The implementation of the proposed model was tested in the scheduling of four algorithms, which include the SAXPY, the FFT and two n-Body solvers, on a heterogeneous platform containing one CPU-chip and two GPUs. The SAXPY, the brute force n-Body solver and the FFT are regular applications, whereas the Barnes-Hut algorithm, implemented according to the implementation of Martin Burtscher et al. [14], is irregular.

To address the key issues on the scheduling of regular and irregular applications on heterogeneous platforms, empirical *per-task* performance-models were identified as an adequate mechanism to schedule regular applications, whereas dynamic workload distribution, in chunks, was identified as a solution to address irregular applications. These methods were proved to be compatible.

In general, the implementation of the proposed scheduling model, which embodies the methods described above, found proper and accurate scheduling decisions, both in regular and irregular applications. As the results with the n-Body brute force solver proved, even inaccurate decisions follow proper workload distributions with respect to the suitability between the devices and the tasks that compose the application. As a result, these methods

are effective to solve the problems that arise on the addressed type of scheduling.

In particular, the implemented empirical performance model proved to be accurate for regular algorithms, both on the estimation of the execution time of a pair (*task*,*device*) and on the estimations of the execution time of an entire queue. The dynamic model's behavior appears to introduce negligible overhead, since the proposed model was able to find the optimal decision on both SAXPY and FFT algorithms.

The failures in which the proposed model incurred in are very likely related with implementation details, either related with the adulteration of the model or the cost of synchronization points in GAMA. Other potential causes of performance losses include poor data locality and excessive overhead caused by the dynamic scheduler, when calculating the scheduling decisions, but none of this has been noticed and they appear not to be causing the model's malfunctioning, since the n-Body algorithm is the only case study affected.

With regard to the side questions which were expected to arise during the experiments with the proposed scheduling mechanism, some conclusions were drawn with respect to: (i) the the class of algorithms that suits HetPlats with GPUs, (ii) the relation between the accelerator's usage and the application's workload size, and (iii) to the scalability of applications in HetPlats.

Regarding to (i), the implementation of algorithms to work on heterogeneous platforms and GAMA-like frameworks must follow certain guidelines to efficiently take advantage of the computational resources:

- Algorithms with low computation/communication ratios are not adequate for HetPlats incorporating GPUs as main accelerators, as shown in the results related with the SAXPY algorithm. The scalability and the percentage of the used platform's theoretical peak performance might be substantially reduced.

- Implementations to run on HetPlats should not be tailored to one particular device on the system. As proved by the CPU-chip tailored FFT implementation, other devices are not selected to perform computation, which leads to low ratios of utilization and occupancy, in addition to large distances to the platform's theoretical peak performance.

Regarding to (ii), the relation between the GPU's usage and the workload size has proved to be straight: when an application is not tailored to the GPU-chip, as the devised FFT implementation, the larger the input set, the higher the GPU's usage. This trend was true for almost every trial expect in a single input set of SAXPY, which is likely an outlier. For the Barnes Hut algorithm, this trend was even more noticeable, since the GPU was not used at all for systems with 2^{14} or less particles.

With respect to (iii), and as proved by the experiments with the presented n-Body solvers, the scalability of the algorithm is strictly related with large input set sizes. This is crucial to hide the latency of PCIe channels, the means of connection between the GPU boards and the rest of the platform. As predicted, similar results may be experimented for different accelerators using similar connecting schemes.

6.2 Future Work

The obtained results in this dissertation motivates further work on this theme, especially in (i) refining the current implementation of the proposed model and (ii) comparing the proposed model with other scheduling models, especially those designed to address irregular applications. Some of the most interesting branches of future work include:

- As in theory the proposed model should not fail at finding the best scheduling solution in the implemented brute force n-Body solver, which is likely to be related to its implementation, some refinement should be made over it.
- Implement the adaptive version of the proposed scheduling mechanism, i.e. with work stealing/donation mechanisms, and evaluate its impact on the performance of the implemented case studies.
- Implement more case studies to test the proposed model. In particular, applications with extreme irregularity (e.g. chaos applications) may be adequate.
- The sampling process, as part of the performance modeling, hurts performance on applications that are tailored to a specific device on the platform. In that sense, other ways of performance modeling might be adopted. Some data analysis techniques may be a potential solution.
- Implement and compare other scheduling models for heterogeneous platforms, especially those which combine techniques to address irregular applications.
- Implement models to take advantage of highly tuned libraries for specific devices, to improve the productivity of the development of applications.

