Fernando Jorge Marques Alexandre

Licenciado em Engenharia Informática

Multi-GPU Support on the Marrow Algorithmic Skeleton Framework

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Hervé Paulino, Professor Auxiliar,
Faculdade de Ciências e Tecnologia da Universidade
Nova de Lisboa

Júri:

Presidente:	Doutora Teresa Romão (FCT-UNL)
Arguente:	Doutor Pedro Tomás (IST-TU)
Vogal:	Doutor Hervé Paulino (FCT-UNL)

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2013

Multi-GPU Support on the Marrow Algorithmic Skeleton Framework

Acknowledgements

I would firstly like to thank my adviser Hervé Paulino, whose unwavering support and patience helped me immensely with this thesis. Furthermore I would like FCT/UNL for allowing the creation of this thesis with an excellent working environment. I would like to thank the projects PTDC/EIA-EIA/102579/2008 and PTDC/EIA-EIA/111518/2009 for providing the equipment used for the development of this thesis.

I would like to thank Prof. Pedro Medeiros and Flávio Martins for the patience when the development machines would mysteriously stop working and required their intervention.

I thank all who frequent the ASC laboratory for providing such a good and amusing work environment, where friendly support is a desk away.

I would like to thank all my friends, specially Filipe Rodriges, Carlos Correia, Paulo Ferreira and Eduardo Duarte who put up with me and my frustrations.

I thank and bow to my family, for allowing me to focus on my work while supporting me wholeheartedly.

Finally, I would like to thank Margarida Mendes, for being with me, and supporting me through all times.

Muito, muito obrigado a todos!

Abstract

With the proliferation of general purpose GPUs, workload parallelization and data-transfer optimization became an increasing concern. The natural evolution from using a single GPU, is multiplying the amount of available processors, presenting new challenges, as tuning the workload decompositions and load balancing, when dealing with heterogeneous systems.

Higher-level programming is a very important asset in a multi-GPU environment, due to the complexity inherent to the currently used GPGPU APIs (OpenCL and CUDA), because of their low-level and code overhead. This can be obtained by introducing an abstraction layer, which has the advantage of enabling implicit optimizations and orchestrations such as transparent load balancing mechanism and reduced explicit code overhead.

Algorithmic Skeletons, previously used in cluster environments, have recently been adapted to the GPGPU context. Skeletons abstract most sources of code overhead, by defining computation patterns of commonly used algorithms. The Marrow algorithmic skeleton library is one of these, taking advantage of the abstractions to automate the orchestration needed for an efficient GPU execution.

This thesis proposes the extension of Marrow to leverage the use of algorithmic skeletons in the modular and efficient programming of multiple heterogeneous GPUs, within a single machine.

We were able to achieve a good balance between simplicity of the programming model and performance, obtaining good scalability when using multiple GPUs, with an efficient load distribution, although at the price of some overhead when using a single-GPU.

Keywords: Algorithmic Skeletons, Multiple GPUs, Auto-Tuning

Resumo

Com a proliferação de placas gráficas (GPUs), tornou-se fulcral a paralelização de trabalho e a otimização da transferência de dados. A evolução natural de um GPU é a sua replicação, criando novos desafios, como por exemplo, a afinação automática das partições de trabalho e o seu balanceamento quando se lida com sistemas heterogéneos.

Programação de alto nível é um ativo muito importante em programação com vários GPUs, devido à complexidade inerente às APIs de GPGPU correntes (OpenCL e CUDA), pois estes são de baixo nível e têm alta verbosidade no código necessário para inicialização do sistema, tal como a comunicação entre CPU e GPUs. Pode-se obter este nível de programação introduzindo uma nova camada de abstração, que tem a vantagem de abstrair otimizações e orquestrações implícitas, tais como balanceamento de carga transparente e menos verbosidade no código necessário para inicialização.

Os esqueletos algorítmicos, originalmente utilizados em *clusters*, foram recentemente adaptados para GPGPU. Estes abstraem código fonte factorizável, definindo padrões de computação amplamente utilizados. A biblioteca de esqueletos algorítmicos Marrow, por exemplo, utiliza abstrações para automatizar a orquestração necessária para a execução eficiente em GPU.

Esta tese propõe a extensão do Marrow para aplicar o uso de esqueletos algorítmicos para uma programação modular e eficiente sobre múltiplos GPUs heterogéneos, numa única máquina.

Conseguimos um bom equilíbrio entre simplicidade do modelo de programação e performance, obtendo uma escalabilidade boa com múltiplos GPUs, com uma distribuição de trabalho eficiente e uma reduzida penalização no desempenho quando só se usa um GPU.

Palavras-chave: Esqueletos Algorítmicos, Multiplas GPUs, Afinação Automática

Contents

List of Figures

List of Tables

Listings

1

Introduction

1.1 Motivation

Graphic Processing Units (GPUs) have become an ubiquitous component in computer systems, being responsible for all graphical-related operations. These specialized units were adopted because the limitations of common CPU architectural design do not scale well with graphics, as graphical operations are commonly parallel pixel-wise computations, entailing long iterations between frames, due to the low number of concurrent threads available. The GPU overcomes the need for long iterations by adopting an architecture which is comprised of many streaming processors. These are relatively weak, currently with frequencies close to 1GHz, in comparison to their CPU counterpart, which boasts of frequencies above 2-3GHz. The AMD 7970 GPU[1], for example, contains 2048 stream processors, each at a frequency of 925MHz, 128 texture units and 128 stencil units as well as 3GB GDDR5 on-board RAM memory.

The GPGPU (General Purpose Computation on Graphics Processing Units) concept was first formally proposed by Harris et al. as an approach to general computations using GPUs. The untapped potential of the GPUs was initially recognized by the graphics processing community, the audience of GPGPU's initial presentation [Lue+05], in 2005, and later the supercomputer community, where the first work regarding GPGPU was published [Lue+06], in 2006. Consequently, the hardware evolved to support these needs, and is considered an important asset for the resolution of computationally-heavy problems.

More recently, with the proliferation of GPGPU, the need for additional computation

[1] Taken from www.amd.com/us/products/desktop/graphics/7000/7970/Pages/radeon-7970.aspx

power evolved the system architectures, enabling the concurrent operation of several GPU devices within a single system, as well as clustered solutions where nodes can be GPU-accelerated with one or more GPU devices. In this work, we are interested in the first of these two architectures, as there has been recent work [Hua+06; Pot+12; LBB11] being developed in this hardware infrastructure, with good performance results. Additionally, this type of architecture has been increasingly popular as a base platform for GPU-accelerated clusters. An evidence of such claim is the growing use of the GPU potential in supercomputers, as the 17 from the 100 fastest supercomputer clusters [Meu+st] (including the fastest), such as the Titan and the TH-1A, are GPU-accelerated, some of which also present exceptional power-efficiency [Comst]. From these, two already adopt the latest architecture development, boasting of multiple GPUs per node, such as the Tsubame 2.0 (three per node) and Ha-Pacs (four per node).

1.2 Problem

The most popular GPGPU API developed with native GPU support was CUDA [NVI08], where GPU computations are defined using a C-like programming language, subsequently compile to generate a binary for the GPU's architecture. As the GPU computations (*kernels*) are defined through a subset of the C language (with some extra keywords), and the main program can be written in C++ or other languages (such as Fortran and Python) using bindings, the learning curve associated with GPGPU is lessened, simplifying the creation of GPU-accelerated applications. As CUDA is a proprietary technology, locked to NVIDIA GPUs, the need for an open standard emerged and a conglomerate of hardware developers such as Intel, AMD and NVIDIA, collaborated to create OpenCL [Khr12]. Inspired by CUDA, OpenCL also uses a subset of the C language to define kernels. A unified architecture is exposed to the programmer, hiding heterogeneous details of OpenCL-enabled devices, such as CPUs and GPUs. The main usage of GPGPU APIs is scientific simulation, as these commonly require a high-degree of parallelism to process large volumes of data, which GPUs excel at, by taking advantage of their on-board memory and high number of streaming processors.

The challenges entailing the usage of these APIs arise from the large responsibility delegated to the programmer, as well as from their low-level nature. All computations must be explicitly launched and managed (a particularly complex task when the computation requires the execution of multiple kernels), and there is no automated GPU memory management, delegating all allocations and deallocations, as well as its optimizations (by picking the most suited memory type and appropriate transfer timing) to the programmer. This not only requires an in-depth knowledge of the architecture exposed by the API, but also of the underlying GPU hardware, as each architecture's performance differs. This is shown by Spafford et al. [SMV10] where the best execution configuration (work-group size and buffer chunk size) widely varies from GPU to GPU.

When taking advantage of multiple GPUs several concerns arise, such as workload

scheduling and auto-tuning, akin to the more mature distributed CPU architecture, which also requires communication for a cooperative execution. Even though the currently available GPGPU APIs enable the multi-GPU execution, they do not address these issues [SK09]. These components tend to become more complex to develop in multi-GPU systems, due to the limited nature of the APIs, such as the lack of profiling tools and the limited feedback provided by the GPU executions. To address these issues several languages have been incrementally adapted, to interface with the GPGPU APIs implicitly, such as, StreamIt [TKA02], Lime [Dub+12] and Chapel [Sid+12]. These are not mainstream programming languages, and therefore their impact is somehow limited, as it is a known fact that library based approaches are usually more effective to convey a given feature to a wider audience. In this context, algorithmic skeleton frameworks are playing an important role.

Algorithmic skeletons [Col91] are common-pattern abstractions, whose initial focus was on clusters (from which T4P [Dar+93] and P^3L [Bac+95] are examples). This concept created a shift in the programming paradigm at the time, enabling a higher-level programming, as most of the synchronization and communications are predefined, reducing the number possible errors as well as reducing amount of knowledge required from the programmer. Skeletons are an important asset in the multi-GPU context, as CUDA and OpenCL do not support an efficient multi-GPU execution natively. As skeletons abstract well-known patterns, these can be also used to reduce the code overhead present in the GPGPU APIs regarding computation initialization and communication. This layer also allows transparent optimizations such as load balancing mechanisms.

