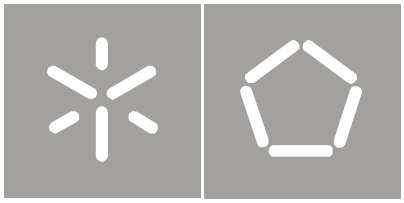
Universidade do Minho
Escola de Engenharia

Roberto Carlos Sá Ribeiro

Portability and Performance in Heterogeneous Many-core Systems

Outubro de 2011



Universidade do Minho
Escola de Engenharia

Roberto Carlos Sá Ribeiro

Portability and Performance in Heterogeneous Many-core Systems

Tese de Mestrado
Informática
Trabalho efectuado sob a orientação do
Professor Luís Paulo Santos

Outubro de 2011

Agradecimentos

A concretização desta dissertação muito deve ao apoio, motivação e estímulo de várias pessoas que fazem parte da minha vida pessoal, social e académica, pelo que, gostaria de expressar meus sinceros agradecimentos:

Ao Professor Luís Paulo Santos, não só pela orientação mas pela motivação, interesse, experiência e disponibilidade incontestável;

Ao João Barbosa, pelo apoio e confiança depositada;

Ao Professor Doutor Alberto Proença, pela docência, aconselhamento e oportunidades oferecidas com vista ao engrandecimento da minha experiência;

À restante equipa docente, pela introdução e formação em *high performance computing*;

À instituição Universidade do Minho, pelo ambiente de formação de qualidade proporcionado ao longo de cerca de cinco anos;

Aos amigos e companheiros, pelo apoio e incentivo;

Um agradecimento especial à Daniela, pelo apoio incondicional e paciência;

Aos meus pais e irmãos, pelo incentivo, confiança e fonte de inspiração.

Portabilidade e Desempenho em Sistemas Paralelos Heterogéneos

Resumo

Os sistemas de computação actuais são constituídos por uma multiplicidade de recursos computacionais com diferentes arquitecturas, tais como CPUs multi-núcleo e GPUs. Estas plataformas são conhecidas como Sistemas Paralelos Heterogéneos (SPH) e à medida que os recursos computacionais evoluem este sistemas oferecem mais paralelismo e heterogeneidade. Explorar de forma eficiente esta multiplicidade de recursos exige que o programador conheça com alguma profundidade as diversas arquitecturas, modelos de computação e ferramentas de desenvolvimento que lhes estão associados. Problemas de portabilidade, espaços de endereçamento de memória disjuntos, distribuição de trabalho e computação irregular são alguns exemplos que precisam de ser abordados para que seja possível explorar eficientemente os recursos computacionais de um SPH.

O objectivo desta dissertação é conceber e avaliar uma arquitectura base que permita a identificação e estudo preliminar dos potenciais obstáculos e limitações ao desempenho de um sistema de execução que explore SPH. É proposto um sistema de execução que simplifica a tarefa do programador de lidar com todos os diferentes dispositivos disponíveis num sistema heterogéneo. Este sistema implementa um modelo de programação e execução com um espaço de endereçamento unificado controlado por um gestor de dados. É também proposta uma interface de programação que permite ao programador definir aplicações e dados de forma intuitiva. Quatro diferentes escalonadores de trabalho são propostos e avaliados, que combinam diferentes mecanismos de partição de dados e diferentes políticas de atribuição de trabalho. Adicionalmente é utilizado um modelo de desempenho que permite ao escalonador obter algumas informações sobre a capacidade de computação dos diversos dispositivos.

A eficiência do sistema de execução foi avaliada com três aplicações - multiplicação de matrizes, convolução de uma imagem e um simulador de partículas baseado no algoritmo Barnes-Hut - executadas num sistema com CPUs e GPUs.

Em termos de produtividade os resultados são prometedores, no entanto o combinar de escalonamento e particionamento de dados revela alguma ineficiência e necessita de um estudo mais aprofundado, bem como o gestor de dados, que tem um papel fundamental neste tipo de sistemas. As decisões influenciadas pelo modelo de desempenho foram também avaliadas, revelando que a fidelidade do modelo de desempenho pode comprometer o desempenho final.

Portability and Performance in Heterogeneous Many Core Systems

Abstract

Current computing systems have a multiplicity of computational resources with different architectures, such as multi-core CPUs and GPUs. These platforms are known as heterogeneous many-core systems (HMS) and as computational resources evolve they are offering more parallelism, as well as becoming more heterogeneous. Exploring these devices requires the programmer to be aware of the multiplicity of associated architectures, computing models and development framework. Portability issues, disjoint memory address spaces, work distribution and irregular workload patterns are major examples that need to be tackled in order to efficiently explore the computational resources of an HMS.

This dissertation goal is to design and evaluate a base architecture that enables the identification and preliminary evaluation of the potential bottlenecks and limitations of a runtime system that addresses HMS. It proposes a runtime system that eases the programmer burden of handling all the devices available in a heterogeneous system. The runtime provides a programming and execution model with a unified address space managed by a data management system. An API is proposed in order to enable the programmer to express applications and data in an intuitive way. Four different scheduling approaches are evaluated that combine different data partitioning mechanisms with different work assignment policies and a performance model is used to provide some performance insights to the scheduler.

The runtime efficiency was evaluated with three different applications - matrix multiplication, image convolution and n-body Barnes-Hut simulation - running in multi-core CPUs and GPUs.

In terms of productivity the results look promising, however, combining scheduling and data partitioning revealed some inefficiencies that compromise load balancing and needs to be revised, as well as the data management system that plays a crucial role in such systems. Performance model driven decisions were also evaluated which revealed that the accuracy of a performance model is also a compromising component.

Contents

Chapter 1

Introduction

Current computing systems are heterogeneous machines in the sense that, besides the general purpose central processing unit (CPU), also integrate additional co-processors which, most often, exhibit architectures and computing paradigms different from the main CPU. Multi-core CPUs, many-core GPUs, Cell Broadband Engine are some examples of devices that integrate these platforms which are known as Heterogeneous Many-core Systems (HMS). Computing systems are thus offering more parallelism while simultaneously becoming more heterogeneous [37, 54]. Amdahl's law of the multi-core era [31] suggests that these systems have more potential than homogeneous systems to improve parallel applications.

However, using a HMS may not be an easy task due to the architectural differences that compose a heterogeneous system. Efficient program execution in these platforms requires the programmer to be aware of the multiplicity and heterogeneity of the available devices in order to efficiently exploit them. The user must have some knowledge of the different execution and memory models associated with the different devices that are also reflected in the devices' programming models.

Due to the divergences in the architectural descriptions of most of the co-processors they feature their own interface and development tools. Code used by these devices is generally compiled with a specific compiler and according to a specific programming model. This makes the migration of computation a problem not only in run-time but also in compile time since the program code is designed for a specific platform or device.

This lack of portability is not only relative to the code but also to the performance. As-

sumptions of performance in an algorithm implementation may not be valid if the computing platform changes. As mentioned, associated with each platform there is a set of models that impact the code, memory hierarchy, control flow, data access and concurrent execution of threads, among others. All performance considerations must take this diversity of factors into account; if one or more of these factors changes, then the performance may seriously be affected. Code and performance portability is thus required and constitutes a research challenge[59].

One of the potential drawbacks when using HMS is the disjoint memory address spaces of each device. Each device may have its own memory space that requires explicit copy of data in order to perform the computation. For example, performing a matrix multiplication on a GPU requires to explicitly copy the three matrices to the device, execute the computation and copy the result back to the host system. These copies not only require explicit intervention from the programmer as well as they become more error prone and may represent a bottleneck due to a limited bus bandwidth. Programming HMS requires explicitly handling data movements and accesses, hindering the programmer from concentrating on the application functionality. Providing an intuitive and efficient data management abstraction level, that releases the programmer from such concerns, is a major requirement to enable increased programming productivity and consistency.

Heterogeneous systems are inherently parallel, both between different devices as within each device, since these are often composed by multiple processing cores. Deciding how to partition an application workload into tasks and on which device shall each task be executed are thus fundamental questions.

Each device in a heterogeneous architecture platform has different purposes and compute capabilities. For example, a GPU has about 1000 GFLOPS of theoretical peak performance and is an ideal platform for data parallel tasks but limited in memory and bus bandwidth, while a CPU is denoted by complex control logic with a sophisticated memory hierarchy. The matching of algorithm classes to the different platforms available is still a research topic [40]. In an HMS work must be distributed among devices such that a single device will not stall the others because it takes too long to produce some result that is required to complete the job or to satisfy some data dependency that will allow other tasks to proceed. There is a scheduling choice to be taken in order to keep the system busy and balanced such that the full computing power of the heterogeneous system can be leveraged in order to minimize the

execution time.

One of the greatest causes of unbalancing are the computation and data access patterns of the application. Irregular applications tend to be hard to map in a heterogeneous system since there is no pre-knowledge of the amount of work and data accesses. When assigning a certain amount of work to a device it could take an arbitrary amount of time to finish. This causes load balancing to be hard, making scheduling of irregular applications in HMS a challenge.

The goals of this dissertation are threefold: (i) identify the main requirements of a run-time system that efficiently addresses heterogeneous systems, ensuring code and performance portability, thus increasing programmer productivity; (ii) design a base architecture and implement a prototype for such system; (iii) assess the above design performance and suitability for efficiently handle heterogeneous systems, thus enabling the identification and preliminary evaluation of major bottlenecks and limitations. These goals will provide insight into the development of a platform that tackles HMS, which rises new challenges, as compared to homogeneous systems, that will be identified and approached throughout this work.

Therefore, this dissertation entails designing and evaluating an unified programming and execution model for HMS, which handles scheduling and data management activities, thus releasing the application programmer from such concerns and allowing him to concentrate on the application's functionality. The proposed programming and execution model introduces transparency with respect to the burden of handling multiple platforms and simultaneously enables the runtime system to make dynamic decisions without any intervention from the programmer.

An Application Programming Interface (API) is proposed, through which the user specifies the application functionality and data. Functionality will be exposed using a kernel approach. The user will provide a kernel implementation for each platform that will be called by the run-time system to perform the computation in the assigned device. Data will be specified using a data abstraction mechanism that encapsulates information about the problem data. This allows the system to transparently manipulate data separating the computation from data management.

As previously mentioned the existence of multiple disjoint memory address spaces, associated with different devices and hierarchy levels as well as exhibiting diverse transfer band-

widths and latencies, constitutes a potential performance bottleneck for exploiting HMS. The proposed system is endowed with a data management system that will be responsible for tracking all data transfers and locations in a transparent way. This will enable the framework to leverage data locality using a simple cache protocol that will reduce memory copies whenever possible, trying to minimizing the problem of limited bus bandwidth. Data management is a very subtle and complex subject that can seriously influence the final performance results.

Several factors may influence scheduling decisions in a HMS. A decision may be based on several parameters that describe the capabilities of the available devices and the load description of a given amount of work. This suggests the definition of a performance model that describes both the workload and the devices' characteristics and which allows for informed scheduling decision making. One such performance model is proposed, allowing a simplified description of both the workload and the computing capabilities and requiring the application programmer to provide some mechanisms in order to gather the required information. Additionally, several scheduling strategies are evaluated which explore different work decomposition and mapping policies in order to tackle the load balancing and resource utilization problem.

The proposed approach is evaluated over NVIDIA GPU cards and Intel XEON quad-core processors. The obtained results provide some insights of the impact of efficient scheduling and data management strategies across a HMS, for both regular and irregular applications.

This dissertation is composed by five more chapters. Chapter 2 introduces the problem statement, presents an overview of the available heterogeneous technologies and some related work. Chapter 3 describes the proposed programming and execution model and framework design details. Chapter 4 and 5 are dedicated to evaluations and results, and the dissertation is concluded in Chapter 6.

Chapter 2

The problem statement

2.1 The search for performance

In the last decades the scientific community has substantially increased the demand for high performance computing (HPC) systems . With more simulation instead of physical testing, more complex phenomena modelling, more products and systems being simulated, computing technology - either in intelligence, life sciences, manufacturing, energy, defences, etc. - has become more ubiquitous and challenging.

To face this claim for performance, chip manufacturers improve their computational solutions following the technological evolutions and limitations. The most prominent and broadly used solutions nowadays embrace parallelism as a key factor to achieve performance. This is the result of a change in the technological evolution whose grounds are increasing clock frequency and transistor count. When chip manufacturers reached the physical limitations with those type of improvements they focused their efforts in revising their product architectures minimizing latency and maximizing throughput [49].

Computing technology today offers several solutions with several different purposes. For example, a Central Processing Unit (CPU) is endued with execution units and complex control primitives which allow it to be a versatile and flexible tool for process and control. It implements several types of parallelism as well as good support for conditional branching and speculative execution. On the other hand Graphical Processor Units (GPU) were designed to solve rasterization problems from computer graphics. The main objective of a GPU is to provide game and film industry with better graphics and real-time rates. These devices offer

high level of throughput with a high core count but sacrifice control and execution of complex computation. Manufacturers kept improving it and with the introduction of new features, the HPC community was able to explore the potential of parallelism that these devices offer [48]. The Cell architecture proposed by IBM is a successfully heterogeneous architecture which tries to gather the best of the two previous architectures in a single platform. It is developed for high-end server markets but it is also present in the video-game console Sony PlayStation 3. Other co-processing devices can be found across the market that best fit the different purposes with different compute capabilities and different ends.

Furthermore, some of these devices can work together in a single system or in high-bandwidth network connected systems (e.g. Clusters) exposing even more parallel processing power. These type of systems are becoming widely used and are known as Heterogeneous Many-core Systems. They can be easily found in a consumer market or in a larger scale in the Top500 list[62].

But there is a setback. There is an increasing demand for performance and parallel computing power and vendors are remarkably releasing more and more powerful and affordable computing resources. But there is a wide and growing gap between the resource capabilities and the actual benefits that can be extracted by programming and implementing applications. This gap is known as "software gap" and it is the object of several research studies [7].

2.2 The Heterogeneity problem

The HPC community has increased substantially the demand for computational power to which manufacturers answered with more parallelism and more types of computing platforms with larger number of processing elements. This lead the developers to explicitly express parallelism in their applications in order to fully exploit the available resources. As the demand grows wide in the various research fields, not only the processing elements are increasing but they are also becoming more heterogeneous and thus raising more challenges.

In order to minimize the latency, manufacturers equipped some devices with their own memory banks. Some CPU devices have up to 3 levels of cache and a set of registers. The

system host RAM is shared with other CPUs and devices. A GPU is connected to the system by a PCI Express (PCIe) bus through which all the communication is done. Data must be explicitly copied to this address space in order to perform any computation since the memory space is disjoint from the system host memory. This model is similar to distributed memory systems, where the different processors access their own memory banks.