Bibliography

[1] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shangai, China, December 2010.

[2] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *Proceedings of the International Euro-Par Workshops 2009, HPPC'09*, Lecture Notes in Computer Science, Delft, The Netherlands, August 2009. Springer.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue*, 2010.

[4] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for GPU architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 105–114, New York, NY, USA, 2010. ACM.

[5] David H Bailey and Allan Snavely. Performance modeling: Understanding the past and predicting the future. In *In Euro-Par Parallel Processing: 11th International Euro-Par Conference*, 2005.

[6] Ioana Banicescu and Vijay Velusamy. Load Balancing Highly Irregular Computations with the Adaptive Factoring. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[8] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000.

[9] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations By Work Stealing. *J. ACM*, 46(5):720–748, September 1999.

[11] Brett Bode, David M. Halstead, Ricky Kendall, Zhou Lei, and David Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, pages 27–27, Berkeley, CA, USA, 2000. USENIX Association.

[12] J. Brantley and C. Gregg. Dynamic Scheduling of Parallel Code for Heterogeneous Systems. Technical Report, 2010.

[13] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 79–92, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Martin Burtscher and Keshav Pingali. *GPU Computing Gems Emerald Edition: An efficient CUDA implementation of the tree-based barnes hut n-body algorithm*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[15] K. Hazelwood K. Skadron C. Gregg, M. Boyer. Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data. In *Proceedings of the 2nd Workshop on Applications for Multi- and Many-Core Processors*, San Jose, CA, June 2011.

[16] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[17] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[18] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68:1370–1380, October 2008.

[20] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[21] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297–301, 1965.

[22] Eugen Dedu, Stephane Vialle, Claude Timsit, and Supélec Metz Campus. Comparison of OpenMP and Classical Multi-Threading Parallelization for Regular and Irregular Algorithms. In *In SNPD*, 2000.

[23] Gregory Diamos. *Harmony: An Execution Model For Heterogeneous Systems.* PhD thesis, Georgia Institute of Technology, December, 2011.

[24] Gregory Diamos and Sudhakar Yalamanchili. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In *HPDC'08*, Boston, Massachusetts, USA, June 2008. ACM.

[25] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[26] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[27] T. Gautier, J.L. Roch, and G. Villard. Regular Versus Irregular Problems and Algorithms. In *In Proc. of IRREGULAR'95*, pages 1–25. Springer, 1995.

[28] W. M. Gentleman and G. Sande. Fast Fourier Transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, AFIPS '66 (Fall), pages 563–578, New York, NY, USA, 1966. ACM.

[29] R. L. Graham. Bounds on multiprocessing timing anomalies. *Siam Journal on Applied Mathematics*, 17(2):416–429, 1969.

[30] Chris Gregg and Kim Hazelwood. Where is the Data? Why You Cannot Debate GPU vs. CPU Performance Without the Answer. In *International Symposium on Performance Analysis of Systems and Software*, ISPASS, Austin, TX, April 2011.

[31] Dominik Grewe and Michael O'Boyle. *A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL*, volume 6601, pages 286–305. Springer Berlin Heidelberg, 2011.

[32] Minyi Guo, Jiannong Cao, Weng-Long Chang, Li Li, and Chengfei Liu. Effective OpenMP Extensions for Irregular Applications on Cluster Environments. In *GCC (2)'03*, pages 97–104, 2003.

[33] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast fourier transform. *IEEE Signal Processing*, 1(4):14–21, 1984.