Applications using multi-GPU architectures are dominated by data-parallelism, as it is the simplest approach to take advantage of the degree of parallel threads available in GPU devices, at the expense of computation flexibility as these can only have very limited dependencies. Current skeleton frameworks only offer premature support for this type of execution, by limiting the computation type as well as limited auto-tuning and scheduling techniques, as these frameworks are still under active development. Frameworks, such as SkelCL [SKG11], SkePU [EK10] and Muesli [CK10] (detailed in Subsection 2.4.1), provide support for multi-GPU execution, by providing parameterizable algorithms, such as the Map and Scan, requiring only the data-set in a predefined structure as well as the functions that will be applied at each stage (for example a split and merge function in the Map model).

There are, however, other classes of applications that, although still data-parallel, commonly using Map pattern [SO11; CA12] models, start to incorporate some task-parallelism. These hybrids commonly combine coarse-grained task-parallelism with fine-grained data-parallelism, which present other programming models such as data-flow processing (as presented by Boulos et al. [Bou+12]) and stream processing (as presented by Repplinger et al. [RS11] and Dubach et al. [Dub+12]). These applications can normally be factorized into basic constructs, such as pipelines, loops and stencils.

Recently, the Marrow [Mar12] algorithmic skeleton library was proposed, addressing

these concerns using a single GPU architecture. The goal of this thesis is to adapt Marrow into an architecture with a single machine containing multiple, and possibly, heterogeneous GPUs.

1.3 Proposal

Marrow is a framework for the construction and execution of complex computations in GPUs (OpenCL enabled devices) while addressing modularity and performance concerns. Modularity is achieved by using skeleton nesting, a novel approach in the GPGPU skeletons context, which allows the composition of basic constructs to create complex and structured computations. This feature enables a higher degree of control from Marrow, by abstracting individual skeleton execution from the programmer and enabling the library to integrate optimizations transparently.

The computations in Marrow are represented in a tree structure, where the intermediary nodes are skeletons, and leaves are computational kernels. This structure is obtained using skeleton nesting, which gives the programmer an opaque vision of the computation, only exposing the result. This allows Marrow to take complete control of the communication as well as computation orchestrations.

The currently supported skeletons are: Loop (and its specialization For), Pipeline, Stream and MapReduce. The Loop and Pipeline skeletons are specially useful for commonly used for GPU-accelerated applications, such as, simulations and image processing algorithms, as they give the programmer easy access to iterative and sequential stage constructs. The Stream skeleton introduces the communication and computation overlap, by using a task-parallel model internally. This feature, allows two-way data-transfers while there is ongoing computation on the GPU device. Tasks are internally submitted over time to the GPU device, enabling the orchestration of the overlap. The MapReduce skeleton gives Marrow support for the Map-Reduce data-parallel programming model. Marrow is detailed in Chapter 3.

This thesis intends to adapt the Marrow framework to address the aforementioned problems regarding multiple, possibly heterogeneous, GPUs, computations as well as the complexity these entail. Additionally, this adaptation introduces new constructs to the multi-GPU context such as the Loop and Pipeline, as well as new techniques to this context such as skeleton computation trees for complex and structured computations. Our multi-GPU execution entails a new challenge, which is the efficient mapping of computations into a set of variable-performance GPUs. Following the algorithmic skeleton paradigm, this mapping should be as seamless as possible, as to abstract the underlying platform from the programmer, being another goal of this thesis. The multi-GPU support is detailed Chapter 4.

As such, the ultimate goal of this thesis is to use task- and data-parallel algorithmic skeletons for a modular and efficient usage of multi-GPU systems.

1.4 Contributions

The contributions of this thesis are:

- The exhaustive performance evaluation of the original, single-GPU Marrow to obtain a baseline for comparison. This contribution helped in the creation of a paper [Mar+13] published at EuroPar 2013;
- Adaptation of the Marrow algorithmic skeleton library to an architecture containing a single node with multiple GPUs, with a particular focus in task-parallel skeletons. Given that Marrow is the first to feature these skeletons, their systematic orchestration in a multi-GPU context is a contribution to the community. This contribution has been published [AMP13] at INForum 2013;
- The adaptation of skeleton computation trees for the multi-GPU context, for the creation of complex and structured computations. Given that Marrow is the first to introduce this feature in the GPGPU context, the efficient distribution of these computation trees to multiple GPUs is also a contribution to the community.
- The evaluation of the previously presented contributions, by measuring: 1) overhead versus OpenCL implementations and the original, single-GPU Marrow, 2) our work's multi-GPU scalability, 3) work distribution efficiency and 4) an attempt to evaluate the productivity gains associated with Marrow's usage in the multi-GPU context.

1.5 Document Structure

This document has the following structure:

Chapter 2 In this chapter the state of the art is presented, relating to GPU and OpenCL architectures, multi-GPU execution frameworks (with special focus on algorithmic skeletons), approaches to auto-tuning and workload scheduling in heterogeneous multi-GPU environments.

Chapter 3 The original Marrow algorithmic skeleton framework is detailed, before the integration of our contributions.

Chapter 4 This chapter details the implementation of Marrow's multi-GPU support.

Chapter 5 Our contributions are validated in this chapter, using several metrics, such as: 1) comparison with case-studies in OpenCL, 2) in the original, single-GPU, Marrow, 3) multi-GPU scalability as well as 4) a decomposition quality evaluation.

Chapter 6 In this chapter this thesis' conclusions are presented.

2

State of the Art

As the work we propose involves algorithmic skeleton programming in a multi-GPU execution environment, we overview in detail the state of the art regarding algorithmic skeleton libraries and languages which take advantage of multi-GPU architectures. Seeing that we are handling heterogeneous GPU devices, an overview is presented, detailing the current approaches to auto-tuning and workload scheduling (which is further categorized to static and dynamic), specifically in the multi-GPU context, as these already take into account the limitations present in the GPGPU environment. Finally, the last section of this chapter presents the final remarks relating to the state of the art, highlighting the tendencies of the studied topics.

Before diving into these more advance topics, however, it is necessary to present a base platform of knowledge to allow the reader to comprehend the technical terms and implications more easily. This platform comprises of a general overview of the GPU architecture, a detailed presentation of OpenCL's architecture and programming model, as it will be used for this work's implementation, as well as a taxonomy definition, where several categories of GPU-accelerated systems are defined, with a brief overview of the state of the art in each one, to simplify the categorization of systems in the remainder of this document.

2.1 GPU Architecture

The GPU architecture is generally analogous to Figure 2.1, boasting of several streaming multiprocessors (SMs), which are autonomous, independent of other SMs. In this section we present the Fermi architecture. Each SM is comprised of two groups of 16 streaming processors (SPs), registers, a SM-wide private cache (unaccessible by other SMs), special

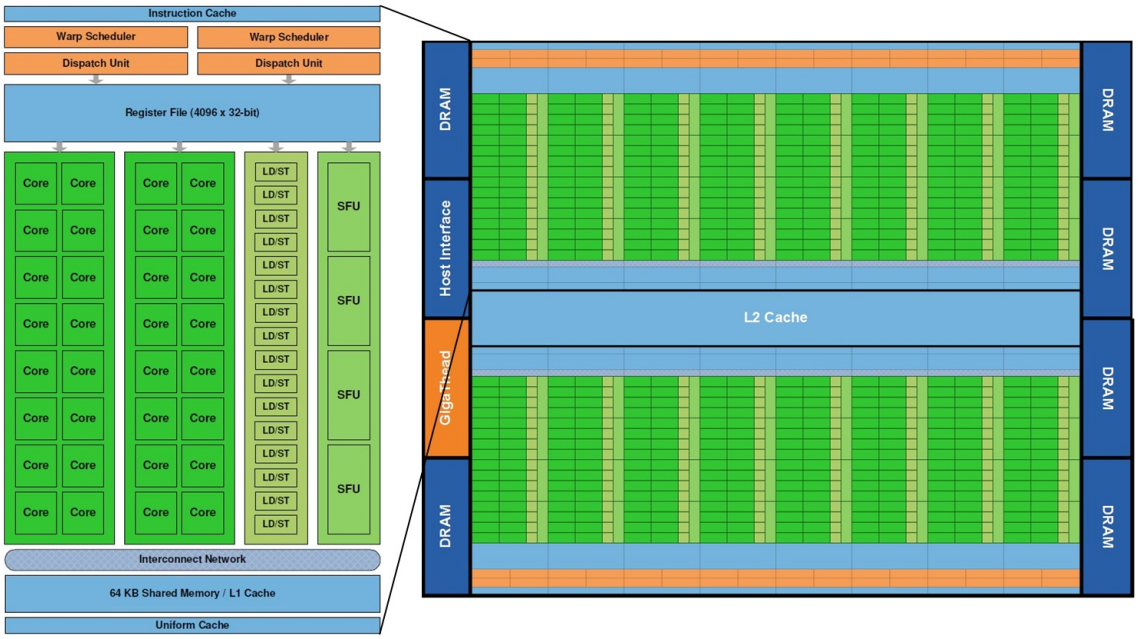

Figure 2.1: The Fermi architecture, showing a detailed streaming multiprocessor (edited from [NVI09]).

function units (SFUs) as well as load-store units. The SPs are responsible for integer and floating-pointer operations. The SFUs are responsible for transcendental functions such as sin, cosine and square root. The load-store units allows the implicit translation of memory addresses (cache or DRAM) for each thread within each SP group. All memory (cache and DRAM) is manually allocable and controllable by the programmer, enabling better optimized applications, in exchange of programming complexity.