Memory transfer cost is thus, a potential performance bottleneck when using co-processors. Mainly duo to the bus limited bandwidth but also to reduced development productivity, since the programmer needs to explicitly handle all memory copies and be perfectly aware of data location and coherence. Thus, in order to efficiently exploit a HMS the memory transfer cost must be minimized such that system throughput levels increase. In order to increase both development and productivity and runtime performance it is fundamental to separate data management from the computation, which implies that the developer is kept agnostic to the data location. Such data management suggests that data be transferred dynamically, on demand, minimizing the volume of data movement while simultaneously maintaining a database of data location and status.

Heterogeneous architectures imply different execution and programming models. Each device may have its own programming language and tools which in most cases are not portable [13, 45, 29, 43]. In practical terms, for a programmer to develop applications for a HMS he needs, at least, to have some knowledge of each programming model of each device [57]. This seriously reduces development productivity when using heterogeneous platforms efficiently.

In order to maximize performance, an application is designed and carefully tuned to fully utilize each resource computing capability. It follows the device specific architecture and execution model and the algorithm is implemented accordingly. For example, consider a vector addition in a GPU and CPU. Usually, in the GPU, a thread is assigned to perform the sum of the two values of a single position resulting in invoking the same number of threads as the number of elements. This design exhibits a good performance in this type of devices since they were developed to support a high thread count. However, if implementing this model in a CPU, the performance will probably be unacceptable due to the higher overheads relative to thread creation and management. Each algorithm implementation is designed targeting a platform computing model that the developer explores and tackles in order to achieve the minimum execution time. If the platform changes, the developer needs

to re-write and potentially re-design all the algorithm such that it suits the new platform.

Although these portability issues have been widely studied[21, 58, 68], code and performance portability are still a research challenge [59].

In computer science one can find several different types of applications. Data intensive applications are those who exhibit more data accesses with simple computations e.g. matrix multiplication. Compute intensive applications use complex algorithms that push to the limits the functional units of the device with lower memory access rates, e.g., Data encryption. Applications can also be categorized has regular or irregular. A regular application has predictable memory accesses and computations that can be managed by a programmer and improved. An irregular application exhibits an unpredictable workload and data access patterns making optimizations a much more challenging task.

There are also different types of parallelism, such as instruction-level, data-parallelism and task-level parallelism, each one suited for a different computing model. Single Instruction Multiple Thread (SIMT) devices, such as the GPU, for example, are designed for data-parallelism in regular applications benefiting from techniques like coalesced memory accesses. On the other hand, a IBM Cell is a very flexible co-processor that can offer high-end performance in several computing paradigms. Several details influence a good match between the device computing model and the application patterns, an improper assignment of a task with an improper workload to a device may seriously affect the performance. Therefore, different devices exhibit different performance levels when addressing the same application type. This fact raises some challenges when handling these systems. For example a system composed with one GPU and one CPU executing a matrix multiplication algorithm. If the matrix is equally divided to both devices, the GPU, due to its higher number of cores and simple computation, will finish much faster than the CPU and the whole system has to wait until the CPU finishes to deliver the result. Properly distributing and balancing work among processors is thus a relevant task in order to keep all the processors busy and minimizing the execution time. This is known as load balancing and it is one of the major causes of inefficiency in a HMS.

An application can be seen as a collection of jobs and job dependencies. This allows identification of stages of the application that can run concurrently, thus reducing the execution time. A job is a computation not yet ready for execution that is performed over some data. While a task is a part of a job that can be done in parallel. A job can be partitioned into

smaller tasks. The granularity of these tasks is not trivial since it is influenced by several factors, most of them related with the device architecture and overall performance. To achieve good parallel performance, choosing the right granularity for the application is crucial [51]. Granularity is the amount of workload each parallel task will have. If the granularity is too coarse the system may become imbalanced, on the other hand if granularity is too fine the communication overhead will potentially increase reducing throughput.

The challenge is to determine the right granularity avoiding load imbalance and communication overhead. Choosing the proper granularity is heavily related to device capabilities and architectural features like cache sizes, thread model, among others. Selecting the correct task granularity for a given device is thus a fundamental requirement in order to minimize execution time within an heterogeneous system.

The workload of an application is generally related to the associated data, e.g. a matrix multiplication workload depends on the size of the matrices, a signal processing algorithm depends on the length of the signal. It is the data set size that typically defines the amount of computation required to achieve the solution, particularly on regular applications. This association enables the programmers and runtime systems to quantify a task workload and perform load partitioning defining tasks with different granularities. This is achieved by partitioning the original data set into smaller subsets and associating a task with each of these subsets. However, this feature requires that data is kept consistent and mechanisms to perform data divisions are required.

Summarizing, a heterogeneous systems programmer is faced with the following set of difficulties which must be tackled: (i) disjoint memory address spaces which require explicit data movement among computing devices; (ii) code and performance portability problems, due to the different architectures and computing models; (iii) correct choice of granularity and proper map of task across devices considering the different application load patterns and different compute capabilities.

All these challenges and difficulties are thus a consequence of the heterogeneity, which suggests tackling them with an unified execution model. Such model combined with a programming model, data management system (DMS) and a scheduling methodology will increase development productivity and improve overall performance.

A solution for efficient and productive use of heterogeneous platforms should then em-

brace the following features:

- data abstraction and distribution mechanisms: communicate between heterogeneous devices transparently to the application programmer, minimize memory transfers and provide generic work division operations maintaining algorithm and data consistency;
- an application programming interface: an intuitive interface for job description, association of data and computational kernel, data access and program flow control;
- scheduling mechanisms: light scheduler able to trade-off scheduling overhead with resource utilization, using runtime metrics to efficiently distribute workloads across the available resources;
- performance model: workload metrics for scheduling and providing to the scheduler detailed information about the task and device compute capabilities;

2.3 Heterogeneous Technologies

2.3.1 Hardware Perspective

Central processing units (CPU) are the computing base history. CPU evolution had its roots with Von Neumann architecture and marked its pace with transistor technology evolution, approximately doubled every two years as predicted by Gordon Moore [44], and increased clock frequencies. Clock frequency increase is no longer possible due to physical limitation of silicon, material in which these integrated chips are built [49]. To overcome these limitations chip manufacturers turned their efforts to architectural details, re-adjusting the use of transistors and silicon space implementing new techniques to increase overall performance.

The new techniques tried to use parallelism as a fundamental key to increase the throughput of the architecture. Parallelism denotes the ability to simultaneously execute several instructions. Designers started by proposing and implementing Instruction Level Parallelism (ILP) maximizing the functional units usage. This technique is a transparent way for a chip to perform parallel operations without any change in the application [49]. Pipelining, Superscalar techniques,Out-of-order Execution and Vector processing are the most used techniques of ILP that still are implemented in today's CPUs and some devices. Also Simultaneous Multi-Threading (SMT) - that enables data access to be overlapped with computation using

switching context between threads - is a popular technique to achieve parallelism.

With all new approaches and features, adjusting and re-organizing the different units in a single chip reached a point where cost overcame the benefits. Driven by the need of more parallelism, manufacturers decided to sacrifice parallelism transparency and increase the number of execution units connecting them using an interconnecting bus. Later this technique known as Symmetrical Multiprocessing (SMP) evolved to a very familiar and broadly used solution known as Chip Multiprocessing (CMP). This approach places the different CPUs in a single chip grouping the execution units in cores. From this technique emerged the multi-core era.

The heavy demand for computing led to alternative chip designs serving specific application domains. These devices entitled Co-processors enable computation off-loading from the main processor and also try to maximize throughput instead of minimizing latency. Typically this co-processors sacrifice logic control, which simplifies the core complexity and allows the number of cores to increase substantially. A co-processor is used to complement the functions of the CPU like floating point arithmetic, graphics, signal processing, etc.

IBM Cell-BE (CBEA) is a well known co-processing architecture in the HPC community [35, 12]. It equips the supercomputer Roadrunner that hold its place in the Top500 supercomputer but it is also explored by the consumer market through the video game console PlayStation 3 produced by Sony. The field-programmable gate array (FPGA) is a flexible device that can be programmed after manufacturing [67]. Instead of being restricted to any specific domain or hardware architecture, an FPGA allows the developer to program product features and functions and reconfigure hardware for specific applications even after the product is bought and installed. FPGAs are used by engineers in the design of specific-domain integrated circuits enabling them to tailor microprocessors to meet the domain specific needs.

Graphical Processor Unit (GPU) is a technology driven by computer graphics. It is a special purpose device aimed to serve the game and film industry. Typically used to perform rasterization pilelines these devices offered quite good performance but low or zero programmability to solve other complex problems. These devices ship a high core count (many-core devices) with larger bandwidth for memory access.
Due to the industry demand for more flexibility in order to implement customized shaders,

manufacturers introduced programmable units. With each new chip generation more programmable units were added and the programmability of these devices reached a point where the HPC community was capable of exploring the parallel computing power that these co-processors offer. This is an emerging technique and it is known as general purpose GPU (GPGPU) [50, 48].

GPU computing follows the computation offloading model and the developers can profit from a high level of threading with a high number of simple cores. This makes the device ideal for data parallel applications and there are already many libraries and algorithm implementations using these devices.

Current market leaders of these co-processor are NVIDIA and ATI/AMD. NVIDIA latest implementation, codename Fermi [29], features up to 512 Stream Processors (SP) grouped in Stream Multiprocessors (SM) of 32 elements. Each SM shares a L2 cache and each SP shares a L1 cache within a SM (Figure 2.1). The device has its own memory and support DMA to system memory through the PCIe communication BUS. It is a heavily threaded device that organizes threads in blocks and the block into warps of 32 threads which makes it a 32-wide Single Instruction Multiple Data (SIMD) execution model. Fermi is designed for maximal throughput and memory latency hiding achieved by managing resources efficiently with a large number (thousands) of threads and fast context switching.

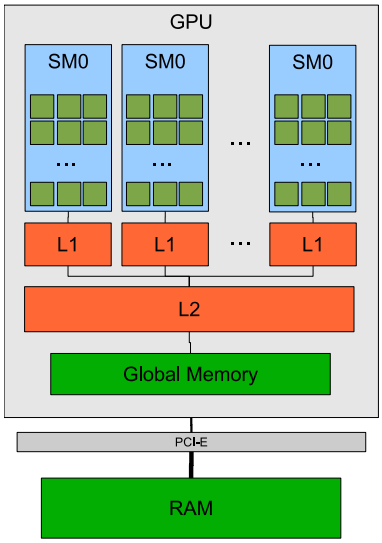


Figure 2.1: NVIDIA Fermi architecture

ATI/AMD is another well known GPU manufacturer. Their devices implement an architecture similar to NVIDIA Fermi with code name Evergreen.

Intel Larrabee [53] is an approach that tries to close the gap between CPU and GPU. This architecture will try to gather the throughput of a GPU and the programmability of a CPU in a single device. The Sandy-Bridge and Fusion platforms, from Intel and AMD respectively, also take this hybrid approach in an attempt to tackle all paradigms of parallel computing in a single chip. None of these platforms has yet been released, and little details are known, but the anticipation is high since they claim to solve some of the problems when using GPUs and CPUs.

2.3.2 Software Perspective

In order to induce more parallelism, chip designers enabled developers to explicitly use parallelism exposing multiple functional units with multi-core chips. The initial transparent parallel techniques are thus not enough, which has driven developers to re-think their applications in order to fully explore all the parallelism that the new devices were offering. Parallelizing an application may not be trivial and addresses some subtle aspects such as identifying potential sections of the algorithm that can run in parallel, data dependencies, concurrent control, etc.

A thread is a fundamental keystone of several computing paradigms and platforms. A thread runs a segment of the application instructions concurrently with other threads sharing resources such as memory. There are two major types of threads: hardware threads and software threads. The former are controlled by the hardware and aim to increase utilization of a single core (e.g. Hyper-threading technology from Intel). The latter are typically controlled by the operative system or by an API that implements a thread model.

One of the most used APIs for threads is the POSIX Thread Library (PThread) that offers thread controlling methods such as instancing, joining, synchronization, etc. However, using threads is a very low level approach to explore parallelism, since the developer needs to explicitly deal with concurrent issues, such as deadlocks and race-conditions, in order to ensure correctness. OpenMP [18] is a higher level approach that uses compiler directives to specify parallelizeable sections of code. The pragmas enable the specification of the behaviour of parallel control structures, work sharing, scope of variables and synchronization. It aims

shared memory systems and supports C, C++, and Fortran. Cilk/Cilk++ [14] is a runtime system developed by Intel for multi-thread parallel programming in C/C++. It is highly focused in reducing overhead using approaches like work and critical paths, non-blocking threads, specific thread communication and sync techniques. Uses a protocol to control work-stealing that also tries to minimize the overhead (THE protocol).

Like Cilk, TBB [32] is a multi-thread parallel library, also developed by Intel and based in C++. It provides a powerful API with parallel constructs such as parallel-for, parallel-reduce and parallel-scan along with parallel containers, memory pools, mutual exclusion mechanisms, atomic operations and task interfaces. The internal scheduler also tries to explore the cache mechanism and workload balancing. Task Parallel library (TPL) [41] is a parallel programming API incorporated in the Microsoft .NET framework. It enables .NET developers to express parallelism and make use of multiprocessing technologies. The task manager has a queuing system in order to perform efficient work distribution and work stealing among available threads.

Summarizing, PThreads offers flexibility with a low-level API, but low-level means more potential problems. OpenMP tries to abstract some parallel details but lacks a runtime and flexibility. Cilk , TBB and TPL support a high level flexibile parallel programming approach, but are bound to a shared memory system and offer no native support for co-processing.

Message Passing Interface (MPI) is the most used API for distributed memory systems (e.g. Clusters). It uses the notion of processes instead of threads where each process is, by default, associated to a core. Processes communicate through message passing supporting point-to-point and collective communication modes. Data and work must be explicitly divided among process and each process workflow is controlled independently. It is suitable for large data problems where each node will handle part of the problem. However, communication costs and low level development requiring close process control which reduces productivity.

Partitioned Global Address Space (PGAS) model [69] is a potential contribution to increase productivity. The main goal is to provide a continuous global address space over the distributed memory system. This feature ease the developers task of being aware of communication details - such as the data marshalling, synchronization, etc - allowing the developer to focus in computation.

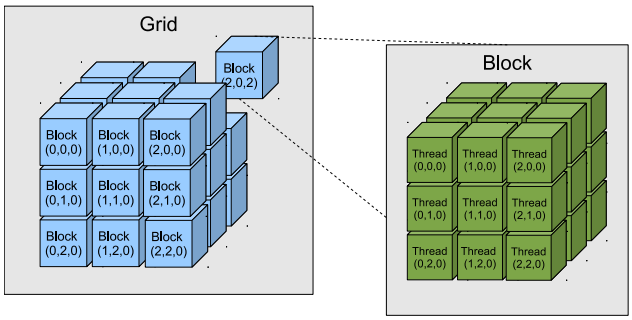


Figure 2.2: NVIDIA CUDA thread hierarchy

CHAPEL [20, 16] is an innovative language that tries to tackle data abstraction in parallel computing. CHAPEL uses a global view abstraction of data providing a high-level support for multi-threading. It combines data and distribution abstractions increasing productivity and leveraging data locality. It defines the concept of Domain, which encapsulates size and shape of data, supports domain-to-domain operations, parallel iterations, etc. A domain may be distributed across processing elements (PE) - the API provides different types of distributions and set ownerships over domains, mapping from global indices to PE local indices, etc. CHAPEL provides a powerful API with support for task parallelism and data parallelism, however it is still at an early stage and its efficiency requires more development in order to meet HPC community requirements.