[34] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[35] Dixie Hisley, Gagan Agrawal, Punyam Satya-narayana, and Lori Pollock. Porting and Performance Evaluation of Irregular Codes using OpenMP. In *Proceedings of the first european workshop on OpenMP*, pages 47–59, September, 1999.

[36] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM.

[37] Xiaohuang Huang, C.I. Rodrigues, S. Jones, I. Buck, and Wen mei Hwu. XMalloc: A Scalable Lock-free Dynamic Memory Allocator for Many-core Machines. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1134 –1139, July 2010.

[38] Francisco D. Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, Timothy Wentz, and Robert van de Geijn. Unleashing DSPs for General-Purpose HPC - FLAME Working Note #61. Technical Report TR-12-02, The University of Texas at Austin, Department of Computer Sciences, February 2012.

[39] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] Sixto Ortiz Jr. Chipmakers ARM for Battle in Traditional Computing Market. *Computer*, 44(4):14–17, April 2011.

[41] David Kaeli and David Akodes. The convergence of HPC and embedded systems in our heterogeneous computing future. *Computer Design, International Conference on*, 0:9–11, 2011.

[42] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A Characterization and Analysis of PTX Kernels. In *IISWC'09*, Austin, Texas, USA, Ocetober 2009. ACM.

[43] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. Modeling GPU-CPU workloads and systems. In *Proceedings of the 3rd Workshop on General-Purpose Computation*

on Graphics Processing Units, GPGPU '10, pages 31–42, New York, NY, USA, 2010. ACM.

[44] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(1):1–47, December 2003.

[45] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? *SIGPLAN Not.*, 44(4):3–14, February 2009.

[46] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Y. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS*, pages 287–296, 2008.

[47] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO*, pages 45–55, 2009.

[48] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[49] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[50] Ananya Muddukrishna. Exploiting locality in OpenMP task scheduling. KTH Information and Communication Technology. *MSc thesis*, 2010.

[51] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient support for irregular applications on distributed-memory machines. *SIGPLAN Not.*, 30(8):68–79, August 1995.

[52] Alin Muraraşu, Josef Weidendorfer, and Arndt Bode. Workload balancing on heterogeneous systems: A case study of sparse grid interpolation. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 345–354. Springer Berlin Heidelberg, 2012.

[53] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka. Statistical power modeling of GPU kernels using performance counters. In *Green Computing Conference, 2010 International*, pages 115 –122, August, 2010.

[54] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[55] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[56] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. MDR: performance model driven runtime for heterogeneous parallel platforms. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 225–234, New York, NY, USA, 2011. ACM.

[57] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. Technical Report TR-09-05, Department of Computer Science, The University of Texas at Austin, February 2009.

[58] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: accelerating flow analysis with GPUs. *SIGPLAN Not.*, 46:511–522, January 2011.

[59] Keith H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[60] Vignesh T. Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, New York, NY, USA, 2010. ACM.

[61] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[62] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[63] Oliver Sinnen. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[64] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[65] Michael Süßand Claudia Leopold. Implementing irregular parallel algorithms with OpenMP. In *Proceedings of the 12th international conference on Parallel Processing*, Euro-Par'06, pages 635–644, Berlin, Heidelberg, 2006. Springer-Verlag.

[66] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A Tighter Analysis of Work Stealing. In *ISAAC (2)*, pages 291–302, 2010.

[67] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[68] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In Michael Doggett, Samuli Laine, and Warren Hunt, editors, *High Performance Graphics*, pages 29–37. Eurographics Association, 2010.

[69] J. D. Ullman. NP-complete scheduling problems. *J. Comput. Syst. Sci.*, 10:384–393, June 1975.

[70] Sundaresan Venkatasubramanian and Richard Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd international conference on Supercomputing*, ICS 2009, pages 244–255, New York, NY, USA, 2009. ACM.

[71] Markus Wittmann and Georg Hager. Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems. *CoRR*, 2010.

[72] Shucai Xiao, Heshan Lin, and Wu-chun Feng. Accelerating Protein Sequence Search in a Heterogeneous Computing System. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1212–1222, Washington, DC, USA, 2011. IEEE Computer Society.