The SMs operate using a SIMT (Single Instruction Multiple Threads) model, containing two warp schedulers within each SM. Each warp scheduler is in charge of a group of 16 SPs. A warp (or wavefront in AMD's documentation) is the indivisible scheduling unit (where each warp can contain multiple instructions), executed in groups of 32 threads in the Fermi architecture. Each single-precision float and integer operations are issued twice over one group of 16 SPs (to create the illusion of a 32-thread warp), double-precision operations are issued over two groups of 16 stream processors, occupying a full SM, temporarily suspending the execution of the other warp scheduler. The four shared SFUs, imply an $\frac{1}{8}$ throughput as the 32 operations (size of the warp) must be issued using only four units per cycle. Some considerations should be taken into close account regarding warps, when programming with GPU devices, as it is possible for these to diverge. This commonly happens due to conditionals present in the computation which implies that some threads are executing different operations of varying complexity. This incurs longer execution times as not all streaming processors might be used at each cycle, lowering the throughput.

To reduce DRAM access overhead (significantly higher than cache's), the Fermi architecture attempts to coalesce them. DRAM access is done using various fixed-size memory

transactions (32-, 64-, or 128-bytes), that must have their first address aligned to a multiple of its size. This entails that useless information is transfered if memory accesses that are not a multiple of these transaction sizes. Ideally, each warp should use all the data acquired by the implicit memory transactions in an ordered way (for example, thread one uses the first element, thread two the second element, and so on) to avoid unnecessary transfer overheads. Thus the programmer is responsible for parallelizing the computations in a way that takes advantage of coalesced accesses.

2.2 OpenCL

There are two main APIs for general programming on GPUs, the NVIDIA's CUDA [NVI08] and OpenCL [Khr12]. Brook [BH03] was the first framework of its kind to be created in 2003, demonstrating the potential of GPGPU in GPUs, using Cg [Mar+03], a C-like programming language which compiles to OpenGL shaders for GPU execution. CUDA was released in 2007, after the widespread of the GPGPU concept from Brook's creators and others. CUDA is limited to NVIDIA GPUs which created a need for an open standard. OpenCL was developed in 2008 by the Khronos Group to be a general computation standard for many heterogeneous devices, such as CPUs, GPUs and DSPs (Digital Signal Processors). A significant portion of hardware developers dedicated to computation adhere to the OpenCL standard as a unified access to processing resources. These developers include NVIDIA (where OpenCL is translated into CUDA), AMD and Intel.

2.2.1 Architecture

OpenCL serves as an abstraction to heterogeneous devices, thus exposing a similar architecture for any computing device. As seen in Figure 2.2 each computational device is comprised of an arbitrary number of *work-groups* (shown as *compute units*). Each work-group contains a set of *work-items* (shown as *processing elements*), and each is translated into single core with fast *private memory*. Each work-group has *local memory*, an internal memory shared by all elements within the group. Several work-groups can be used to for the computation of a single kernel which makes the management of the memory an important concern. OpenCL does not enforce any type of behavior in regards to global and constant memory cache, as well as does not provide a API to modify the underlying device's behavior (due to varying cache policies).

Table 2.1: Memory allocation and accessibility by Host and Kernel (adapted from [Khr12])

	Global	Constant	Local	Private
Host	Dynamic (Read/Write)	Dynamic (Read/Write)	Dynamic (No Access)	No Allocation (No Access)
Kernel	No Allocation (Read/Write)	Static (Read/Write)	Static (Read/Write)	Static (Read/Write)

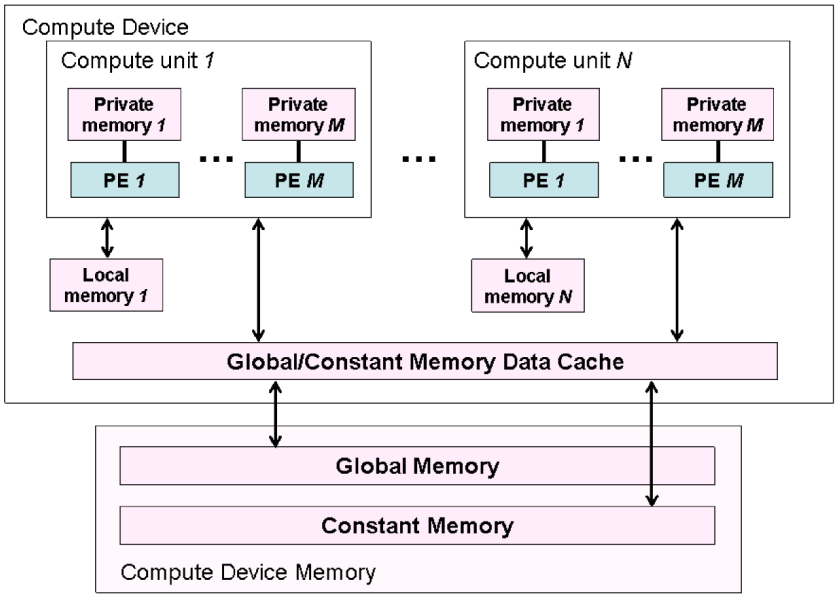

Figure 2.2: OpenCL device architecture, Host is not shown (taken from [Khr12]).

There are two categories of globally-accessible memory within OpenCL devices: *global* and *constant*. Constant memory cannot be changed, within a kernel execution, once set. The amount of available memory in any of these categories vary depending on the underlying hardware. All memory types can be explicitly managed by either the Host or the Kernel as shown in Table 2.1. To create efficient implementations in OpenCL, the programmer should manage these different memory types to optimize the locality of the data. The devices cannot access host memory (cache nor RAM) directly due to hardware limitations (although when using the CPU as a OpenCL device, it technically has access to host memory), which influenced OpenCL's design to treat each device as autonomous.

OpenCL has a relaxed memory model where the state of memory is not guaranteed to always be consistent over all the work-items. Private memory within a work-item has load and store consistency as it is the only one reaching it. Local memory is consistent across work-groups (and their work-items) when there is work-group barrier. Global memory is only consistent across the work-items of a single work-group at a work-group barrier, but there are no guarantees among different work-groups while executing a kernel.

2.2.2 Programming

A OpenCL program has two components: *kernels* that are executed in one or more OpenCL-enabled devices and a *host program* which triggers and manages the execution of the kernels. The kernels are written in a subset of the C language with some new primitives

(such as __kernel for the definition of a kernel function) and some limitations, such as being unable to allocate an array whose a size cannot be determined in compilation-time. As presented in the previous subsection in the architecture, each work-group is comprised of a variable amount of work-items. These are assigned an *index space* (which can have multiple dimensions), within a single kernel execution, creating a unique identifier by combining a work-group and work-item identifiers. When a kernel is executed the programmer can use these to create a disjoint work distribution among all the work-items with ease.

The host program interacts with the device using a *command-queue*, which can schedule the asynchronous execution of the kernels. There are three command categories: *kernel triggering* commands (to start execution), *memory* commands (which are used for allocation and transfer of data) and *synchronization* commands (which define constraints to the execution order of the enqueued commands). There are two execution orders: *in-order*, and *out-of-order*. In-order execution launches enqueued commands in the order they are submitted, but only one command can be executing at a time. Out-of-order execution does not wait for a command to finish execution and the programmer must enforce any order constraints using explicit synchronization.

OpenCL supports the following programming models: *data-parallel*, *task-parallel* and hybrids of the two. The data-parallel model is the most commonly used, where computation is defined using a sequence of computations over a data-set and there is an instance of each kernel per work-item. Using the aforementioned identifications, the work-items apply these computations to their assigned elements in parallel. It is possible for the programmer to specify the total number of work-items needed for an execution as well as how many work-items are grouped into a single work-group. Alternatively, the programmer can only specify the total amount of work-items needed and OpenCL can automatically infer the amount of work-items per work-group. The task-parallel model executes a single instance of the kernel independently of an index-space. Parallelism in this model is expressed by either using the vector data types implemented by the OpenCL device; queuing multiple tasks or by queuing native OpenCL kernels. This programming model is logically equivalent to executing a kernel using a single compute unit with a work-group containing one work-item.

GPUs support bi-directional communication while there is computations being executed in the device. This feature is commonly referred as computation and communication overlap. Programmers can take advantage of this feature by manually orchestrating the communication to and from the device, using carefully planned kernel execution commands with a technique known as *double-buffering*. Komoda et al. [KMN12] detail a OpenCL communication library which takes advantage of this feature to ease the efficient usage of OpenCL. They use a stream graph abstraction to describe pipelined parallelism, to know the precedences needed for the correct transfer scheduling. This library shows an improved speedup between $\times 1.1$ and $\times 1.6$ versus a non-orchestrated execution.

2.3 Taxonomy of GPU-Accelerated Computer Systems

The unending need for scalability evolved the initial architecture, containing a single computer with one GPU, into clustered solutions as well as more recently the integration of multiple GPUs in a single computer. Several architectures emerged from this evolution, which motivates the creation of this taxonomy, for their categorizations while taking into account common features. The architectures of GPU-accelerated systems can be categorized according to two dimensions: 1) the number of physical nodes composing the system and 2) the number of GPUs within a node. The Table 2.2 defines the taxonomy of GPU-accelerated architectures, including clustered solutions.

Table 2.2: Categories of GPU system architectures.

	One GPU	Multiple GPUs
One Node	SNSG	SNMG
Multiple Nodes	MNSG	MNMG

The SNSG architecture is the initial GPU accelerated architecture, containing a single machine and a single GPU device. It is the simplest GPU-accelerated architecture, being commonly the first framework development step, before its generalization to a clustered and/or multi-GPU solution. Stone et al. [Sto+07] use a SNSG architecture to accelerate the simulation of molecules by partitioning the simulation space among all the streaming processors in the GPU, obtaining a $\times 40$ better speedup than the best CPU implementation. Elsen et al. [Els+07] present a GPU-accelerated N-Body simulation which simulates the effects of a force law to a set of mass particles in a space. This implementation had a $\times 25$ improved speedup over a SSE-optimized CPU implementation. A dense linear algebra solver was implemented by Tomov et al. [Tom+10] using the MAGMA math library which enables several factorizations such as Choleski, LU and QR in a SNSG environment. An auto-tuning component is used to create a static schedule of the tasks between the CPU and the GPU. Mousazadeh et al. [Mou+11] presents a CUDA implementation of a deformed image registration algorithm, which uses a reference image and a deformed image to calculate the forces involved, which caused the deforming. This algorithm requires a significant amount of sparse matrix operations, thus making it a good candidate for GPU execution. This work can be applied to medical resonance images (MRI), to help the diagnose of illnesses.