In order to enable the HPC community to better explore their devices, NVIDIA and ATI provided libraries that better expose the parallelism levels of their chip architecture and execution model: Compute Unified Device Architecture (CUDA) in NVIDIA case and AMD Stream SDK for ATI/AMD chips.

The CUDA [45] programming model was designed to keep a low learning curve using familiar languages like C and C++. A CUDA application consists in parts of code that can be executed either by the host (CPU) or GPU. The separation of the code is done on compile time by the NVIDIA C Compiler (nvcc), but functions are explicitly tagged by the programmer as being intended to run either on the CPU or GPU. Kernels are C functions defined by the programmer that are executed in parallel by CUDA threads. A core abstraction of the API is the hierarchy of thread groups. A kernel call will create a grid of blocks of threads, a grid and a block can be organized in 1-,2- and 3-D abstractions and to each thread and block is associated an index as figure 2.2 illustrates. This provides a natural

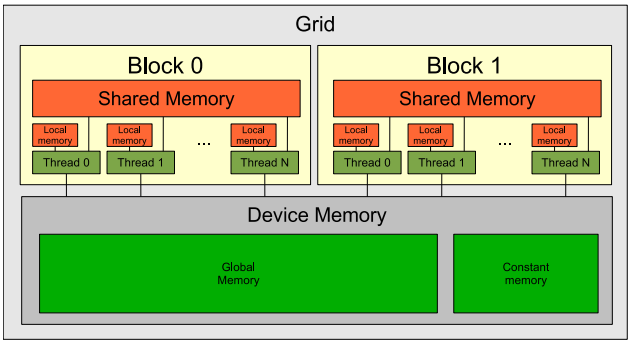


Figure 2.3: NVIDIA CUDA memory model

way to associate computation to a data vector, matrices, or volumes.

The CUDA memory model is also straightforward as Figure 2.3 illustrates. Each thread has a local memory and each thread block has a shared memory accessible to all threads in block scope. All threads have access to device global and constant memory that are persistent across kernels launches.

As mentioned in the hardware section CPU and GPU have disjoint address spaces which require explicit data copy. These copies can be synchronous or asynchronous. CUDA is now at version 4 and ships development tools, libraries, documentation and several code samples.

To explore the AMD/ATI GPUs developers can use the AMD Stream SDK API [13]. Compute Abstraction Layer (CAL) is a device driver interface to interact with the GPU stream cores using an approach similar to CUDA.

In a heterogeneous system (HS) each device may have its own programming language and development tools which affect productivity. In 2008, the Khronos Group gathered CPU, GPU, embedded-processor, and software companies and agreed - following a Apple proposal - to develop a single API that was able to support heterogeneous systems. After five months tailoring the API, the team presented the specification for the Open Computing Language (OpenCL) 1.0. The latest OpenCL support list includes Nvidia GPUs, ATI/AMD GPUs, Intel CPUs, AMD CPUs and IBM Power PCs.

OpenCL [36] is based on C and enables the user to write kernels that execute in OpenCL capable devices. OpenCL execution model and programming model are similar to CUDA. Besides data-parallelism, this framework also enables task-parallelism. The API execution is coordinated with command-queues supporting out-of-order execution and synchronizations primitives. Thus, OpenCL proposes a new standard that works across several devices, fa-

miliar to developers and enables the developer to explore HS. However, it will only achieve its goal when manufacturers release the OpenCL tool stack in full compliance to the standard.

RapidMind Multi-core Development Platform (RMDP) [43] is another tool for expressing parallelism in heterogeneous platforms. RapidMind is a data-parallelism platform also using the concept of kernel over arrays of data in high level C++ hardware agnostic language. In 2009, Rapidmind team joined Intel and are currently developing Intel Array Building Blocks (ArBB) [34]. Their objective is to provide a programming model that enables the developers to parallelize applications while hiding details of the underlying hardware. ArBB seems to be a promising data-parallel platform but uses a code annotation approach similar to OpenMP and is still in a beta version.

Romain Dolbeau et al. developed a Hybrid Multi-core Parallel Programming Environment (HMPP) that proposes a solution to simplify the use of hardware acceleration for general purposes computations [24]. The toolkit includes a set of compiler directives and a runtime system that offers the programmer simplicity, flexibility and portability to distribute application computations over available heterogeneous devices. The approach also uses directives to label a C method as a codelet. The integration of specific device code is done using dynamic linking, making the application ready for receiving a new device or an improved codelet. The execution time of the codelet is explicit, and when invoked the runtime chooses the first available compatible device.

2.4 Related work

Addressing heterogeneous systems efficiently and productively requires that data movements should be transparent to the programmer. This will not only ensure correctness and simplify the development but also should try to minimize data transfers since they represent a potential performance bottleneck.
This has already been widely addressed in the literature, COMIC [39] is a runtime system for Cell that proposes a single shared address space with a relaxed memory consistency model with a lazy approach[1]. GMAC [27] is a very similar approach for GPUs and several other

[1] memory transfer actions are postpone to the next synchronizing point in order to reduce memory transfer overhead

frameworks try to ease this burden [30, 65]. All simplify data access but they do not address workload distribution.

CellSs [11] and GPUSs[8] are implementations of a super-scalar programming model for Cells and GPUs respectively. The model uses code annotations and implements data caching mechanism following a task-parallel approach. They use a task graph dependencies to explore parallelism, overlap communication and computation and dispatch in a centralized fashion.

Sequoia [25] is another framework developed for supercomputers embedded with co-processors also proposing a unified memory space featuring high-level abstractions and techniques. It uses a user-defined task tree were tasks call the lower level tasks, ending on a leaf where the computation is done. Each task has its own private address space and can only reference this space. This way data locality is enforced and by providing a task and division API, Sequoia run-time can map the application in different architectures. However, data and computation mapping are static, lacking flexibility and adaptation.

StarPU [6] is a project that also addresses heterogeneous computing. They propose an unified run-time system with pre-fetching and coherence mechanisms based on the Modified/Shared/Invalid cache protocol which enable relaxed consistency and data replication. The data management also cooperates with the scheduler providing useful information about data location [5]. Combining this with a data transfer overhead prediction, StarPU schedulers are able to predict the cost of moving data and influence their scheduling decisions in order to maximize throughput.

Since the focus of this work lays on scheduling rather than data management, the proposed DMS in this dissertation is based on the StarPU approach. It uses the same cache protocol with lazy consistency and keeps the programmer agnostic to data movements.

The key challenge in heterogeneous platforms is scheduling. The target is to efficiently map the application onto available resources in order to increase the throughput and minimize the execution time. Task scheduling is an NP-complete [26, 66] problem and has been extensively studied with several heuristics proposed. These heuristics can be categorized as list-scheduling algorithms [1, 10], clustering algorithms [28], duplication-based algorithms[2], genetic algorithms [56] ,among others. However, these policies target homogeneous distributed memory systems, scheduling in heterogeneous systems is more challenging. Literature has also addressed these systems [55, 52, 23, 9, 63], but heterogeneity is reduced to different computational powers where the processing elements have identical architectures and programming paradigms.

Designing scheduling strategies for systems that do not share the same computational model is more complex, since more factors come into play, as stated in section 2.2.

Harmony [22], Merge [42] and StarPU [6] are three similar run-time systems that address heterogeneous architecture platforms. They all provide a API to express job and data dependencies, a single address space and different scheduling policies.

Harmony programming model follows a simple approach with three main mechanisms: compute kernels, control decisions and a shared address space. Compute kernels are similar to function calls and represent the algorithm functionality with the following restrictions: (i) pointers are not allowed; (ii) each call can only use one system processing unit and it is exclusive; (iii) kernels may have different implementations according to the architecture, but need to produce the same result for the same input; (iv) temporary kernel local data is not persistent. Control Decision enables the programmer to express dependencies which also enable the runtime system to transparently optimize the work flow with speculation and branch prediction. Finally, the shared address space enables the runtime to manage data according to kernel mapping needs. The execution model is compared by the authors to a modern processor with super-scalar and out-of-order techniques.

Merge, proposed by Collins et al., is another heterogeneous multi-core computing framework that uses EXOCHI [68] as the low level interface to tackle portability and provides a high-level programming language with a run-time system and compiler. EXOCHI is a twofold API: (i) Exoskeleton Sequencer (EXO) is an architecture that represents heterogeneous devices as ISA-based MIMD processors combined with execution model, endued with a shared virtual memory space; (ii) C for Heterogeneous Integration (CHI) is the C/C++ API that supports specific in-line assembly from specific devices and domain-specific languages. The framework uses the MapReduce [19] pattern that enables load balancing between processors and automatic and transparent parallelization of the code. The authors advocate that Merge is applicable to many HS and applications are easily extensible and can easily target new architectures.

Augonnet et al. designed StarPU [5, 61, 6], that provides to HMS programmers a run-time system to implement parallel applications over heterogeneous hardware that also includes mechanisms to develop portable scheduling algorithms. StarPU provides an unified

execution model combined with a virtual shared memory and a performance model working together with dynamic scheduling policies. The two basic principles that droved the execution model were: (i) tasks can have several implementations according to each device architecture available in the system (codelets represent platform specific implementations of a task functionality); (ii) data transfers inherent from the use of multiple devices are handled transparently by the run-time, which originated the above mentioned DMS. StarPU DMS is able to track data locations in order to reduce the number of copies, it also uses relaxed consistency and data replication guaranteed by the MSI protocol. It also explores asynchronous data transfers and data movement and computation overlapping. The framework also provides abstractions to divide data called filters. Filters are used logically to divide data into blocks and can be used dynamically in runtime for computation refinement purposes. Authors advocate that efficient scheduling in HMS and overall performance improvement can only be achieved by a data locality and granularity aware scheduler. StarPU is able to take scheduling decisions according to data location information , provided by the DMS, together with transfer cost prediction.

The scheduler implementations proposed by Augonnet et al. use greedy approaches - all processor share a single queue of tasks with and without task prioritization defined by the user - and cost-guided strategies where mapping of a task follows a performance model. Two type of performance models are proposed: (i) based on device compute capabilities; (ii) based on the Heterogeneous Earliest Finish Time (HEFT) [63] algorithm. In the latter, the cost is predicted using history-based strategies that use hash tables to keep track of tasks' execution time.

In the literature efforts can be found to map irregular applications in typical data-parallel devices like the GPU, balancing computation load across the device available resources. [3] introduced the notion of persistent kernel motivated by the severe variation in execution time of a ray traversal in a ray tracer. The idea is to launch only enough CUDA threads to fill the device resources, support them with work queues and keep the threads alive while there is work to process in the queues. These concepts were implemented and evaluated by Tzeng et .al in [64], that also introduced warp size work granularity and enhanced the notion of persistent threads combining it with the uberkernel programming model[60] and individual work queues.

2.5 Conclusions

Heterogeneous systems are widely available and some frameworks were proposed to deal with the heterogeneity that these systems exhibit. It is arguable however that the development productivity and full utilization of all resources of these systems have not reached the their peak, specially with different computational load patterns.

Several proposals to tackle the identified difficulties were made with different approaches. All presented a shared address space; however, some of them lack an expressive API to submit data to be dealt by the run-time system [22]. Also, all proposed different approaches to scheduling the computation across the different devices (some of them with explicit work division [42]), lacking of a performance model to reason about workload and device capabilities [68, 42].

Using SIMT devices (like GPUs) to handle irregular workloads is not trivial and might require overtaking the device internal scheduler, as proposed by [60, 64]. This item will be addressed on section 3.4.

Augonnet et al. [6] identified similar difficulties and addressed them with an unified programming model endued with a DMS and some scheduling approaches for load balancing across available resources. The approach proposed in this dissertation will follow a similar model for data management, but will resort to different scheduling approaches that try to tackle the challenges of efficiently mapping regular and irregular applications across heterogeneous devices.

Chapter 3

PERFORM runtime system

A solution for efficient and productive use of HMS entails tackling a series of challenges that are consequence of the heterogeneity of these systems. This dissertation proposes an execution and programming model that tackles heterogeneous environments providing mechanisms that meet the identified requirements. This chapter discusses these mechanisms implemented in a framework entitled PERFORM.

3.1 Programming and Execution model

The proposed execution model perceives applications as one or more **computation kernels** that apply some computation to all elements of a given data set (Figure 3.1). In this sense, the execution of one computation kernel is a data parallel problem and the **basic work unit** is the application of the kernel to one data element. Note, however, that the

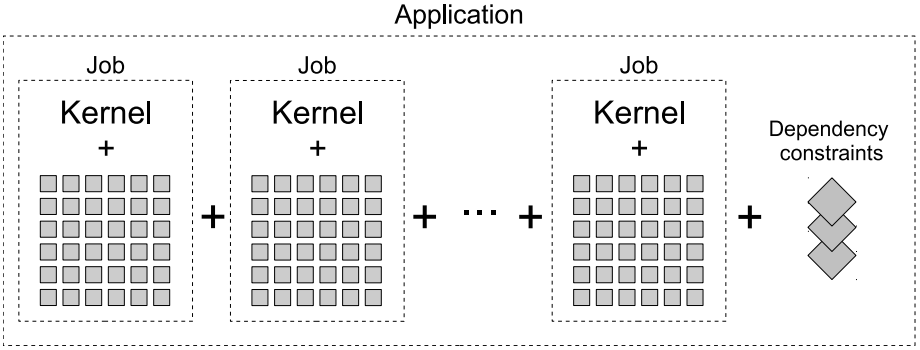


Figure 3.1: Execution Model - Application definition

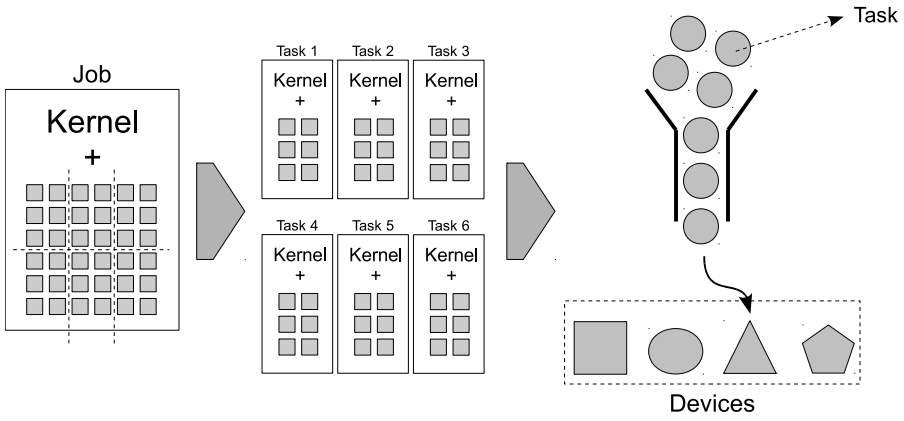


Figure 3.2: Execution Model - Job partitioning into Tasks

workload might be different across the various work units due to the irregular nature of the algorithm and associated data. Dependency constraints among different kernels are expressed using system primitives i.e., must be explicitly coded by the application programmer.