SNMG architectures have simpler implementations in comparison to multi-node architectures, as there are no concerns relating to communication failure, while still providing a solution to increase scalability, albeit limited by the number of supported GPUs in a single machine. The usage of multiple GPUs is a recent development, being an active subject of academic research, to develop efficient and load balanced solutions to take full advantage of the locality of the resources, as well as optimization of the PCIe bus usage. Potluri et al. [Pot+12] use a novel feature in CUDA 4.1, which enable direct GPU-to-GPU communication interface, only using the I/O hub, without CPU interaction, for

inter-process communication, in a SNMG architecture. This work is integrated into MVAPICH2 [Hua+06] (an implementation of MPI for InfiniBand and other high-performance networks). Two types of communication are defined: Two- and One-way. Two-way communication entails synchronization between messages akin to send/receive primitives in distributed communication, while one-way communication is asynchronous and requires explicit synchronization. The work is compared to MPI using CPU interaction and results show a 79% latency improvement in two-way communication, and a 74% improvement in one-way communication. This paper is presented as a SNMG architecture instead of MNMG (as it uses MPI, commonly used in distributed computation) as the whole paper, as well as the evaluation presented was done using a single node. Lalami et al. [LBB11] developed a SNMG implementation of the Simplex method, commonly used in linear programming. The simplex method is represented using the matrix notation, which is partitioned equally among the GPUs. Test results using two GPUs show an 85% better performance versus single GPU execution.

Multi-node architectures presented below, rely on communication APIs, such as *Message Passing Interface* (MPI) and Java-based *Remote Method Invocation* (RMI) as a foundation for distributed computation. The main advantage of these architectures, is the higher scalability potential due to unbound number of nodes. However, there are some concerns inherent to this approach, mainly related to reliability, since latency, error detection and correction are not a trivial issue in a networked environment.

MNSG architectures are commonly used as trade-off between scalability and implementation simplicity, while maintaining unbound potential for scalability (at the expense of budget), as new nodes can always be added. Fan et al. [Fan+04] present an overview of the MNSG architecture and its advantages, such as cost-efficiency, in comparison to pure CPU clusters, as well as possible applications, for example, an airflow simulation in Times Square. When using 32 nodes, each equipped with a GeForce FX 5800 Ultra, presents an improved speedup of about $\times 5$ versus a CPU cluster, translating into about 80% of the (best) linear scalability. It should be noted that at the time of this paper, multi-GPU execution was not yet being studied and was still very premature at a hardware level (as SLI and CrossFire technologies were announced at the end of 2004). Several supercomputers have adopted the MNSG, some of which show exceptional power efficiency, as three of the top five most power-efficient supercomputers are GPU-enabled [Comst]. Sanam sits in the fourth place, with 2,351 MFLOPs/W, comprised of 420 nodes in a MNSG architecture with a AMD FirePro S10000 per node and is the 52nd fastest supercomputer [Meu+st]. The 29th most power-efficient supercomputer is the Titan, with 2,142 MFLOPs/W, made up of 18.688 nodes where each node has an NVIDIA Tesla K20 and is on the second place of the ladder as of the writing of this document.

The MNMG architecture represents the most generalized model. It has unbound scalability while maintaining a best price/FLOPs ratio, as it is possible to add additional nodes when the current ones cannot add more GPUs. The drawback of this architecture is the increased responsibility of the developer, as there are two different layers

of optimization required, one at the network layer and another intra-node, for the load balanced distribution among GPU devices. This complexity entail that most works using this architecture use straight-forward data partitioning on both layers. Danner et al. [Dan+12] present MNMG architecture with 32 nodes connected using MPI, equipped with six NVIDIA Fermi M2070 GPUs each. They use this architecture to accelerate the creation of a digital elevation model using a model similar to Map-Reduce. This work resulted in a $\times 25$ speedup over the pure CPU cluster. Babich et al. [BCJ10] adapt a library with several sparse matrix linear solvers for MNNG architecture by using data-parallelism over the whole cluster. The cluster contains 16 nodes containing two GPUs (NVIDIA GTX 285) each, connected by InfiniBand. The results show a strong scaling, improving about $\times 10$ sustained GFLOPs using 32 GPUs versus two. Additionally there two highly regarded supercomputers (aforementioned in the motivation), which take advantage of this architecture, namely the Tsubame 2.0 and the Ha-Pacs. The Tsubame 2.0 is 21st fastest supercomputer, as well as the 91st most power-efficient, comprised of 1408 nodes, where each node has three NVIDIA M2050s. The Ha-Pacs is the 62st fastest supercomputer, 48th most power-efficient, comprised of 1300 nodes, containing four NVIDIA M2090s per node.

2.4 Algorithmic Skeletons

Parallel execution is commonly obtained by explicitly coordinating threads and synchronizing access to shared resources. The *Parallel paradigm* is naturally more complex than the sequential counterpart and tends to be notoriously error-prone [Yan+12] due to unpredicted interactions among processing entities over shared resources.

To this extent, the identification of patterns that embed known structures and behaviors in this area provide an effective added value. Recurrent steps common to many algorithms can be optimized unitedly.

Algorithmic Skeletons are algorithm patterns for parallel executions which abstract the underlying structure of the execution environment [Col91], providing predefined memory synchronization and communication. The higher-level capabilities of skeletons enable the programmer to focus on the algorithmic problem instead of the concerns intrinsic to its parallel decomposition.

Skeletons can be combined in two ways: *sequential* (of which SkelCL [SKG11] is an example) and *nested* (of which Calcium [CL07], Skandium [LP10], Muesli [CK10] and Marrow [Mar12] are examples).

Combining sequentially exposes the explicit execution of one skeleton at a time. The algorithms code explicitly defines the sequence of steps and has access to all intermediate results. This approach gives high control to the programmer in exchange to the lack of execution orchestration from the frameworks. Thus, the programmer is responsible for inter-skeleton execution optimization, making efficient programming complex.

Nesting allows the combination of elementary skeletons to create complex execution

structures [Col04], creating the illusion that all skeletons are running concurrently, much like processes in an operating system. This illusion allows higher level orchestration of the independent skeletons, because the runtime execution is defined early and delegated to the library.

The systematic usage of skeletons is important as it separates program behavior from intrinsic parallel details [GVL10]. The separation enables performance gains, by taking advantage of the available infrastructure from pattern optimization present on the Skeletons.

Skeleton libraries were initially developed for cluster environments [CL07; ADT03]. In this context there was a need for communication technologies over a network between computation nodes. Among these technologies the most used are RMI (Remote Method Invocation) in the Java context and MPI (Message Passing Interface) standard in several other languages. These technologies provide communication support for distributed computing over a network.

With the development and proliferation of multi-core architectures, new Skeleton frameworks were specially tailored or adapted for these environments, such as Skandium [LP10] and Muesli [CK10]. These typically take advantage of intra-node parallelism by resorting to OpenMP for example.

More recently with the growing popularity of GPGPU, several Skeleton frameworks were proposed to exploit this untapped resource such as SkelCL [SKG11] and SkePU [EK10] which take advantage of parallelism within multiple GPUs. Since our work revolves around this research, we will provide a more in-depth description of such libraries, as well as other approaches, in the remainder of this section.

2.4.1 Algorithmic Skeletons for GPGPU

SkelCL

SkelCL [SKG11] is a C++ library for GPGPU built on top of OpenCL. Its main objective is to provide performance gains in highly parallel tasks using GPU devices.

The main concept of this library is the *Vector*. This data-structure was created to bypass a significant portion of code-overhead present on GPGPU APIs.

All data transfer between host (CPU accessible) memory and device (GPU accessible) memory are optimized by the Vector. This feature includes lazy copying which delays memory transfer to host memory until explicitly invoked, or all computation depending on it ends, enabling a Skeleton to use results from a previous one without redundant transfers to host memory.

The instantiation of the skeletons with computation is performed by supplying C++ functions as plain strings.

SkelCL supports four skeletons: Map, Zip, Reduce and Scan. All these consume input Vectors and produce output Vectors making them compatible with each other. This compatibility is not nesting as multiple skeletons cannot execute concurrently. It simply

enables the sequential composition of skeletons, in the sense that the output of one may be passed directly as the input vector of another.

Map applies a programmer-defined function to all elements of a given Vector. For type-preservations sake, the map function must take an element of a certain type T and produce an element of the same type.

Zip takes a programmer-defined binary operator, which is applied index-wise to a pair of input Vectors of the same length, and returns a single output Vector with the results of the combination.

Reduce applies a binary operation to every pair of input elements recursively until it obtains a single scalar value. For parallel execution of this Skeleton, the binary operation must be associative so it can be applied to arbitrarily sized subranges in parallel. It is the programmer's responsibility to provide an associative operator as SkelCL does not test this property. Intermediate reduce results are saved in fast, local device memory (cache).

Scan applies a binary operation recursively to all previous elements of the element being calculated. It is similar to a reduce with the input Vector containing all the previous elements, including the one being calculated. This Skeleton is optimized to make use of local memory and avoid memory access conflicts.

Some algorithms might require a variable number of arguments on the customizing function (such as Map). In SkelCL, the *Arguments* object allows an arbitrary number of input arguments, by wrapping them like a container which enables implicit memory management between host and device memories.

One premise of OpenCL is portability across several device types, such as CPUs and GPUs. To that extent, it is necessary to compile the kernels at runtime by the device at least once. To avoid unnecessary compilation overhead, SkelCL provides the option of saving the compiled kernels in persistent memory for posterior use.

Multi-GPU Support

More recent work on SkelCL [SKG12] added multi-GPU support by defining the distribution of input Vector over several devices. The distribution and access to each data section is automatically managed by the library.

The Vector concept remains the same. Internally however, it was adapted to support distributions among different devices while maintaining the memory operations in a transparent fashion.

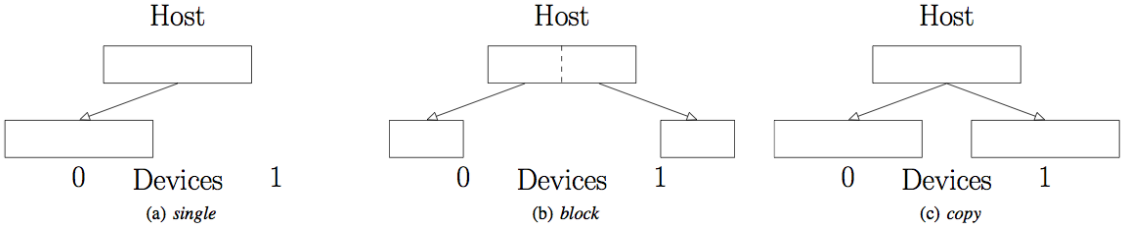

Figure 2.3: Vector distributions supported by SkelCL (taken from [SKG12]).

Currently there are three possible ways to distribute work amongst multiple GPUs: *Single*, *Block* and *Copy* (Figure 2.3). Single transfers the whole vector to a single device. Block splits the input vector equally among all target devices. Copy duplicates the entire input to the all devices, being the output merged at the end of the skeleton execution by a programmer-defined function. If such function is not specified, then the first result from a copy received will be accepted and all others will be discarded.

A Vector's distribution can be modified at runtime either by the system or by the programmer explicitly. This operation is, however, constrained to the execution states, when there is no computation taking place in the GPU devices. All necessary memory transfers will be done by SkelCL implicitly. These are done lazily, only transferring if they are actually needed for the next computation. This reduces communication overhead but forces computations to block until the data-set partition is completely transferred.

The default Vector distribution depends on its role in the Skeleton computation. When it is a main input, the skeleton defines the defaults. On the other hand, if it is an additional input, the default distribution must be defined by the programmer using a customization function.

The developed Multi-GPU support imposed unavoidable adaptations to the implementation of the previously available Skeletons.

Map and Zip suffered similar modifications to take advantage of data-parallelization in a GPU device level. Zip, in particular, has an extra requirements: both input vectors must have the same distribution and if the distribution is set to *single* both vectors must be in the same device. Otherwise, the system will automatically change the distribution to *block*.

The Reduce Skeleton implicitly performs all orchestration required to perform a reduction on multiple GPUs. Each device performs a partial reduction locally. Afterwards the intermediate results are transferred to the CPU where they are further reduced to a single scalar.

Scan requires a more complex approach to be able to execute in multiple GPUs. Contrary to all the other supported skeletons, there is a prefix dependency on previous elements. When the vector is distributed, a local scan is executed on all partitions. From these intermediate results, only the first partition is correct as it has all the elements it depends on. To obtain the correct values on the subsequent partitions, the last element of the previous partition is transferred to the host memory where a Map skeleton is implicitly created. This map applies the value to the current partition the depends on it. This procedure is applied iteratively until the last partition is reached and all dependencies resolved. A common example of this Skeleton is the prefix-sum where each vector element is the sum of all its precedents in the Vector.

SkePU

SkePU [EK10] is a C++ Skeleton template library that supports the compilation to multiple back-ends such as OpenMP, CUDA and OpenCL.

To hide the verbose memory operations required by both CUDA and OpenCL programming, SkePU uses the *Vector* concept. The Vector is similar in all aspects to the one presented in SkelCL. This mechanism does not take explicit advantage of communication and computation overlap that CUDA streams support as the communication is delayed until it is needed by the host.

Skeleton parametrization is expressed through C++ macros which delimit the types of supported behaviors. These are translated to a structure with all the information needed for skeleton execution. The macro list offers: overlap, array and the elementary macros unary, binary and ternary, all with a constant variant that does not allow data reassignment.

The skeletons supported by SkePU are: Map, Reduce, MapReduce, MapOverlap and MapArray. These are accessible as a C++ function and use the cited macros as arguments.

Map and Reduce are similar their SkelCL counterpart. The particular case of Map allows the use of a ternary function, enabling the use of three input vectors.

MapReduce is a conjunction of both Map and Reduce Skeletons that are used to simplify code considering it is a very common programming pattern. It takes both Map and Reduce function structures as arguments on initialization.

MapOverlap is a variant of Map where the computation of each position of the result Vector depends on a range of elements of the input vector. An useful example of this Skeleton is the convolution algorithm, commonly used for image filters.

MapArray applies a binary function with two input vectors. Each element of the resulting Vector depends on the element of the second input vector at the same position, and an arbitrary number of elements of the first input vector.

Multi-GPU Support

Multi-GPU support in SkePU was developed by dividing the work load among the existing devices as well as auto-tuning of the distribution dimension parameters to take full advantage of heterogeneous environments with acceptable performance.

To implement these new features the execution plan [EDK10; DEK11] feature was created. It enables the dynamic specification of which back-end to use considering the actual problem size, as well as provide the default parameters for each back-end. All skeletons include this feature and support their manual parametrization.

A prediction framework is used to supply default auto-tuning values for the execution plans. The mechanisms applied by the framework is twofold: first, at installation time, a set of micro-benchmarks are executed on the devices and secondly a heuristic algorithm based on genetic programing to calculate the best execution plan for a single

back-end. The estimation of different devices capabilities available that are mainly obtained on installation using micro-benchmarking. The data gathered by both these stages are combined to create an execution plan prepared to run the Skeletons in any back-end with default values to each.

The heuristic algorithm attempts to find the best parameter values for a certain back-end. These vary from each back-end, such as "Number of threads" in OpenMP, "Block Size" as well as "Grid Size" for OpenCL or CUDA. Initially there is a predefined set of initial configurations from which the algorithm will benchmark, to try to deduce the best configuration. At each configuration entry, the benchmarking kernels are executed to evaluate the configuration's quality. This algorithm ends when all the configurations have been exhausted. To reduce the overhead this approach entails, there is a threshold parameter which stops the algorithm when it reaches a certain degree of optimality.

The Micro-Benchmarking is twofold: measurements are taken during installation and the same measurements are taken after an interesting real-world kernel execution to improve future estimates. Depending on the back-end, different parameters are saved. The values of interest for GPUs are: data transfer time from host to GPU and vice-versa; kernel execution time, and total execution time (which should be about the sum of the previous, excluding possible architectural overhead). A formula is used to identify repetitive and fixed time. Repetitive time is measured by executing a skeleton over vectors present in host memory, as to include the data transfer overhead. Fixed time is measured by executing a skeleton over the same vector, present in GPU device memory, avoiding data transfer overhead.

Recent work over the SkePU [Kes+12] built an integration with StarPU [Aug+09], a library that simplifies parallel execution by automating the runtime load balancing. This library is presented in detail in Section 2.6.

Muesli

Muesli [CK10] is a C++ template library that takes advantage of distributed and multi-core environments. Message Passing Interface (MPI) is used for inter-node parallelization, OpenMP and CUDA take advantage of intra-node parallelization.

Muesli handles all distributed data-structures implicitly. These structures synchronize automatically (using MPI) when non-local data is needed, enabling a higher-level programming experience. These structures are built akin to Sparse Matrices and can be distributed in diverse ways (for example Block and Round-Robin [CPK09]).

The skeletons supported by the library are: Pipeline, Farm, DivideAndConquer, BrandAndBound, Fold, Map, Scan and Zip. From this set, only Fold, Map, Scan and Zip support GPU execution.

The usage of Multi-Core CPUs is tackled using OpenMP. A OpenMP Abstraction Layer was built to wrap all the functions from OpenMP over Muesli library functions with safe-guards in the event OpenMP is not available (using the _OPENMP macro test).

The OAL provides cleaner code in regards to the reduced macro usage and management.

To take advantage of the GPGPU environment Muesli uses CUDA. A data-structure akin to SkelCL's Vector, the *Device Vector* (DevVector) abstracts the memory operations between the host (CPU) and the device (GPUs) that CUDA entails. DevVector handles all memory transfers as well as their optimization transparently using lazy copying. The Device Vector inherits the communication and computation overlap optimizations provided by CUDA streams but there is no explicit mechanism to enable the programmer to control such feature.

To provide multi-GPU [EK12], each device vector contains a set of execution plans structure, which contains all the pointers to the data-set of each skeleton instance. Data partitioning is done manually by the programmer and assigned to each execution plan entry on the skeleton. It is impossible to obtain additional information, due to the lack of architecture and implementation information in either the published papers or the source-code available (which lacks GPU execution altogether).

Source-To-Source Compilers

Source-to-source compilers in the GPGPU context are used to translate programmer-annotated, single-threaded code into a multi-threaded environment. In this context, the PGI compiler [The10] of the OpenACC [CAP] is the most noticeable work. Our discussion, however, focuses on the use of algorithmic skeletons to convert single-threaded algorithms to known multi-threaded structures. Even though the programmer has to annotate the code while taking into account the desired parallelism, he does not have to be aware of the parallelizing library (such as OpenCL or CUDA).

MultiSkel [Le+12] compiles annotated C++ code into a CUDA-compliant source code (containing both host and device sources) for multi-GPU execution. The workload is split evenly by all GPUs, as the supported skeletons are mainly data-parallel. The supported elementary skeletons are: Map, Reduce, Scan and ZipWith. MapReduce and ZipWithReduce are also implemented as a Skeleton even though they are combinations of the elementary skeletons. Map, Reduce, Scan and ZipWith are analogous with the skeletons presented in SkelCL previously, where ZipWith is equal to SkelCL's Zip. Skeleton nesting is not supported.