A **Job** consists on applying a computation kernel to a data set. It is the runtime system responsibility to partition the data set into blocks of basic work units, referred to as **Tasks**, and to dispatch the execution of these tasks onto available devices (Figure 3.2). The actual mapping of the tasks onto devices is completely transparent to the programmer and handled by the runtime system. Tasks are executed **out-of-order** i.e., once a job is submitted, the runtime system will partition the data set and dispatch the tasks with no execution order guarantees. On the other hand, job execution order will respect specified dependency constraints; if no dependency is specified, two jobs may execute concurrently.

Each kernel implementation is agnostic to the basic work item block (task) size as well as data accesses that are done inside the kernel. It is the application programmer responsibility to provide a method which performs data set partitioning on system demand (a method referred to as **Dice**) and to provide implementations of the kernel targeted and optimized to each device architecture. The dice method renders the runtime system agnostic to algorithm data decomposition. However, current implementation requires data to be represented as multidimensional continuous arrays.

A **unified address space** is provided that keeps the programmer agnostic to data movements and location among the disjoint address spaces. However, the programmer has to explicitly gather back the partitioned data set calling a system primitive i.e., it is program-

mer responsibility to invoke the gather of the specific data set of a job before its submission or access. Furthermore, the runtime system does not ensure data consistency of concurrent jobs, i.e., if two jobs write or update the same data, the serialization of the job must be assured by the programmer using system primitives.

The runtime system follows a host-device model similar to CUDA and OpenCL APIs. The system is composed by a host - CPU - that dispatches work to available devices - CPU, GPU, Cell, etc. Once a task is submitted to a device its execution follows the device specific programming and execution model as the programmer coded it. The runtime system is unaware of the device behaviour following a monolithic approach, i.e., the runtime system assigns the task, the device computes it and produces the result that will be gathered by the system/programmer.

3.2 Programming Interface

A proposed goal of this dissertation is to provide an API for heterogeneous systems that enables the programmer to express applications intuitively. Dealing with the different APIs of each device can be very tedious in a heterogeneous system. Moreover, the disjoint address spaces are both tedious to handle and error-prone. An API is thus proposed that will enable the programmer to intuitively code applications in PERFORM. The API provides primitives for **job creation**, **data** and **kernel association**, **control** and data access; all implemented using the C++ programming language.

Recall that a job consists on applying a computation kernel to a data set and a task is a partition of a data set to which the same kernel will be applied. **Jobs** are explicitly created using C++ object inheritance features. The programmer provides to the system an implementation of an extension of the class Job that is provided by the system and adds as variables the potential parameters required to perform the job. The system will further use polymorphism features to handle Job objects. Although semantically different, jobs and tasks are implemented using the same C++ class.

The **Dice** method is a feature that enables the runtime system to partition data on demand i.e., at any given time the system calls this method with a parameter and it returns a set of new tasks. The parameter provides an hint of the granularity of the new intended tasks. It is the method responsibility to generate new tasks and associated data respecting

```
1  GEMM() {
2     float *matrixA = (float*) (malloc(sizeof(float) * N * N));
3     float *matrixB = (float*) (malloc(sizeof(float) * N * N));
4     float *matrixC = (float*) (malloc(sizeof(float) * N * N));
5     Domain<float>* A = new Domain<float> (2, R, matrixA, dim_space(0, N), dim_space(0, N));
6     Domain<float>* B = new Domain<float> (2, R, matrixB, dim_space(0, N), dim_space(0, N));
7     Domain<float>* C = new Domain<float> (2, RW, matrixC, dim_space(0, N), dim_space(0, N));
8
9     Job_MM *t = new Job_MM();
10    t->associate_domain(A);
11    t->associate_domain(B);
12    t->associate_domain(C);
13    t->associate_kernel(CPU, &k_CPU_GEMM);
14    t->associate_kernel(GPU, &k_CUDA_GEMM);
15    performQ->addJob(t);
16    wait_for_all_jobs();
17    performDL->splice(C);
18 }
```

Code block 3.1: Matrix Multiplication Job

the input parameter, which is defined using a performance model that will be discussed later. The actual data partition is thus twofold: (i) it is performed on execution time by the programmer supplied dice method and upon system demand; (ii) it is application dependent since it is the application programmer who designs it. This is due to the fact that each algorithm or problem has its own way to divide data, and the programmer is the only one that is aware of such division patterns.

Although physical data partition and scatter is transparent to the programmer, the system is not aware of which data needs to be gathered back to the host. Since gathering all data assigned to a job is potentially wasteful, the programmer is provided with a method, named **Splice**, that gathers all the data to the host. If the splice method is not called, there is no guarantee of host data consistency after a job or task completion.

Abstract data handling is hard since there is an arbitrary number of ways to represent data. Applications can use multidimensional arrays, pointer based structures, language API built-in structures, among others. This flexibility hinders the dynamic manipulation of data by the runtime system, and since abstract data manipulation is not the focus of this dissertation, current implementation only supports **multidimensional continuous arrays**. Most data representations can be converted to continuous arrays. The continuity enables the system to move and copy data without any intervention from the programmer.

Data access and registration is done using the DMS object **Domain**, whose implementation will be discussed in DMS section. The order in which the registration is done is important. An index is associated to each registration and it will be used to access the set

```
class Job_MM: public Task {

   //no parameters needed

   dice(Task**& new_tasks_,PM_Metric_normalized param) {
      Domain<float>* A,B,C;
      getDomain(0, A);
      getDomain(1, B);
      getDomain(2, C);
      int divide_task_by = // calculate ho many tasks will be created according to param

      Domain<float>* subA[divide_task_by];
      Domain<float>* subB[divide_task_by];

      for (int i = 0; i < divide_task_by; i++) {
         //creation of subdomains of A and B
      }
      for (int i = 0; i < divide_task_by; i++) {
         for (int j = 0; j < divide_task_by; j++) {
            Job_MM* t = new Job_MM();
            //creation of subdomains of C
            t->associate_domain(subA[i]);
            t->associate_domain(subB[j]);
            t->associate_domain(subdomainC);
            new_tasks[task_count++] = t;
         }
      }
   }
}
```

Code block 3.2: Matrix Multiplication Job class definition

of the registered domains to that job. Domains encapsulate the pointer to the data buffer, the dimensions, the type of data - using C++ template features - and the read/write permissions. The framework also provides sub-domains that enable a data partition hierarchy.

Dependency constraints are typically expressed using a Directed Acyclic Graph (DAG) [10, 9, 28, 4, 6, 11], however for the scope of this dissertation a **simple job barrier primitive** suits the application and runtime needs. This primitive will block job submission until all jobs and tasks submitted until that point are completed. This can also be used by the programmer to ensure that data is ready for gathering since all jobs are completed and data

```
void k_CPU_GEMM(Task* t_) {

   Domain<float> A;
   t_->getDomain(0, A);

   Domain<float> B;
   t_->getDomain(1, B);

   Domain<float> C;
   t_->getDomain(2, C);

   for (int i = C.X.start; i < C.X.end; i++) {
      for (int j = C.Y.start; j < C.Y.end; j++) {
            float acc = 0;
            for (int k = A.Y.start; k < A.Y.end; k++) {
               acc += A.at(i, k) * B.at(k, j);
            }
            C.at(i, j) = acc;
         }
      }
}
```

Code block 3.3: Matrix Multiplication CPU kernel

is ready for gathering.

The code block 3.1 illustrates the code required for a matrix multiplication example. The domains and kernels are associated using a Job object which is submitted to system through the object *performQ* that represents the system job queue. Splice method belongs to DMS, instantiated in the *performDL* object, and it is called after the dependency primitive *wait_for_all_jobs()* returns. The code block 3.2 illustrates the Job class definition where the dice method is overridden. It creates the sub-domains and tasks according to a parameter and stores them in a buffer, retrieving their location to the runtime system for further processing. In code block 3.3 is an example of a simple matrix multiplication CPU kernel that illustrate how agnostic data access is performed.

3.3 Framework design

This section presents an overview of the framework architecture, detailing how the features collaborate in order to tackle the previously identified challenges.

Figure 3.3 gives an overview of process flow from the job submission to the retrieval of the final results. An application is expressed by a set of jobs, each one constituted by a kernel and a data set. These jobs will be submitted to the system using the previously described API. First the Domains will be registered in the DMS and then the job are submitted to the job queue.

Each device is controlled by a hardware driver API implemented by the manufacturer that abstracts the low level interface to deal with the device, e.g., NVIDIA CUDA driver. This API basically provides access to data transfers and functionality programming and execution, is specific to each device and non-portable. To overcome this non-portability the runtime system uses a higher level API for each device that follows a common interface. This API, named *DeviceAPI*, abstracts all the manufacture primitives required to use a specific device. Each device registered in PERFORM runtime has an instance of this API which associates common interface methods to manufacturer methods.

Furthermore, each registered device is associated a thread that uses the DeviceAPI to control the device. This enables each computational resource to run as an individual independent worker that cooperates with the remaining entities of the system concurrently. This controller uses DMS methods to manage data transfers and Domains ensuring data consistency. It also uses a local a task queue for local work management and task buffer for

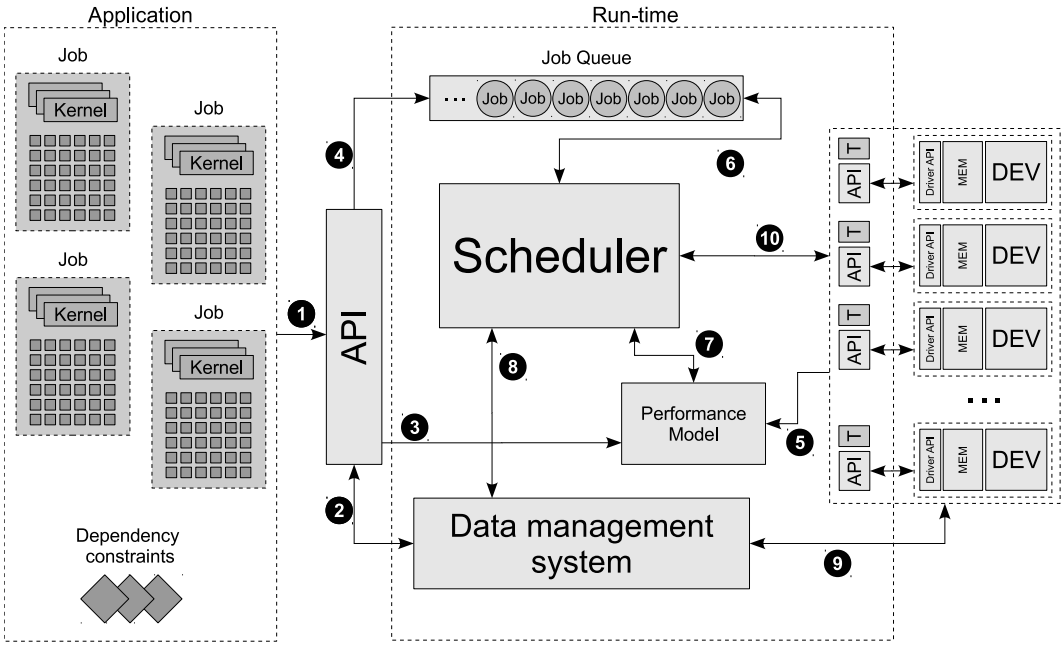


Figure 3.3: (1) Jobs defined by kernels and data are submitted to the system using the API as well as application dependency constraints; (2) the API will register data in DMS and gather(splice) data back; (3) a workload factor is assigned to the job using a performance model; (4) jobs are enqueued in a main job queue for execution; (5) registers the performance model evaluate the compute capabilities of each device; (6) the scheduler dequeues and enqueues jobs or tasks from the main queue; (7) the scheduler tries to assign a job to a device, reasoning about job workload and device compute capabilities that potential trigger dice features; (8) the scheduler signal data movements required for the job to the assigned device; (9) DMS signals data transfers from/to device memory address space; (10) Signals for execution, and job completion

the scheduler to place assigned tasks.

The Scheduler will be responsible for task partitioning and task assignment to devices. The assignment settles over a scheduling policy that is combined with a partitioning policy and a performance model. There is also a thread created and assigned for these operations that is always ready for task dispatching according to the scheduling policy.

All threads and entities in the system operate asynchronously and communicate using *System Operations*, which similar to the message passing communication protocol.

3.3.1 Data management

As stated, due to the disjoint address spaces and different APIs, a DMS is a crucial feature when addressing HMS. Although the focus of this dissertation is not data management, tackling a heterogeneous system requires a system capable of transparently handle data. The [5] DMS features contemplate the basic needs of this dissertation and a similar DMS is proposed with some differences in the interface and data partitioning.

Two main concepts are used, Domains and Data chunks. A **Domain** is a logical representation of a data region,which encapsulates dimension [1], size and type. Size is expressed using ranges for each dimension (for instance to create a Domain for a matrix the programmer specify to the Domain constructor four values that represent two ranges of values, one for the rows and another for the columns). To express data partitioning the programmer can use Subdomains. A **Subdomain** is also a Domain that represents a smaller region of the data set. Its range is relative to the root Domain of the hierarchy. Figure 3.4 shows an example of a two level partition, where Domains and Subdomains instantiations are illustrated.

A **Data chunk** (DC) is a system object that represents physical data. When a task is submitted to a device, the device controller will instantiate - using DMS methods - a DC for each domain associated to the task. When all data chunks are created, the controller uses the DeviceAPI to copy the data to the device address space and signals task execution. The memory address that results from the copy is stored in the DC object along with other addresses from other devices for further data accessing.
These two concepts simplify data management since they separate the actual data copies and location from logical data divisions and length definitions. They also enable the programmer supplied kernels to be agnostic to each range of data available in each execution, being the Domain responsibility to calculate the correct address pointer from global indexes to task

[1] recall that only multidimensional arrays are supported

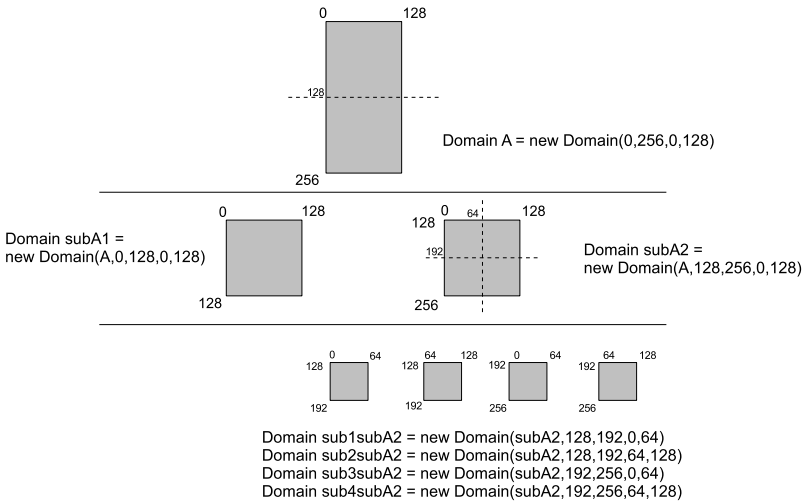


Figure 3.4: Domain hierarchy ranges; The Subdomain ranges are always relative to the root domain A

DC local indexes.