Bones [NC11; NC12] translates annotated C source-code to CUDA or OpenCL. Diverse classes of mathematical operations are skeletons, with a mathematical-like syntax, using an abstract syntax tree (AST) grammar. These operations are commonly applied in a wide array of image operations and filters such as pixelization, convolution, erode and color histogram. Multi-GPU is not yet supported, however, Bones is an interesting framework as most supported operations are stencils, where a single element requires the bordering elements for its computation, which entails shared-data management. The

border's width is defined by the programmer, considerably increasing the system's flexibility. To uphold an appropriate distribution of work, the compiler resorts to a performance prediction model associated to each operation class, to decide which architecture should be used for the computation. The system is statically defined, as the analysis of each operation is done manually by the creators of the framework, taking into account the data-access patterns to predict the complexity. Skeleton nesting is not supported.

2.4.2 Other Approaches

There are other approaches to the abstraction of multi-GPU computation apart from algorithmic skeletons, such as *codelets*, *dataflow processing* and *stream processing*.

Codelets are abstractions which break the code functionality into smaller fragments [Zuc+11]. There are some restrictions that should be followed when implementing an execution model using codelets: these should not block or run indefinitely; they should be indivisible and atomic; they must explicitly define the data that will be accessed and modified and should have low-overhead when hiding long-latency operations (passing results directly to another codelet). StarPU [Aug+09] takes advantage of this abstraction to create functions that might contain several different versions targeting different architectures while maintaining the same functionality. This library is presented in detail in Section 2.6 due to its load balancing capabilities on multi-GPU systems.

Dataflow Processing [LM87] is based on a computational graph abstraction, where the nodes are computation kernels called *filters* and the directed edges are a pipelined description of the data path. Transforming the graph for parallel execution can be twofold: 1) partitioning the graph into relevant groups to minimize the memory transfer overhead and 2) data parallelization within a single filter. Boulos et al. [Bou+12] present a dataflow programming model which allows the specification of the architecture graph where the program will be executed. The programmer can then specify a data flow program by defining the filters and the computational graph in a textual form. Each filter can be manually assigned to each device described in the architecture graph without concerns for the underlying architecture.

Stream processing [Kha+03] is an enhancement of dataflow processing, where each edge becomes a FIFO queue and the execution becomes demand-driven, where the dataset is not predefined to the system. This programming model is very suited for multimedia processing, as shown by Repplinger et al. [RS11], who present a distributed system on top of a multimedia middleware which uses GPU resources implicitly to apply filters to a stream. StreamIt [TKA02] is a stream processing language that was initially created for multi-core architectures and more recently adapted to multi-GPU execution using CUDA [Huy+12; Hag+11]. This adaptation partitions the graph for a coarse-grained parallelism while satisfying the memory constraints of the devices and minimizing the inter-device communication. Dubach et al. [Dub+12] present Lime, a programming language compatible with Java, which targets general purpose processors, FGPAs and GPUs. The

connect operator (=>) enables stream processing of an arbitrary number of filters. Filters are automatically optimized, where the compiler attempts to find kernel-level data-parallelism. As GPU has several types of memory, the memory assignments in Java suffer a transformation where the compiler identifies several types of memory and maps them to the GPU memory hierarchy (e.g. private arrays will be assigned to fast, private memory). Lime does not support multi-GPU execution, however, it is a cornerstone of stream processing, as such, we felt that it should be present in this overview.

2.4.3 Critical Analysis of Algorithmic Skeletons for GPGPU

Skeleton libraries are relevant for the multi-GPU context as they enable the programmer to acquire predefined, parameterizable structures using a familiar language. These structures have the same degree as expressiveness as Lime's connect (=>) operator, without the need to learn a new language and its quirks, by enabling the submission of user-defined operations and data-sets.

Although the benefits of algorithmic skeleton libraries are evident, the multi-GPU libraries available are all still under active development, as it is a novel research topic. The computations offered to the programmers are severely limited in all systems, only offering data-parallel skeletons. SkelCL enables a very accessible approach for GPGPU, due to its Vector concept and well-defined Skeletons, at the expense of very limited programmer adaptability of the underlying system. The multi-GPU support is not suitable for heterogeneous devices, as the data-set partitions are blindly set, lacking an auto-tuning component. SkePU defines a programmer interface using a strict number of macros, which limit the number of arguments the programmer can pass to a skeleton. Multi-GPU execution, is able to support heterogeneous devices efficiently as it takes advantage StarPU (detailed in Section 2.6.2), which uses several variations of work-stealing techniques.

2.5 Auto-Tuning

Auto-tuning [EAH77] is the automated optimization of parameters which influence performance. In the multi-GPU context, auto-tuning is an important component because of the wide variety of GPU architectures and the compatibility between them. Multi-GPU systems can easily integrate heterogeneous devices on the same system, which inherently brings imbalanced work distributions when no tuning is taken into account. There are two main approaches for GPU tuning: 1) benchmarking, relying heavily on empirical data and 2) mathematical performance models, that rely on describing the architecture using equations and use minimal empirical data.

Benchmarking is the most common procedure to obtain empirical performance information of the target devices. If the target architecture is known, the assessment of the GPU hardware specifications narrows the sensible parameter range to benchmark. As multiple factors can affect performance and due to a degree of uncertainty on how

each hardware architecture will behave, auto-tuning systems using benchmarks, typically have two development stages.

Initially the manual tuning of a specific algorithm is used to survey the possible performance gains. The survey involves running a relevant set of execution configurations and manually identifying the best. Volkov and Demmel [VD08] present the optimization of several dense linear algebra factorizations such as LU, QR and Cholesky using this methodology. The GPU computations, implemented in CUDA, were developed taking into very close account the specifications of the target NVIDIA GPU hardware. This approach reports a 80–90% improvement of the peak FLOPs in large data-sets. Gu et al. [GLS10] present a CUDA multi-GPU 2D/3D FFT implementation using empirical tuning. The development, like in the previous example, very close attention to the target hardware's specifications, such as shared memory size within a work-group and coalesced memory access. Additionally, fine-tuning the amount of concurrent kernels is an important exercise, to avoid too many accesses to slow, global memory, while attempting to fit all the relevant work-set inside the shared work-group memory. The work is compared with NVIDIA's FFT implementation, CUFFT. The reported results show a $\times 2.8$ speedup in 2D FFT calculations and a $\times 22.7$ in 3D calculations relative to CUFFT.

The second development stage is the generalization to an automated system, which adapts the workloads while taking into account the devices capabilities. The Rodinia [Che+09; Che+10] benchmark suite targets multi-core CPUs and GPUs alike, using a wide range of algorithms with distinct data access patterns. This suite focuses in evaluating the impact of subtle architectural decisions made on hardware development by allowing each benchmark to have multiple implementations. Danalis et al. [Dan+10] created the SHOC Benchmark suite that provides predefined benchmarks to evaluate raw device capabilities and quantify the performance of real-world applications, using several popular algorithms such as Fast Fourier Transformation, Reduction and Scan. SkePU, presented in Section 2.4, uses a component that runs micro-benchmarks at installation-time, to determine a set of default values posteriorly used to create workload configurations for later executions.

Another approach to auto-tuning is the prediction of the device's performance, using mathematical models. It uses a minimal amount of benchmark information, to predict an approximate performance value of a device with a certain data-set size. The aforementioned works do not use this approach as they do not try to create a mathematical model which describes the relations between components and their impact, instead, they tend to focus mainly on the GPU's raw performance. Schaa and Kaeli [SK09] present a mathematical model which takes into account several variables such as: transfers speeds between CPU and GPU, RAM access speed, disk throughput and times spend on CPU and GPU executions. CPU and GPU execution times are inducted by extrapolating the time spent on a single item to the size of the data-set. The presented prediction framework shows an 11% average deviation from real performance values. Zhong et al. [ZRL12] developed a hybrid performance model for both CPUs and multi-GPU systems integrated

with data-set partitioning. This formulation is more relaxed as the functions do take into account more empirical information to modulate the performance functions. The performance model is used to assign adequate workloads to both CPU and GPUs by taking into account their performance, the problem size and communication overhead (while taking into account communication and computation overlap). The overlap results show a 30% improvement in operation speed (measured in GFLOPs) using a single NVIDIA GTX 680.

Auto-tuning is closely coupled to *static scheduling*, detailed in the next section, by providing a device-aware work distribution. It can also be used with *dynamic schedule*, by helping the runtime algorithm to decide how much work can a device handle and thus enable it to distribute adequate amounts of work to each processor, lowering the overall execution time.

2.6 Work-load Distribution for Multi-GPUs

2.6.1 Static Scheduling

Static scheduling [CK88] deterministically schedules tasks over a completely known architecture. This scheduling type is mainly applied on purely data-parallel programming models where the framework has complete knowledge of the data-set and the computations that will be executed. SNMG frameworks tend to take advantage of this type of scheduling as the architecture is known. Together with auto-tuning, static scheduling becomes capable of assigning tasks to the devices with the most adequate capabilities. If the auto-tuning adapts the scheduling over time with empirical data, the scheduling is called *adaptive static scheduling*. Maestro [SMV10] uses this adaptive scheduling and will be presented in detail below.

SkelCL [SKG12], detailed in Section 2.4, applies a static partitioning strategy without any kind of tuning, offering only predetermined distributions such as transferring the whole data-set to a single device, partitioning the data-set equally to all devices and replicating the data-set to all devices. This is a very simple implementation for strictly data-parallel executions, though not appropriate for heterogeneous systems.

StreamIt, presented in Subsection 2.4.2 is a streaming language with support for SNMG architectures. Multi-GPU support resorts to static scheduling with auto-tuning, using benchmarks in the compiler. The auto-tuning defines a heuristic based on the capabilities of the GPU devices, specifying the number of parallel stream executions, the number of computing threads within each stream and the number of memory threads used to prefetch global memory, minimizing the access overhead. The stream graph, representing a StreamIt computation, is partitioned for coarse-grained parallel execution and each partition mapped to each GPU with a coarsening-uncoarsening algorithm which minimizes communication among devices.

Qilin [LHK09] is a programming system which allows the programmer to build CPU

and GPU hybrid applications using a single, architecture independent source-code. It uses a compiler to convert this code into TBB (Thread Building Blocks) and CUDA, and automatically map it over the processors. This mapping uses a database which keeps track of the empirical performance values of every run. When a program is ran for the first time, a training run is triggered and the data-set is partitioned evenly between the CPU and the GPU. It is then partitioned further within both to create enough elementary computation runs to create an average of the duration. Using these averages, Qilin uses a mathematical model to infer a performance curve which will be used to predict the best mapping given a different data-set size on the next run.

Maestro

Maestro [SMV10] is a library that eases OpenCL usage and specializes in the automatic fine-tuning of application. For that purpose, it resorts to static scheduling and auto-tuning to obtain the best data-set partition for a given environment.

To auto-tune the OpenCL kernel execution, Maestro uses two components: measurements taken during the installation process and runtime profiling information.

The measurements taken during installation, attempt to define the best initial group size, buffering chunk size and workload partitioning for the current environment. The SHOC Benchmark Suite [Dan+10] is used to obtain initial performance values of the devices such as peak FLOPs as well as device memory bandwidth and then translate these values to an initial configuration.

Every time a kernel is run, whether at install-time or at normal runtime, there is a weighted average associated to each combination of *device* and *kernel* which measures the rate at work is done. This weighted average is used to adapt the workload partitioning over time with the aim of improving the framework's overall performance.

Maestro takes advantage of the overlap of communication and computation to optimize the usage of the bus and minimize the data-transfer overhead. The results from the overlap tend to depend heavily on the algorithm used as well as on the hardware. In the vector outer product test, it improved the execution time 40% to 60% less time to execute when using large chunks.

The multi-GPU execution on Maestro shows a load balanced work distribution which shows an improved speedup between $\times 1.6$ and $\times 1.8$ against its imbalanced counterpart.

2.6.2 Dynamic Scheduling

Dynamic scheduling [CK88] is useful when the system architecture and processor assigned to a certain data block is not known before-hand. Scheduling is usually distributed by the multiple participating processors, which coordinate to acquire actively work from one another, instead of work being delegated to them by the framework. *Load-balancing* is closely related to dynamic scheduling, creating evener workload among the participating threads, so that their execution finish at the same time. In multi-GPU

frameworks, dynamic scheduling is adequate for task farms where tasks are dynamically submitted for computation (i.e. the framework cannot predict when a computation will be submitted).

The most common policy of load balancing is *work-stealing*. It is commonly implemented by associating to each processor a task queue of tasks waiting to be executed. When a processor ends all the work on the respective queue, it will steal work from another processor's queue. Cilk [Blu+95] was the first system to incorporate a work-stealing technique in a runtime scheduler. A task dynamically generated by the application, through the *spawn* construct, is encapsulated in a *closure* that defines all the information needed for a thread to execute the computation. Data dependencies are identified in a tiered acyclic graph where the lower tier cannot be computed until the relevant higher-tiered closures have been executed. StarPU (which will be presented in the next section) shows the several variations of work-stealing for runtime load balancing. CudaZero [CCZ12] is a compiler which adapts CUDA source-code designed for SNSG into SNMG architectures. Work-load distribution is performed by a mix of static scheduling and dynamic scheduling among the multiple GPUs. Then each partition is further decomposed into work-tasks and added to a task pool where a work-stealing mechanism is used for the dynamic scheduling of the workloads. This compiler can only be applied to the most common usages of kernels, where the data-set is explicitly defined, because automated work distribution is not a trivial problem for all kernels.

Cudasa [Str+08] is a programming language that supports MNMG architectures using CUDA and MPI. It uses a variation of the work-stealing technique to maintain a global pending-work queue, instead of one per processor. Thus there is no actual work stealing from other processors, just acquiring new work-sets from the queue concurrently against others.

Binotto et al. [Bin+11] present a CPU-GPU hybrid framework to solve systems of equations. This framework provides a load balanced execution using a first-assignment scheduler and a runtime scheduler which adapts over time using empirical information. The first-assignment scheduler assigns a task without any empirical performance information. It uses a predefined cost function, which is dependent on task costs, as well as a performance value associated with each processor. The runtime scheduler uses empirical performance values in a database, collected by a profiling component, which keeps track of all the execution times. It is possible for the scheduler to reassess all the processor assignments of tasks waiting to be executed.

Acosta et al. [Aco+10] present a MNMG framework that focuses on a single problem entitled "Resource Allocation Problem". The definition of this problem is the load balanced allocation of an arbitrary number of indivisible data elements to an arbitrary number of predetermined heterogeneous processors for execution. It uses adaptive autotuning over time to achieve an approximately load balanced solution, starting by dividing the work equally to all processors and on the successive iterations adapt the amount of work proportional to the previous execution performance.

StarPU

StarPU [Aug+09] is a library that offers a unified interface with implicit parallel execution in several architectures such as multi-core processors, OpenCL devices and NVIDIA GPUs (using CUDA) as well as clusters of these (using MPI) by abstracting heterogeneous details. It offers primitives to define task dependencies, priorities and weights using a task graph. Dynamic scheduling with heterogeneous devices are also supported as well as automated data transfers in the cluster context for implicit data availability.

The primary data structure is the *codelet*. This structure describes the computational kernel that can possibly be implemented in one or more architectures.

A task is a wrapper for the *codelet* which describes the data-set it uses and how it is reached during computation (read and/or write). Priorities and weights are also optionally defined in the task data structure.

A performance model in StarPU is a structure which contains values that describe the potential performance of the current computational environment [ATN09]. It is possible to define a performance model in StarPU in several ways: providing performance values manually, running benchmarks present in StarPU before-hand, using runtime execution values or by adapting the scheduling dynamically during the execution. It is also possible to use a combination of the previous methods, for example, defining initial performance values using benchmark values, and then use runtime execution values as a feedback-loop to provide fine-tuning to the performance model.

The performance models are used by the scheduling algorithms to determine the workloads by determining dynamically the workloads of each device hence load balance the system. These scheduling algorithms can be implemented using *push*/*pop* constructs. The predefined schedulers provided by StarPU are:

- `eager`: uses a single task queue from where each worker takes tasks;
- `prio`: uses also a single task queue but orders the tasks by priority;
- `random`: each device/processor is assigned an acceleration factor that describes the respective computational potential. Every time a task is submitted for execution, a worker is picked with a probability proportional to its factor;
- `ws` (work-stealing): schedules tasks to workers. When a worker finishes its work, it steals tasks from the worker with most tasks left;
- `dm` (dequeue model): uses tasks performance models to supply an estimated task duration, and schedules tasks to processors by minimizing the total estimated duration;
- `dmda`: dequeue model which takes into account data transfer duration;

- `heft` (heterogeneous earliest finish time): uses work volume estimations supplied by the user in the task graph, as well as task duration estimations from the performance models to minimize the estimated execution time.

`Heft` and `dmda` allow the prefetching of the data-set associated with each task to its processor (i.e. uploading the data-set to GPU before computation is triggered), as the task scheduling is defined early in the execution, and there is not a work-stealing mechanism.

The StarPU architecture decouples the scheduling algorithms and tasks from the types of workers. This decoupling enables the support of new architectures by simply implementing a new device driver. Adhering to this driver interface guarantees the interoperability with StarPU.

2.7 Final Remarks

Clearly, the multi-GPU support in a SNMG architecture is still premature from libraries, specially algorithmic skeleton ones. These present significant limitations, whether in the possible computations available, which are mainly data-parallel, or in their usage of auto-tuning and workload scheduling techniques.

As our work requires auto-tuning and workload scheduling in a multi-GPU context, an overview of their state of the art was presented, illustrating their tendencies. The auto-tuning techniques are dominated by benchmark-based performance profiling, while static scheduling tends to take advantage of the auto-tuning component to adapt the sizes of each data partition assigned to each GPU, and dynamic scheduling tends to use auto-tuning to improve the performance of work-stealing techniques, by defining priorities assigned to each GPU based on their computational capability.

3

Marrow

Marrow [Mar12] is a C++ Skeleton library tailored for the construction of complex GPU computations by orchestrating the structure and execution of OpenCL kernels. Skeleton nesting is used to achieve complex GPU computations, by structuring multiple algorithmic skeletons, while abstracting their execution from the programmer. Current algorithmic skeleton frameworks for GPGPU, focus on data partitioning and distribution, while Marrow focuses in communication and computation orchestration. This focus shift enables the introduction of completely new skeletons, such as Stream, Loop and Pipeline, besides the more common Map (and variants). These new skeletons are important as they enable transparent GPU optimizations to commonly used structures.

3.1 Architecture

Marrow's architecture, shown in Figure 3.1, comprises four main layers: *User C++ Applications*, *Skeleton Library*, *Runtime* and *OpenCL-Enabled Device*. The layers has a downward dependency, being that each layer only has vision of itself and the one beneath. The User Application layer represents the programs which take advantage of Marrow and only the Skeleton library is exposed to it, for the creation of computation trees as well as triggering their execution. The *OpenCL-enabled Devices* represents the GPU device which Marrow uses.