New DC creation and data gather is performed using two DMS methods. To perform a partition a new chunk will be created according to a Domain S. The algorithm will search in the higher levels of the hierarchy which parent Domain P of S has a DC associated. When found the method will allocate a DC respecting S sizes and copy data from Domain P Data chunk to this new DC. When a DC is created according to a Domain it is associated to that Domain. This mechanism can be seen as establishing a domain private address space.

The gather method, called **Splice**, does the inverse. It copies back all the chunks to original positions according to the domain hierarchy. It is a recursive method that traverses the whole tree, when it finds a non-divided Domain it will copy its DC to the eldest Domain with a DC.

As proposed by Augonnet et al.[6], the core of the proposed DMS is a DC registry table based on a MSI (Modified/Shared/Invalid) cache coherence protocol. This table registers all DCs movements according to R/W request types that influence the state of the chunk. According to the MSI protocol the state of a DC is either (i) Modified, when a chunk has been requested for writing and the computation is still going, (ii) Shared, when a chunk is copied and no writing requests have been made and (iii) Invalid, assigned to all copies of a chunk that has been requested for writing. In Figure 3.5 is an example of an assignment

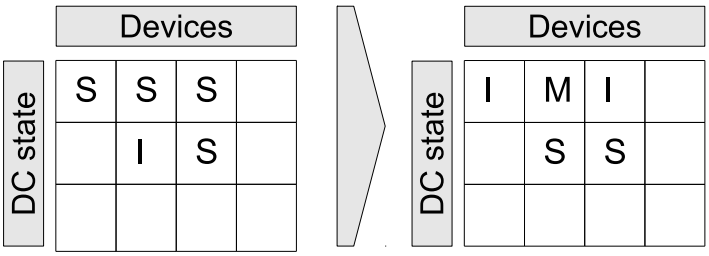


Figure 3.5: Device 0,1 and 2 have a valid copy a DC 0, device 1 has a invalid copy and device 2 has a valid copy of DC 1. If a task, requesting DC 0 to write and DC 1 to read, is assign to device 1 the DMS will: (i) copy DC 1 to device 1, marking it has shared; (ii) mark DC 0 in device 1 as modified; (iii) and declare all the other DC 0 copies invalid because it was requested for writing.

of a task with two DCs to a device. Device 0, 1 and 2 have a valid copy a DC 0, device 1 has a invalid copy of DC 1 while device 2 has a valid one. If a task, requesting DC 0 to write and DC 1 to read, is assigned to device 1 the DMS will: (i) copy DC 1 to device 1, marking it has shared; (ii) mark DC 0 in device 1 as modified; (iii) and declare all the other DC 0 copies invalid because it was requested for writing. This feature not only ensures consistency, but also enables data replication that combined with a lazy approach may potentially reduce data transfers. After a task completion, the chunks that were requested remain in the devices (lazy) and are marked as shared (if the state was modified, otherwise it is already in shared state) in the MSI table. For the next task that requests a chunk, the DMS will check if the device has a valid (shared) copy and if so the data transfer is suppressed.

Furthermore, to overcome limitations in bus bandwidth some devices support asynchronous data transfers and computation overlapping (e.g. NVIDIA GPU CUDA API with Streams support). As previously detailed each device has a DeviceAPI and a controlling thread associated. If the device supports concurrent copy and execution, the controller will use a two task window execution, i.e., right after signalling the task execution, the controller requests another task, processes its data chunks and signals the necessary copies while the device is executing the previous task. In the ideal case, when a task is signalled for execution, the necessary data associated to that task would be ready in device memory address space. This data pre-fetching technique reduces the data transfer overhead enabling, in an optimal case, a task to be ready for execution right after the previous task finishes. This will potentially increase the device throughput and consequently increase system performance.

Due to the lazy memory approach devices may end up with useless data in their memory

spaces that is required by the following tasks. If the memory is not enough to execute a given task, the DMS will perform a flush of specific shared data chunks (copying them to host memory), releasing useless data from device memory. This is done adding new tags to the MSI protocol and device memory available checks.

3.3.2 Scheduling

Scheduling defines how the work is distributed over the devices and considers several factors that influence the decision. This dissertation evaluates and proposes four different schedulers that can be divided in two categories: (i) static, where all the enqueued tasks are assigned to the devices at once; (ii) demand driven, where tasks are assigned upon device request. Each scheduler has two stages, a partition stage and a mapping stage that are enhanced from scheduler to scheduler.

Round Robin with dummy dice (RR_DUM)

Basic static scheduling approach that partitions tasks in equal chunks according to a static predefined value and assigns tasks to devices in a round robin fashion. In the first stage it dequeues a job from the main queue, if it is partitionable and has not been partitioned, the scheduler calls the dice method with the parameter corresponding to a default static value (dummy dice).

In the second stage the scheduler will select in which device the task will be mapped to. Notice that the scheduler is unaware of the total number of tasks that the programmer will submit, and one of the objectives of the runtime system is to balance the load across all available devices. Round robin scheduling is simple scheduling algorithm typically used for assigning execution time slices to processes in a circular order. In this case the scheduler will use this circular order to assign tasks to devices, alternating between them and thus distributing the tasks evenly. After the target device is set, the task is enqueued in the device controller queue and the scheduler repeats the process with another task from the main queue.

Weighted Round Robin with dummy dice (WRR_DUM)

This scheduling approach is similar to the previous one, it only differs in the mapping stage. When the programmer registers a device in the framework, a performance metric will be used to characterize the device compute capabilities. This performance metric is defined according

```
loop
   check scheduler device work queue for tasks

   if no tasks found
      check main queue
   if no tasks found
      check other devices' queues
   if no tasks found
      block until tasks are submitted

   if task found
      if task_dice_level 0
         dice according to default value, buffer with N tasks returned
         enqueue N-1 tasks in main queue
         task = remaining buffer task

      if task_dice_level 1
         dice according to device relative compute capability, buffer with M tasks returned
         enqueue M-1 tasks in main queue
         task = remaining buffer task

      if task_dice_level > 1
         add task to device controller queue

end loop
```

Code block 3.4: Demand driven with dynamic dice scheduling algorithm

to the performance model implementation. This metric will be used by the scheduler to define the relative number of tasks that are assigned to each device. The circular assignment order is also used but enhanced in order to perform the weighted balanced assignment of tasks. The number of tasks assigned to each device is thus coherent with the computational power that the device provides to the system.

Demand driven with dummy dice (DD_DUM)

This scheduler follows an inverse communication pattern to assign tasks i.e., the scheduler will process a task upon a device request for work. When a device gets idle or finishes a task it sends a message (using System Operations messages) to the scheduler thread indicating that it is available for processing. The scheduler then fetches a task, applies the same first stage algorithm as the previous two approaches, and enqueues the task in the device work queue. This will automatically balance the task assignment since the mapping strategy respects each device requests.

Demand driven with dynamic dice (DD_DYN)

DD_DYN follows a slight different approach. It also follows a demand driven mapping but it is combined with a two level dice stage. Upon device work request, the scheduler fetches a task and checks its dice level. If task is dice level zero it calls the dice method with a parameter calculated according to a default value (just like the other schedulers), marks the resulting tasks as dice level one tasks and enqueues them in main queue. If task is dice level one, the

dice method will be called with a parameter calculated according to the requesting device relative compute capability (RCC). Notice that this is not the same value that characterizes the devices used in WRR_DUM scheduler. This a normalized relative performance metric that enables the scheduler to be aware of the gap between devices' compute capabilities. For instance, in a system with two GPUs and one CPU, if the performance model dictates that the CPU is four times slower than the GPU1 and the GPU2 is two times slower than GPU1, the CPU, GPU1 and GPU2 RCCs are 1/4, 1 and 1/2 respectively.

If the requesting device RCC is 1, the dice level one task is directly assigned to device and no second level is applied. Otherwise, task is diced according to device RCC and the resulting tasks are marked and enqueued in a queue, corresponding to the requesting device, from a set of queues that the this scheduler maintains. This set of queues, one for each device, enables the scheduler to organize different granularity tasks in a fast and low-overhead mechanism. If the task level is two it is enqueued in device controller work queue for execution.

The scheduler local queues technique also enables work-stealing. Upon device work request, if there are no more tasks on the device queue as well as on the main queue, the scheduler will search in the remaining device queues for tasks; if found one is popped and assigned to the requesting device.

The DD_DYN work-flow is illustrated in Code block 3.4. Upon device work request, the scheduler checks the device queue, the main queue and the other device queues for work, if found it applies the dice method according to the task dice level and requesting device compute capabilities. Finally, the task is enqueued in the device controller queue for execution.

3.3.3 Performance Model

When assigning task to devices the scheduler must take appropriate decisions. These decisions are influenced by several factors that need to be considered in order to achieve an efficient scheduling. These factors might include device characteristics, algorithm and workload size. The performance model purpose is to encapsulate all this information and provide useful hints to the scheduler in order to achieve efficient scheduling.

As Thibault et al.[61] argued, it is very hard to find a single performance model that provides precise and reliable information in a generic way. It needs to capture and combine details about device architecture, device execution and memory models, data workload patterns, data set size, etc.

Nevertheless, a performance model can be defined over a set of parameters that provide

estimations of performance behaviours. For this dissertation scope a simple performance model is proposed that enables the characterization of the device capabilities. This performance model is based on the theoretical peak floating-point operations per second of the device: it represents the theoretical number of operations a device can perform per second. This value is provided by the device manufacturer and associated to the device. It enables the scheduler to estimate the gap in performance among devices. It is a very incomplete performance model, but its implementation offers enough modularity and abstraction that enables the enhancement and implementation of other more complete performance models.

3.4 Addressing irregularity

One of the great causes of inefficiency when exploring heterogeneous systems is the irregular computation pattern of the applications. The application behaviour is not known a priori, the workload and memory accesses pattern vary from data element to data element, which difficults the efficient mapping of the workload to the available devices. For instance, in an n-body simulation with the irregular algorithm Barnes-Hut (described in Section 4.1.3), the time required to process a particle is arbitrary, thus a device takes an arbitrary amount of time to process a set of particles, which potentially causes system imbalance. This suggests the use of adaptive load distribution mechanisms, like work stealing and work donation, that enable the runtime system to balance load across devices, which also requires breaking the monolithic approach featured by the proposed model.

Another required feature is secondary work generation. In an irregular application this would require the basic work unit to be further decomposed into finer granularity tasks in order to enable the balancing. For example, in the Barnes-Hut irregular algorithm, one could associate the basic work unit (BWU) to a complete processing of the particle interaction forces, which requires a traversal of an octree of an arbitrary length. A possible further decomposition is to associate the BWU to the visit of a single level of the tree and if the algorithm requires descending in the tree a new BWU is created and stored for further local processing or mapping to another device. This will decompose the arbitrary workload into smaller chunks of load that will be balanced across available computational resources, and therefore tackling the irregularity of the application. This requires not only that the system handles distinct work granularities, but also that the application programmer rethinks the BWU associated to the algorithm and re-design the provided kernels in order to use sec-

ondary work instantiation.

Implementing these mechanisms requires the definition of a pipeline in order to manage the generated work. In single instruction multiple thread (SIMT) devices like GPUs this is not trivial due to several control limitations that these devices exhibit. In literature efforts have been made in order to map irregular workloads onto SIMT devices like GPUs [64]. It implements a task-based irregular workload model that combines a uberkernel with persistent threads, warp size work granularity and task stealing and donation. The uberkernel is a regular CUDA kernel that executes different work according to a condition where the main objective is to eliminate kernel switching overhead. Persistent threads fetch work in a loop during kernel execution until no more work is available using work queues. When a new BWU is generated it is added onto the work queue. Warp size work reduces the CUDA thread architecture to 32 threads per block assigning a block to each SM, reducing synchronization and providing a worker view of a SM.

Mechanisms to balance the workload of an irregular problem between devices and within the device using a similar task-based irregular workload model proposed by Tzeng et al.[64] were developed and assessed through out this dissertation's work. With this pipeline approach the SIMT internal device computational resources are balanced and also provides the ability to donate work to other devices balancing the arbitrary workload.

However, work donation requires communication between devices while tasks are being executed, i.e., while a persistent kernel is executing. In CUDA programming model this feature requires the use of complex communication mechanisms so that the kernel is not interrupted. Therefore, the proposed approach will focus in intra-device load balancing,i.e., no work will migrate from device to device, it will only be balanced within the device compute resources.

The proposed pipeline and flow control is illustrated in Figure 3.6. It follows the same work queueing model, using try lock mechanisms in order to reduce the queueing contention overhead, warp size work granularity and persistent threads. The application functionality is applied by a device function pointer provided by the programmer that shares work queues and data buffers.

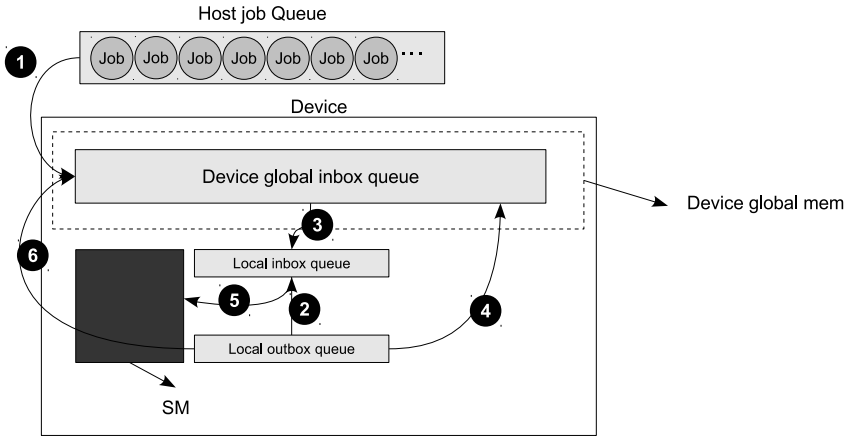


Figure 3.6: (1) A job or task, that potentially represents a range of particles, is copied to device global inbox queue(GIQ); (2) Enqueue in local inbox queue (LIQ) elements from local outbox queue (LOQ), as a result from the previous iteration, until LIQ is full or LOQ is empty; (3) Check if the LIQ is not full, if slots available try lock GIQ and get more work until LIQ is full; (4) If the LIQ is full and there is still elements in LOQ, try lock to enqueue in GIQ (5) Retrieve 32 (warp size work granularity) elements and execute; (6) If there is not enough room in LOQ to store all secondary tasks, force GIQ lock and enqueue all the elements of the LOQ; repeat from step 2.

However, the implementation of this pipeline is still not trivial. The work management overhead must be minimized so that it does not cancel the gain achieved from these mechanisms, specially if the algorithm imbalance levels are low. This overhead is characterized by the queue management and contention, that has been reduced with the try lock mechanisms, but essentially by the memory allocation overhead required to instantiate the secondary work. This suggest the recycling of memory by the means, for example, of a memory pool.

This approach was implemented and evaluated with Barnes-Hut algorithm but the overhead exhibited by the current implementation overwhelms the gains achieved by using these mechanisms. The low unbalance levels of this algorithm and the current overhead does not compensate the use of this model when compared to simply assigning one CUDA thread to a particle and process the complete traversal.

Given the very low performance levels achieved with current implementation results are not presented on this dissertation and this topic is relegated to future work.

Chapter 4

Study cases and Methodology

To evaluate the proposed models and mechanisms a small set of applications were implemented in PERFORM framework. These applications try to evidence the impact of the implemented features as well as providing some insights of behaviours in terms of scalability and efficiency of the heterogeneous system with PERFORM. They were chosen mainly due to their computational features, implementation complexity and research importance. The following sections briefly describe the three chosen applications: Matrix Multiplication (MM), Convolution with Fast Fourier Transform (CONV) and N-body Barnes-hut (BH).

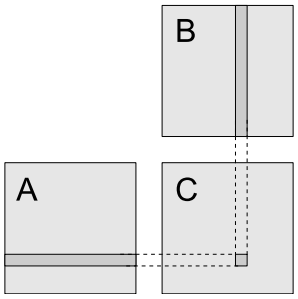


Figure 4.1: Matrix multiplication algorithm: Each element C_{ij} from the resulting matrix is calculated by a dot product between the row i of matrix A and column j of matrix B

4.1 Applications

4.1.1 Dense Matrix Multiplication

Matrix multiplication is a broadly used mathematical operation that multiplies two dense matrices and adds the result to a third matrix. The main operation is a simple dot product of two arrays, each operation result defines the value of each position in resulting matrix (Figure 4.1). The operation is described has $C \leftarrow \alpha AB + \beta C$ and has $O(n^3)$ complexity for $n \times n$ matrix. There is no data dependency between output elements and the operation is always the same, which makes it a SIMD application and a typical data-parallel program. The block decomposition of the algorithm is trivial. It is defined by dividing the matrix C in blocks where both blocks and block elements may be in computed in parallel. Therefore, according to PERFORM model, the basic work unit is an element from the resulting matrix plus the two required arrays.

The implemented version uses highly optimized libraries: CuBLAS [46] for GPU and Intel Math Kernel Library [33] for CPU. And all the tests were made with single precision floating point methods.

A Domain is created for each matrix. Input domains A and B are read-only and diced according to algorithm needs, whereas Domain C is read/write.

4.1.2 Image Convolution with the Fast Fourier Transform

Convolution is a technique used in image processing for low-level filtering. Convolution is defined in few steps, one of them is the well known Discrete Fourier Transform (DFT) which is an expensive operation with $O(n^2)$ complexity. Cooley and Tukey [17] though presented the Fast Fourier Transform (FFT), which is a faster version of this algorithm that reduces it to $O(n \log n)$.

The Fourier Transform converts a signal from the time domain to the frequency domain. It is typically used as two inverse operations: (i) forward transform, that transforms a signal to the frequency domain; (ii) inverse transform , which transforms a spectrum (frequency domain signal) into the time domain. Applying a forward transformation to a signal and applying inverse transform to the result, the initial signal is obtained.

The algorithm operates in one-dimension or in two-dimensions. When processing a image signal - typically a matrix - both 1D and 2D versions of the algorithm apply. The 1D version is applied to all rows independently and repeating the process in the columns. In order to

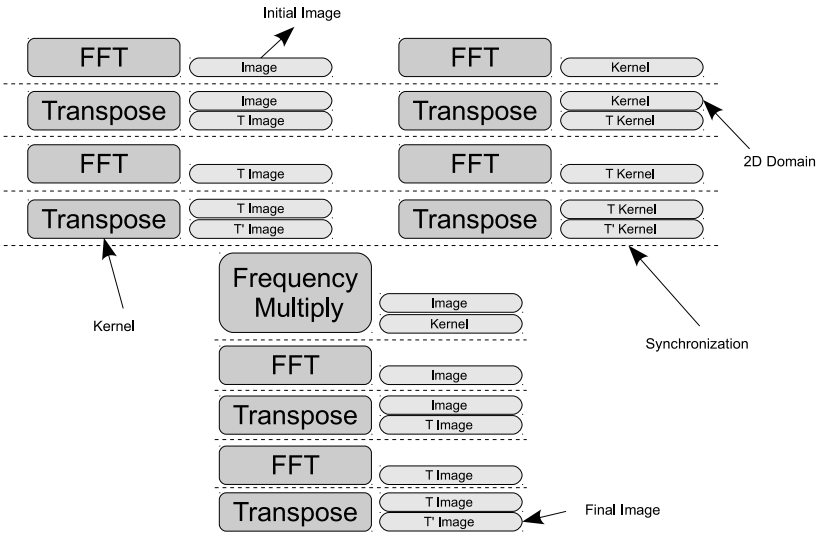


Figure 4.2: Image Convolution. Top down execution flow. Image result in T' Image domain.

apply it to columns the row result - both real and imaginary - is transposed and the method is repeated. On the other hand the 2D version processes all the signal at once. Both versions produce an image frequency spectrum that is composed by a real and imaginary part.

The Convolution is a mathematical way of combining two signals and form a third signal. The first signal corresponds to the image and the second is a filter kernel that corresponds to the intended effect. The result signal yields the final image.

The algorithm is defined in three steps: (i) FFT of the input image and filter kernel, to transform these into the frequency domain; (ii) Multiplication of the real and imaginary parts of the input image with the real and imaginary parts of the kernel, which results in the real and imaginary parts of a third signal; (iii) Inverse FFT of the third signal to obtain the final convolved image.

Parallelism can be easily exploit since each FFT row is independent (in 1D case) as well as the FFT column, however a data dependency between rows and columns apply. After both frequency signals are available, the frequency multiplication can be performed which is a simple data-parallel SIMD operation. And finally the inverse FFT is applied to obtain the result image where the same parallel assumptions apply.

The implemented version also uses high-performance libraries - Intel MKL for CPU and CUFFT for GPU CUDA [47]. These libraries use a data-type that supports the real and the imaginary part in a single type, thus, only two domains are required, one for image spectrum and another for the kernel. All the domains are dice-able since rows (or columns) can be

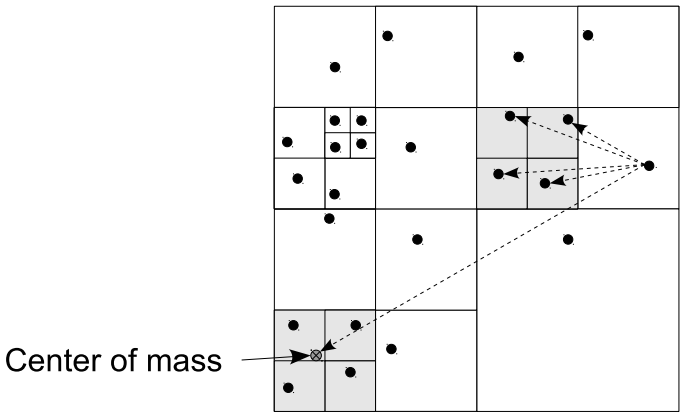


Figure 4.3: Barnes-Hut data division and forces calculation

calculated independently and in parallel.

As stated applying the FFT algorithm to an image signal requires processing the rows, transpose the result and repeat it. This transposition is done out-of-place which requires an extra domain and can be done completely in parallel. Figure 4.2 illustrates the kernels and associated domains with dependency constraints. Note that each dependency constraint requires data synchronization which means that all data is copied back to the host. This is a potential performance bottleneck specially in limited memory bandwidth systems.

4.1.3 N-Body Barnes-Hut

N-Body Barnes-Hut is a well known algorithm that performs an n-body simulation of a set of particles. It differs from other n-body simulations by approximating force calculation between particles using the center of mass of distant particles. The algorithm is supported by an Octree data structure that divides the space hierarchically. The center of mass is calculated for each cell enabling the algorithm to approximate the forces that particles induce upon each other. Figure 4.3 illustrate an interaction between a particle and particles near and far away. For each body the algorithm starts from the tree root and checks the center of mass distance for each cell. If close all child cells are visited recursively, otherwise the center of mass is used in order to approximate the forces calculation.

The algorithm is repeated for a known number of time-steps, at each step five main operations are performed: (i) Compute bounding box of the all particles; (ii) Build the octree; (iii) Compute the center of mass for each cell (iv) Calculate the interaction forces (gravitational forces); (v) Update the particle positions according to calculated interactions.

In the end of each iteration a data synchronization is performed.

Barnes-hut efficient implementation is not trivial, specially in GPUs. Burtscher et al. [15] has studied efficient mapping of this algorithm in CUDA programming model, arguing that the algorithm poses as a challenge due to its recursive properties and irregular data accesses and computation patterns.

The proposed implementation performs the three first stages - Compute bounding box, Build the octree and Compute the center of mass - of each iteration in sequential CPU code and it is not considered in the evaluations. Although not negligible, these set of operations represent a small percentage of the computation load. The focus is in forces calculation and position update. In the CPU, forces are calculated using the parallel API TBB that divides the particles across available threads and applies a kernel to each particle. The kernel will traverse the tree, calculating the interaction of one particle with all the others. This defines the basic work unit of this application. In the GPU, the kernel is similar but a single particle is assigned to a single thread.

To simplify the implementation the tree and particle Domains are read only and non diceable and a dice-able domain is used to store force interaction results, allowing each particle to be calculated independently in parallel.

4.2 Measurement models

To validate the analysis two simple metrics are used: Time-to-solution and Efficiency. Minimizing the **time-to-solution**(TTS) is the typical target of these frameworks. It is a simple and straightforward metric that enables direct and intuitive comparisons between scheduling approaches and device configurations. In order to reason about the system behaviour and gain insight about the obtained results, TTS is further decomposed into the time a device is actually busy processing tasks, T_{exec}, and the time is not processing a task, referred to as T_{idle}. T_{idle} might be due to load imbalance (the device has no task assigned), data transfer overhead and runtime system overhead. Note that under this model a device is either busy or idle, thus each time instant is accounted as either T_{exec} or T_{idle}. Therefore, for each device, $TTS = T_{exec} + T_{idle}$, which in practice, and due to potentiality measurement errors, result in $TTS \approx T_{exec} + T_{idle}$.

Ideally, $T_{idle} = 0$ meaning that all devices are always busy and $TTS = T_{exec}$ for all the devices in the system.

Efficiency (also used in [6]) expresses how the devices in HMS work together at the same

E	CPU	GPU	GPU + CPU
110%	5	3	1.7 (70% - GPU, 30% CPU)
93%	5	3	2 (60% - GPU, 40% CPU)

Table 4.1: Efficiency metric calculation example: In the first GPU + CPU execution 70% of the workload was assigned to GPU and the remaining 30% to CPU.In the second execution 10% more was assigned to CPU. Values in seconds.

time. It is a ratio between the computational power exhibited from all the architectures working together, referred to as $P_{all\_devices}$, and the sum of computational powers of each device, referred to as P_{dev_i}, executing the whole application individually. P_{dev_config} is the computational power exhibited from a device configuration executing an application with a given workload in TTS_{dev_config} seconds(Equation 4.1). The efficiency E is thus calculated as Equation 4.2 shows.

$$P_{dev\_config} = \frac{workload}{TTS_{dev\_config}} \tag{4.1}$$

$$E = \frac{P_{all\_devices}}{P_{dev\_1} + P_{dev\_2} + ... + P_{dev\_N}} \tag{4.2}$$

If the workload is not known a priori, one can use Equation 4.3, which is equivalent to equation 4.2 since workload is the same for all configurations.

$$E = \frac{\frac{1}{TTS_{all\_devices}}}{\frac{1}{TTS_{dev\_1}} + \frac{1}{TTS_{dev\_2}} + ... + \frac{1}{TTS_{dev\_N}}} \tag{4.3}$$

This enables reasoning about how computation affinity and load balancing is explored by the runtime in an heterogeneous system. In an homogeneous system the power exhibited from all devices together should usually not exceed the sum of computational powers of each device [6]. In an heterogeneous system, due to the computation affinities, the set of devices, in an ideal case, may outperform the sum of the individual powers. In Table 4.1 is an example of two hypothetical executions of the same application with different workload assignment. The GPU implementation outperforms the CPU implementation and the hybrid result is influenced by the amount of work assigned to each device, which is clearly reflected in the efficiency.

The potentially major cause of inefficiency is idleness, which is caused by several fac-

tors such as runtime system management overhead, memory transfer overhead, performance model assumptions, etc.

Chapter 5

Results

This chapter presents experimental results and evaluates the proposed runtime system, analysing how the resources are explored using different mapping policies and how the increased programming productivity is reflected in the applications.

5.1 Test setup

All measurements were made in a workstation characterized by the following details:

- **CPU:** Intel Xeon Quad-core E5630 Nehalem 32nm micro-architecture, 12M L3 Cache, 2.53 GHz, 4 Cores, 8 Threads, 40 GFLOPS theoretical peak performance per core, resulting on 160GFLOPS of global peak performance;
- **GPU:** 2x NVIDIA GeForce GTX 480 Fermi architecture, 480 CUDA Cores, 1536 MB GDDR5, 1350/168 GFLOPS single/double precision theoretical peak performance;
- **System RAM:** 12 GB RAM DDR3 1600Mhz
- **SO:** Linux 2.6 64bits
- **Compiler tools:** Intel C++ 64 Compiler 12.0, NVIDIA CUDA compiler 4.0
- **Libraries**: Intel TBB 3.0, CUDA toolkit 4.0, Intel MKL 10.3

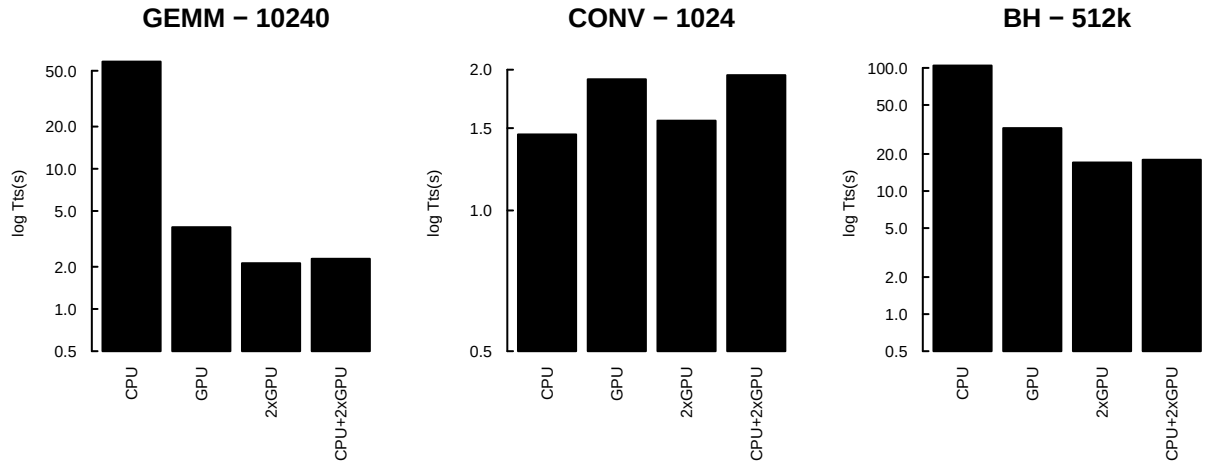


Figure 5.1: Time to solution with different device configurations and DD_DYN scheduling policy (note the logarithmic scale on the vertical axis

5.2 Exploring resources

One of the proposed goals of this dissertation is to explore the computational capabilities of the available resources in a heterogeneous system in order to reduce the time-to-solution. Figure 5.1 shows the time-to-solution (TTS) achieved with the proposed applications as devices are added to the system. Each plot depicts the result for each application with the biggest problem size and DD_DYN scheduling policy.

In the GEMM and BH applications, GPUs clearly outperform the execution of the CPU whereas in the Convolution the gain with the high level parallelism of the GPU is overwhelmed by the memory transfers performed in each blocking step of the algorithm. With two GPUs the increased parallelism level hides part of the memory transfer overhead and the performance is improved, but still lays behind the single CPU system.

However, introducing a CPU to the 2 GPUs configuration will further decrease the performance which is also noticeable in GEMM and BH. Through out the dissertation load balancing has been directly associated to data partitioning, which renders the system scheduler trapped to it. Data partitioning mechanisms have been proposed that enable the runtime system to partition data on demand and in execution time. Data logical partitions (Domains) are defined in cooperation with the application programmer and scheduler, and dynamically processed by the DMS (Section 3.3.1). In an ideal situation, task chunk sizes are defined according to available resources and/or relative performances, which requires the definition

		GEMM	CONV	BH
	T_job	2.3	2	18.2
CPU	T_exec	1.89	0.31	13.65
	T_idle	0.2	1.47	8.59
	Data Vol. Transf. (MB)	1800	2560	600
GPU0	T_exec	0.17	0.12	15.22
	T_idle	2.1	2.03	7.07
	Data Vol. Transf. (MB)	1265	1183	3084
GPU1	T_exec	0.12	0.12	14.63
	T_idle	2.15	2.03	7.66
	Data Vol. Transf. (MB)	1251	1162	3074

Table 5.1: Execution time decomposition with CPU+2xGPU device configuration and DD_DYN scheduling policy

of arbitrary task sizes, and , according to the proposed model, arbitrary data divisions.

Arbitrary data division is hard to achieve with multidimensional arrays. Dividing the data into multiple chunks with different sizes (and/or forms) results in heavy data fragmentation and might even lead to situations where no further chunks of the desired size are possible. Managing all these different and arbitrarily sized data chunks would result in additional overheads and complexity.

Therefore, the proposed data partitioning approach is based on a hierarchical division, where all chunks on the same level of the hierarchy are equally sized. Descending on level of this hierarchy (i.e. generating additional data chunks with finer granularity) requires subdividing the associated data chunk into a number of smaller equally sized sub-chunks. For example on a system with two GPUs and one CPU, where each GPU has a computing capability much larger than the CPU, the initial data is divided into a number of large, equally sized data chunks. One of these data chunks is then further subdivided into a number of smaller equally sized sub-chunks in order to generate tasks appropriate to the CPU, whereas larger ones are mapped onto the GPUs. When the GPUs finish their tasks, they will start receiving the remaining smaller data chunks, which the CPU did not processed yet. Processing small chunks of data reduces device throughput due to several factors, such as increased kernel invocation overhead, potentially reduced device occupancy levels, reduced parallel execution levels, among others. The penalty is more evident if an added device compute capability does not compensate the increased loss of throughput consequent of considering the new device compute capabilities. Figure 5.1 depicts this situation for the three applications where a CPU with about 1/10 relative compute capability is added to a 2xGPU device configuration.

Table 5.1 illustrates an approximate decomposition of the execution time which shows that the devices are idle for a significant part of the execution time. This is specially no-

		DD_DUM	DD_DYN
	T_job	1.3	2
CPU	T_exec	0.37	0.31
	T_idle	1.01	1.47
	Data Vol. Transf. (MB)	2560	2560
GPU0	T_exec	0.08	0.12
	T_idle	1.37	2.03
	Data Vol. Transf. (MB)	844	1183
GPU1	T_exec	0.06	0.12
	T_idle	1.39	2.03
	Data Vol. Transf. (MB)	716	1162

Table 5.2: CONV execution time decomposition with CPU+2xGPU device configuration. Data Vol. Transf. is the amount of bytes transferred by each device. The CPU represent the transfers that the host performed in order to provide a task private address space (Section 3.3.1)

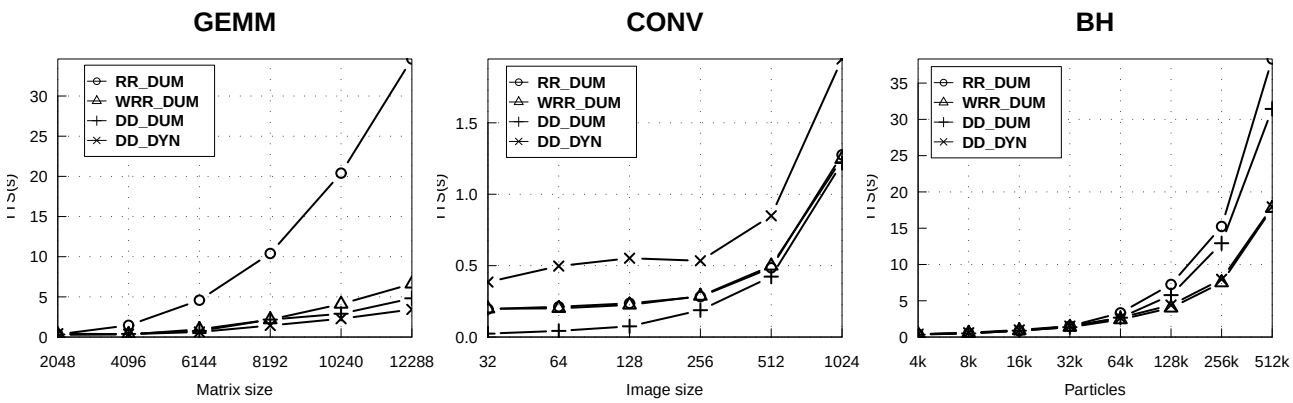


Figure 5.2: Time to solution comparing different schedulers for different problem sizes with 2xGPU+CPU device configuration

ticeable for the CONV study case that performs several intermediate data synchronizations (T_idle includes data transfers times). This is potentially due to memory transfers and related overheads, which suggests poor data management efficiency, that needs to be enhanced in order to minimize the device exhibited idle times, increasing usability and overall performance.

5.3 Schedulers' behaviour

Figure 5.2 depicts the behaviour of the four schedulers for each application with a 2xGPU+CPU device configuration. In the GEMM and BH case the RR_DUM scheduler result is seriously affected by the load imbalance that causes the GPUs to idle while the CPU is processing its

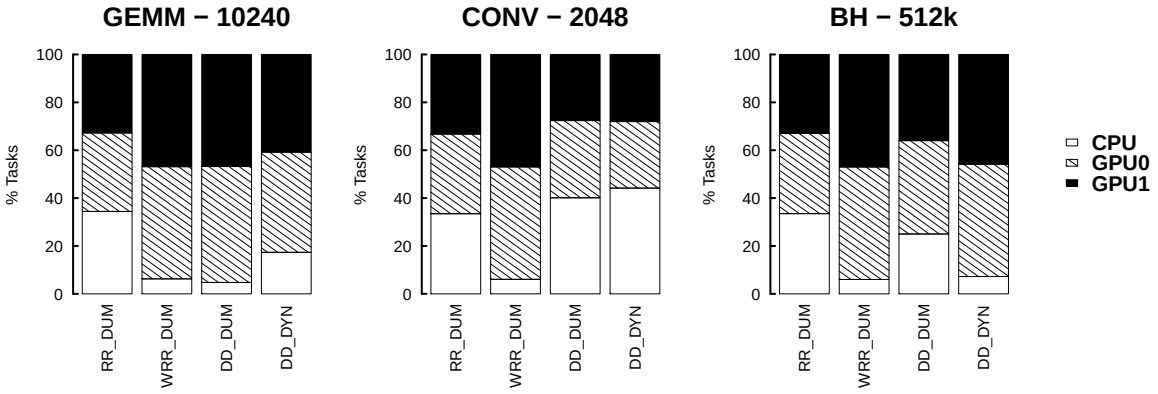


Figure 5.3: Workload distribution to the different devices, comparing different schedulers

tasks. On the other hand, the DD_DYN is well favoured by the demand driven mechanism coupled with good performance insights provided by the performance model, i.e., the theoretical peak FLOPS are close to the real FLOPS that the devices deliver performing the algorithm. In the BH case the WRR_DUM is slightly better than DD_DYN due to more work assigned to the GPU and the DD_DYN scheduling policy overhead (work-stealing).

However, for CONV DD_DYN's performance is affected by a data partitioning that follows a performance model which does not correspond to the real performance of the kernels. Table 5.2 shows that the devices are idle for a substantial period of the execution time. This scheduler is thus outperformed by the DD_DUM policy that uses coarse-grain tasks reducing the memory transfer overhead.

As also evidenced in the previous section, the CONV study case reveals that the efficiency of the data-management systems is a crucial factor when addressing heterogeneous systems. The limited bandwidth and the overhead required to maintain data consistent may substantially contribute for a poor overall performance.

Figure 5.3 depicts how the schedulers assigned the work load to available devices. The static characteristic of the round-robin mechanism is easily noticed, as well as the dynamism of the demand driven policies. The static nature of the RR policies is responsible for imbalances in most cases, except if the performance model associated with WRR_DUM accurately matches the delivered performance, i.e., when the theoretical peak performance corresponds to the sustained performance, which on the case studies happens only for the BH application.

Figure 5.4 shows the efficiency of the four schedulers with multiple problem sizes. The GEMM and BH applications reveal increased efficiency for DD_DYN and WRR_RUM policies

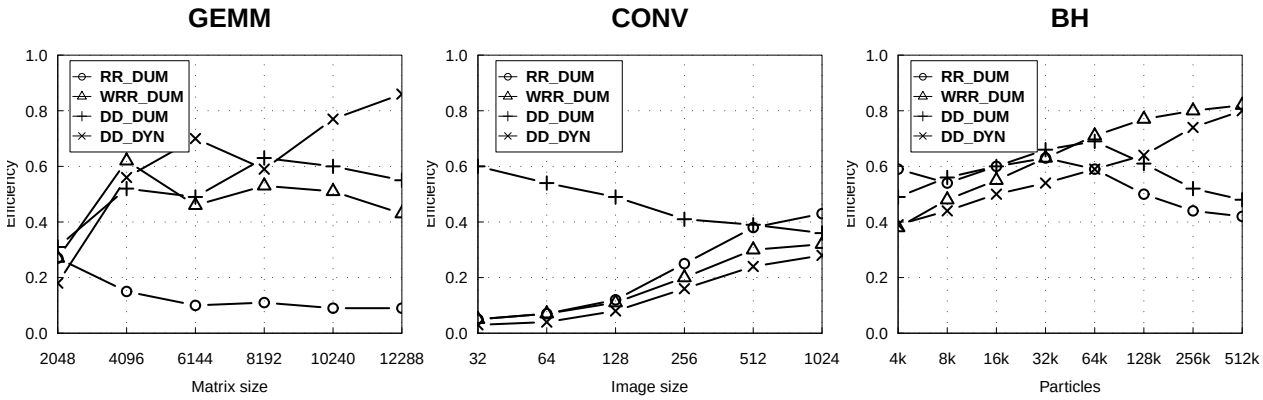


Figure 5.4: Efficiency obtained with different schedulers for the different problem sizes. 2xGPU+CPU device configuration.

as the problem size grows, which reveals the important contribution of a performance model when it matches the real sustained performance. However, for the CONV study the overall efficiency is quite affected as the image size changes, which also reveals that the DMS renders the system inefficient. It is also noticeable that the performance model driven approaches present a lower efficiency due to the mentioned inaccuracy of the proposed performance model when evaluating this application (i.e. WRR_DUM and DD_DYN for CONV).

5.4 Programmer's Productivity

One of the goals of this dissertation is to propose an intuitive programming interface to express applications, easing the handling all the devices' tools and disjoint memory spaces, and also maximizing the use of a HMS, therefore increasing the programmer's productivity.

Adding a new device with a new architecture to the system only requires the programmer to provide implementations to the Device API interface and kernels. If adding a new device with an architecture that is already in the system, it is only required to register the new device with a system primitive and no additional code is required.

An alternative to using a framework that explicitly targets multiple heterogeneous devices is to develop applications for a given architecture using optimized libraries and computing kernels. This latter option will often guarantee added performance, but hinders scalability onto different architectures and requires the programmer to master device specific development tools and APIs.

Application	Device	Label	Description
GEMM	CPU	CPU_MKL	Multi-core GEMM using Intel MKL
	GPU	GPU_cuBLAS	Single GPU, GEMM using cuBLAS
CONV	CPU	CPU_MKL	Multi-core, FFT using Intel MKL and Intel TBB to transpose kernels
	GPU	GPU_FFT	Single GPU, FFT using cuFFT and a regular CUDA kernel to transpose
BH	CPU	CPU_Burtscher	Multi-core, Martin Burtscher [38] code using Intel TBB
	GPU	GPU_Burtscher	Martin Burtscher [15] code using CUDA

Table 5.3: Optimized libraries used for each case study

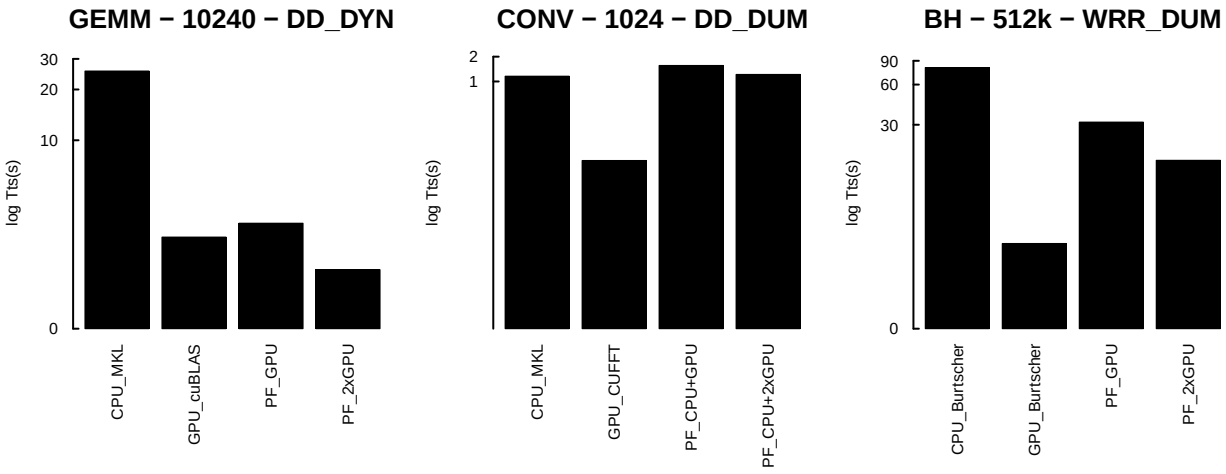


Figure 5.5: Comparing PERFORM best scheduler and device configuration with other implementations (note the logarithmic scale on the vertical axis).

In order to assess the performance losses associated with using the PERFORM framework (versus platform specific applications) optimized versions of the three case studies were used according to Table 5.3. Note that all these optimized versions use a single device, i.e., one CPU or GPU, but fully exploit the parallelism within the device.

To compare performance the best PERFORM solutions were selected, meaning that the time-to-solution reported was obtained with the best scheduler and device configuration. Results obtained with this framework are labelled with the prefix PF to ease readability.

Figure 5.5 clearly shows that for the GEMM case study there are clear advantages on using PERFORM. PF_GPU is slightly worst than GPU_cuBLAS, but the former scales to multiple GPUs without any programmer effort and achieves better results on 2xGPUs than any of the device optimized alternatives.

However, for the CONV and BH case studies the single GPU optimized application clearly outperforms the PERFORM alternatives even with multiple GPUs. The throughput reduction consequent of the regular data partitioning mentioned in previous sections

(that also applies to the GEMM case) is a potential cause for this loss of performance. In the CONV case the major cause is the memory transfers that are performed each time a synchronization is required, whereas in the GPU_CUFFT the whole data is kept in the device the whole time. This memory transfer overhead seriously affects the performance that even the CPU_MKL slightly outperforms the 2xGPU PEFORM version. The PERFORM versions of BH are similar to CPU_Burtscher, whereas the GPU_Burtcher is a hand-tuned CUDA specific approach that is designed to explore the maximum capabilities of a CUDA device.

The approach studied throughout this dissertation aims at achieving scalability across multiple heterogeneous devices, while requiring minimum programmer effort. The achieved results look promising for regular applications requiring minimum intermediate synchronization stages and data movements as demonstrated by the matrix multiplication case study. Applications requiring multiple intermediate synchronization and data transfer operations, such as CONV, still suffer from data management overheads that impact significantly on the time to solution. Data management must thus be revised and an optimized solution which minimizes overheads is mandatory. The BH case study also requires intermediate synchronization stages and associated data transfers (at the end of each timestep) thus suffering from the same problem as CONV. Furthermore, it is compared to a highly optimized GPU version of BH code, which is currently considered among the fastest BH implementations available. The PF_GPU kernels are by no means optimized to this level.

Chapter 6

Conclusions and Future Work

This dissertation proposes a runtime system that tackles heterogeneous many-core systems. It provides a programming and execution model that explores the computational resources available in a heterogeneous system. The programming interface provides primitives for job creation, data and kernel association and application flow control that enables the programmer to express application functionality and data. Applications are expressed using Kernels, one for each architecture, and Domains following a data-parallel computing paradigm. The runtime is composed by an unified address space, which keeps the programmer agnostic to data movements, and a data management system, based on a related project, that provides consistency among multiple copies of data and mechanisms for data partitioning and transparent data transfers. The applications are mapped onto the available devices using scheduling and data partitioning policies. Four schedulers are proposed and evaluated that try to balance the workload across the available devices. Two of the schedulers follow a static round-robin map approach (RR_DUM,WRR_DUM), whereas the remaining two follow a demand driven approach (DD_DUM, DD_DYN). Both WRR_DUM and DD_DYN use a simple performance model that provides performance insights to these schedulers, influencing their data partitioning policies and mapping mechanisms.

The framework was evaluated using three different case studies - single precision matrix multiplication (GEMM), image convolution with FFT (CONV) and a n-body simulation with Barnes-Hut irregular algorithm (BH) - implemented in a system composed by a CPU and two GPUs.

In terms of productivity, the achieved results look promising for regular applications that have few data synchronization points as demonstrated with the GEMM application. The

reported results reveal clear advantages in using the PERFORM framework.

Associating scheduling and data partitioning prevents the runtime system from generating tasks with arbitrary granularity, since these are tightly coupled with data and this can not be partitioned into arbitrary subdomains. The proposed data partitioning model divides data hierarchically in equally sized data chunks, which reduces the devices' throughput and overall performance, as evidenced for all the study cases. This throughput reduction should thus be tackled by revising data partitioning mechanisms, providing efficient and adequate data division techniques that suit the needs of such scheduling methodologies, but also that do not compromise the overall performance.

Static work decomposition and mapping, ignoring the devices relative performances, is clearly inadequate as demonstrated by the RR_DUM scheduler in GEMM and BH case studies. Dynamism in data partitioning and scheduling adaptation seems mandatory to achieve acceptable levels of efficiency, as well as performance model driven decisions. However, the performance model used through out this work was too simple to capture all the subtleties of the applications behaviour, as demonstrated by the CONV with the DD_DYN policy. More accurate and sophisticated performance models are required and will be studied in the context of future work.

Results have also shown that the data management system is a crucial component when addressing heterogeneous many-core systems, mainly due to the disjoint address spaces and limited bandwidths . The CONV study case, that performs several intermediate data synchronization points, demonstrates a clear need for a more efficient data management. The DMS must thus provide more efficient data transfer mechanisms and minimize data transfers with a more complete exploitation of data locality.

Summarizing, results have shown that more flexible data partitioning strategies are required to allow for arbitrary tasks' granularity, an accurate performance model is mandatory to allow for informed dynamic scheduling and the data management system must provide efficient data transfers and an extensive exploitation of data locality, otherwise the whole system performance might be compromised. The proposed runtime system performs well for regular applications with minimal synchronization points, which support our initial hypothesis that an unified programming and execution model would increase the programmer's productivity without impacting too severely on performance. These conclusions will moti-

vate future work, concentrating on guaranteeing acceptable efficiency levels for the set of components identified throughout this dissertation, thus providing an unified execution and programming model that releases the programmer from dealing with the challenges that a HMS pose, allowing him to focus on the algorithm functionality, while fully exploiting the computational power of such systems.

Bibliography

[1] I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors, May 1996.

[2] Ishfaq Ahmad and Yu-kwong Kwok. A New Approach to Scheduling Parallel Programs Using Task Duplication. *1994 International Conference on Parallel Processing (ICPP'94)*, pages 47–51, August 1994.

[3] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*, page 145, 2009.

[4] A.H. Alhusaini, V.K. Prasanna, and C.S. Raghavendra. A unified resource scheduling framework for heterogeneous computing environments. *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*, pages 156–165.

[5] Cédric Augonnet, Jérôme Clet-ortega, Samuel Thibault, and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. 2010.

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-andré Wacrenier. StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par 2009 Parallel Processing 15th International Euro-Par Conference*, volume 2009, 2009.

[7] David August, Keshav Pingali, Derek Chiou, Resit Sendag, and Joshua J. Yi. Programming Multicores: Do Applications Programmers Need to Write Explicitly Parallel Programs? *IEEE Micro*, 30(3):19–33, May 2010.

[8] Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors,

Euro-Par '09 Proceedings of the 15th International Euro-Par Conference on Parallel Processing, volume 5704 of *Lecture Notes in Computer Science*, pages 851–862, Berlin, Heidelberg, August 2009. Springer Berlin Heidelberg.

[9] R. Bajaj and D.P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, February 2004.

[10] S. Baskiyar and P.C. SaiRanga. Scheduling directed a-cyclic task graphs on heterogeneous network of workstations to minimize schedule length. *2003 International Conference on Parallel Processing Workshops, 2003. Proceedings.*, pages 97–103, 2003.

[11] Pieter Bellens, Josep Perez, Rosa Badia, and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. *ACM/IEEE SC 2006 Conference (SC'06)*, (November):5–5, November 2006.

[12] Carsten Benthin, Ingo Wald, Michael Scherbaum, and Heiko Friedrich. Ray Tracing on the Cell Processor. *2006 IEEE Symposium on Interactive Ray Tracing*, pages 15–23, September 2006.

[13] P. Bhaniramka. AMD Stream SDK - Introduction and Overview. *PDC/AMD Workshop on GPGPU Programming*, 2008.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk. *ACM SIGPLAN Notices*, 30(8):207–216, August 1995.

[15] Martin Burtscher and Keshav Pingali. An Efficient CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm. In *GPU Computing Gems*, page 886. Morgan Kaufmann, 2011.

[16] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[17] 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.

[18] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107, January 2008.

[20] R.E. Diaconescu and H.P. Zima. An Approach To Data Distributions in Chapel. *International Journal of High Performance Computing Applications*, 21(3):313–335, August 2007.

[21] Gregory Diamos and Nathan Clark. Ocelot : A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. *Computer Engineering*, 2010.

[22] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing - HPDC '08*, page 197, New York, New York, USA, June 2008. ACM Press.

[23] a. Dogan and R. Ozguner. LDBS: a duplication based scheduling algorithm for heterogeneous computing systems. *Proceedings International Conference on Parallel Processing*, pages 352–359, 2002.

[24] Romain Dolbeau. HMPP : A Hybrid Multi-core Parallel. In *GPGPU Workshop October*, pages 1–5, 2007.

[25] Kayvon Fatahalian, Timothy Knight, Mike Houston, Mattan Erez, Daniel Horn, Larkhoon Leem, Ji Park, Manman Ren, Alex Aiken, William Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. *ACM/IEEE SC 2006 Conference (SC'06)*, (November):4–4, November 2006.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.

[27] Isaac Gelado, Javier Cabezas, Nacho Navarro, John E. Stone, Sanjay Patel, and Wenmei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. *ACM SIGPLAN Notices*, 45(3):347, March 2010.

[28] a. Gerasoulis. DSC: scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[29] Peter N Glaskowsky. NVIDIA Fermi : The First Complete GPU Computing Architecture. (September), 2009.

[30] Tianyi David Han and Tarek S. Abdelrahman. hi CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, pages 52–61, New York, New York, USA, March 2009. ACM Press.

[31] Mark D. Hill and Michael R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, July 2008.

[32] Intel. TBB Home.

[33] Intel. Intel Math Kernel Library 10.2, 2009.

[34] Intel. Intel® Array Building Blocks - Documentation - Intel® Software Network, 2010.

[35] J. a. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, July 2005.

[36] Khronos. Khronos OpenCL API Registry.

[37] Samuel Webb Williams Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[38] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A Suite of Parallel Irregular Programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software*, 2009.

[39] Jaejin Lee, Sangmin Seo, Chihun Kim, Junghyun Kim, Posung Chun, Zehra Sura, Jungwon Kim, and SangYong Han. COMIC: a coherent shared memory interface for cell be. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques - PACT '08*, page 303, New York, New York, USA, October 2008. ACM Press.

[40] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per

Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38(3):451–451–460–460, June 2010.

[41] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *ACM SIGPLAN Notices*, 44(10):227, October 2009.

[42] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. *ACM SIGARCH Computer Architecture News*, 36(1):287, March 2008.

[43] Michael D Mccool, Weber St N, and Waterloo Ontario. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. *Program*, 2006.

[44] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, January 1998.

[45] NVIDIA. Compute Unified Device Architecture Programming Guide. 2007.

[46] NVIDIA. *CUDA CUBLAS Library 4.0*. nVidia Corporation, August 2011.

[47] NVIDIA. CUDA CuFFT Library, 2011.

[48] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[49] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface*. Morgan Kaufmann, 2008.

[50] M Pharr. Interactive Rendering in the Post-GPU Era. *Graphics Hardware Workshop 2006*, 2006.

[51] Michael J. Quinn. *Parallel Programming in C with Mpi and Openmp*. McGraw-Hill Education, 2008.

[52] Andrei Radulescu and Arjan J. C. Van Gemund. Fast and Effective Task Scheduling in Heterogeneous Systems. page 229, May 2000.

[53] Larry Seiler, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Pat Hanrahan, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, and Jeremy Sugerman. Larrabee. *ACM Transactions on Graphics*, 27(3):1, August 2008.

[54] Fadi N. Sibai. Nearest neighbor affinity scheduling in heterogeneous multicore architectures. *Journal of Computer Science & Technology*, page 22, 2009.

[55] Gilbert C Sih and Edward A Lee. A Compile-Time Scheduling Heuristic Heterogeneous Processor Architectures. 4(2):175–187, 1993.

[56] H. Singh and A. Youssef. Mapping and Scheduling Heterogenous Task Graphs Using Genetic Algorithms, 2002.

[57] Håkon Kvale Stensland, Carsten Griwodz, and På l Halvorsen. Evaluation of multicore scheduling mechanisms for heterogeneous processing architectures. In *Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '08*, page 33, New York, New York, USA, May 2008. ACM Press.

[58] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. *MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs*, volume 5335 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, November 2008.

[59] Computing Systems and Consultation Meeting. Research Challenges for Computing Systems. *ICT Work Programme*, (November 2007), 2010.

[60] Andrei Tatarinov and Alexander Kharlamov. Alternative Rendering Pipelines on NVIDIA CUDA. *SIGGRAPH*, 2009.

[61] Samuel Thibault and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. *Processing*, (Hppc), 2009.

[62] TOP500. TOP500 Supercomputing List.

[63] H. Topcuoglu and S. Hariri. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.

[64] Stanley Tzeng, Anjul Patney, and John D Owens. Task Management for Irregular-Parallel Workloads on the GPU. 2010.

[65] Sain-Zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen-Mei W. Hwu. *CUDA-Lite: Reducing GPU Programming Complexity*, volume 5335 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, November 2008.

[66] J D Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[67] S. L. Sze W. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, J. E. Mahoney, L. T. Ngo. A user programmable reconfigurable gate array. In *Proc. Custom Integrated Circuits*, pages 233 – 235, 1986.

[68] Perry H Wang, Jamison D Collins, Gautham N Chinya, Hong Jiang, Xinmin Tian, Milind Girkar, Nick Y Yang, Guei-yuan Lueh, and Hong Wang. EXOCHI : Architecture and Programming Environment for A Heterogeneous Multi-core Multithreaded System. *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–166, 2007.

[69] Katherine Yelick, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, Tong Wen, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, and Paul Hilfinger. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation - PASCO '07*, page 24, New York, New York, USA, July 2007. ACM Press.