Skeleton Library This layer provides the applications the constructions required to create a complex, skeleton-based, computations, such as the Skeleton implementations. It comprises the skeleton implementations that will be detailed in Section 3.4, the KernelWrapper as well as the *Kernel Data-Types*. The KernelWrapper is used to wrap an OpenCL

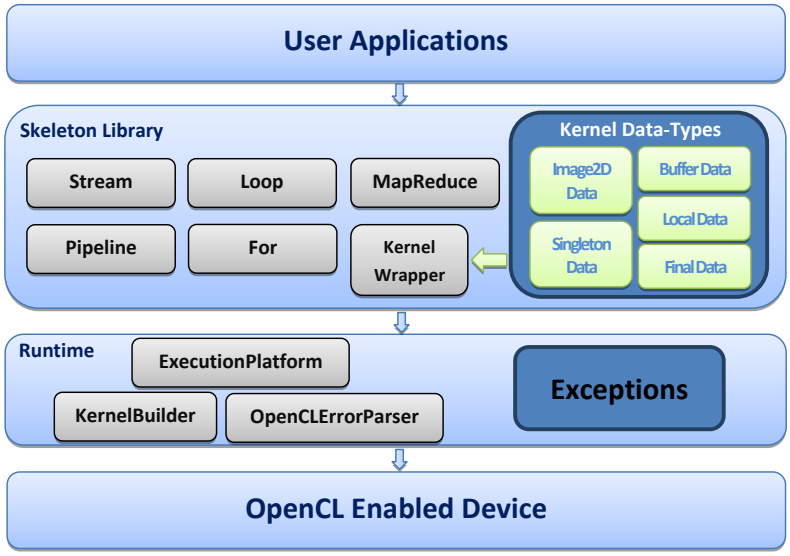

Figure 3.1: Marrow's architecture (taken from [Mar12]).

computation (kernel), providing a usable interface by all the skeletons, as it is the computation unit to which these apply their execution behaviors to. The kernel data-types are components used by Marrow to define the OpenCL kernel arguments within the KernelWrapper, by detailing their order, type and size, as well as enabling the transparent allocation of the memory required by the computation.

Buffer represents a contiguous memory region.

Image2D offers two dimensional space indexing, commonly representing an image, which instead of being stored in global GPU memory, is stored in texture memory.

Singleton is a single-element data-type whose value is defined when an execution is requested.

FinalData is constant single-element data-type, defined when the KernelWrapper is being defined.

LocalData preallocates a memory region in local GPU memory, which is guaranteed to be available when a computation is requested.

Runtime This layer is used to obtain the resources required for execution on the GPU devices, such as GPU memory and command queues, using the ExecutionPlatform for example.

This layer's purpose is to simplify many complex and verbose operations which OpenCL entails. It provides the upper layer the resources required for OpenCL computation using the aforementioned ExecutionPlatform, the OpenCL kernel compilation (using the KernelBuilder), as well as error detection and handling, by translating them to C++ exceptions (OpenCLErrorParser and Exceptions).

3.2 Execution Model

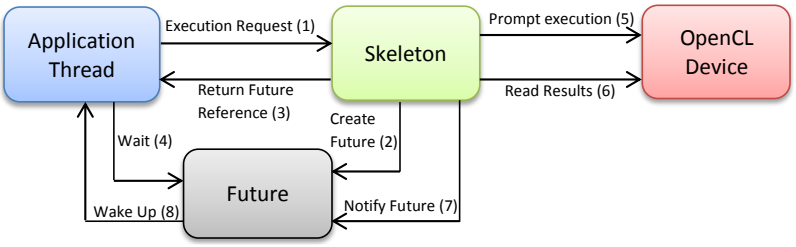

Figure 3.2: Marrow's execution model (taken from [Mar12]).

The execution model used by Marrow, shown in Figure 3.2, is similar to the one presented in Skandium [LP10] where the *Future* concept is used to represent the computations that might not have yet finished. When a skeleton execution is requested (with the input data defined, shown in step 1), a Future object is created (step 2) and data is sent to the GPU device for execution (omitted). The reference to the Future is returned to the application thread (step 3). Its interface offers the programmer the possibility to block the execution of the invoking thread until the result is ready (step 4), or just use a non-blocking query of its state to attest the latter's availability (omitted). The execution is then triggered on the OpenCL device (step 5), and when all the enqueued transfers and computations finish, the output data is read to host memory (step 6) and the Future object is subsequently notified (step 7). As a result, any application thread blocked on the Future object will be woken up (step 8).

3.3 Skeleton Nesting

Skeleton nesting enables the combination of multiple skeletons as well as computation kernels by defining a computation tree, depicted in Figure 3.3 and Figure 3.4, where the Kernel is actually an instance of a KernelWrapper. The delegation of such tree to the library greatly empowers it to orchestrate the graph execution, as each skeleton introduces new behaviors to its sub-tree. As such degree of responsibility is assigned to the library, the programmer is given the illusion that all skeletons within the computation tree are being executed at the same time, although this is normally not true, due to data-set dependencies and hardware limitations.

3.4 Supported Skeletons

The skeletons currently provided by the Marrow framework are: Pipeline, Loop, For, Stream and MapReduce.

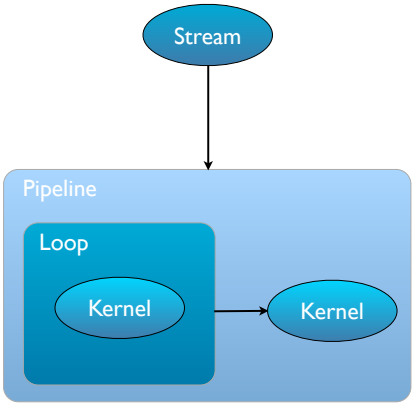

Figure 3.3: A computation in Marrow. Rectangle skeletons can be nested while others cannot.

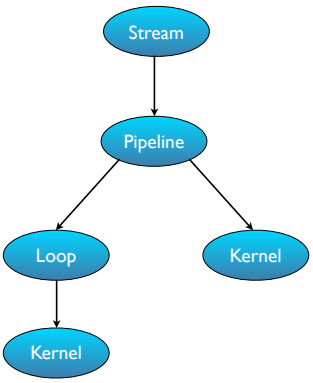

Figure 3.4: An example of a computation tree in Marrow.

Pipeline This skeleton, represented in Figure 3.5, defines a sequence of data-dependent stages of computation. Each stage can be executed in parallel in resemblance to an assembly line. Memory is retained in the device between execution stages in the pipeline, thus eliminating the transfers to host memory on intermediate results. Although only supporting two steps per skeleton instance, for simplicity sake, the Pipeline can be nested, thus enabling the creation of N-staged pipelines.

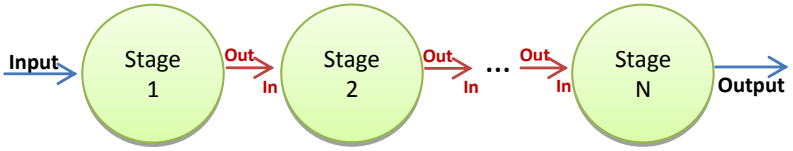

Figure 3.5: N-staged pipeline (taken from [Mar12]).

Loop The loop skeleton, depicted in Figure 3.6, applies the same computation tree iteratively according to a condition affected by predetermined information, such as a for loop statement, or from intermediate results (computed by the preceding iteration), akin to a while loop statement. This dependency is expressed by the *step* function, which updates the condition's value for each iteration, using the previous iteration's output, or not. It is possible to execute several loops in parallel within a device using distinct data-sets (using a Stream skeleton detailed ahead). The For skeleton is a specific case of Loop. The same computation is applied over the data-set a specified number of times, by predefining the condition and step function. Nesting is supported in both Loop and For skeletons.

Stream This skeleton, shown in Figure 3.7, gives the illusion of computation persistence over distinct work submissions, by introducing computation parallelism among them. This parallelism is obtained by taking advantage of the communication and computation overlap optimization. Stream is the only skeleton to introduce these optimizations, as

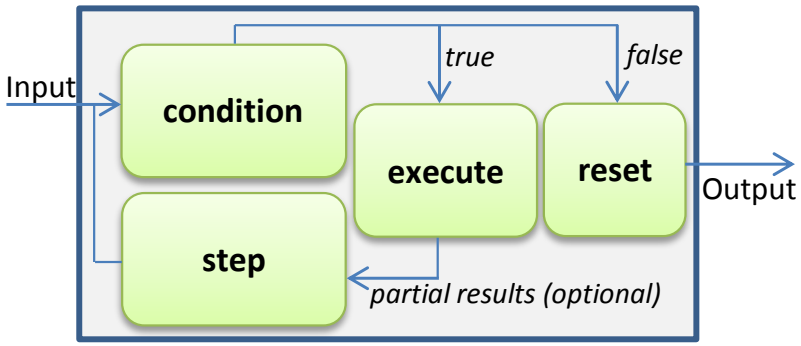

Figure 3.6: The Loop skeleton (taken from [Mar12]).

the use of skeleton nesting allows such behavior to be inherited by the nested computation subtree. Although these optimizations are transparent to the programmer, Marrow's API allows its configuration, to control the degree of parallelism possible by defining the number of instances of the computational tree that may execute in parallel. This configuration affects the effectiveness of the communication and computation overlap as well as the amount of device memory required as more data-sets are stored simultaneously on the device. As it requires full control over the input and output, this skeleton can only be used a root node of the computation tree.

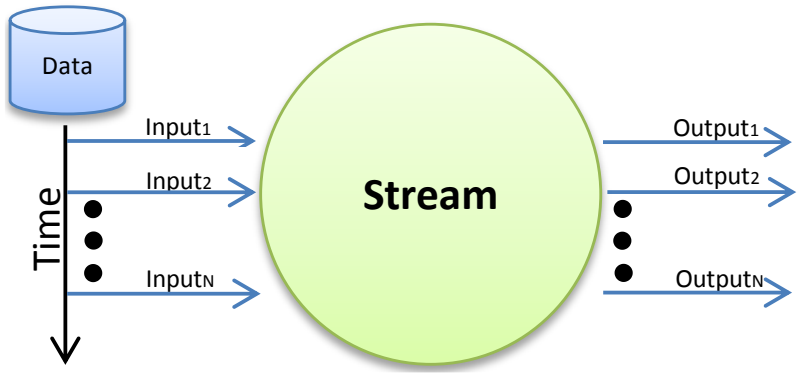

Figure 3.7: The Stream skeleton (taken from [Mar12]).

MapReduce The map-reduce skeleton, depicted in Figure 3.8, has similar semantics to SkePU's. Using user-defined functions for the split and merge operations, the data-set is decomposed and computed over several executions of the computation tree. The reduction step can be defined in two ways: with an OpenCL kernels and a CPU function, where the OpenCL kernel reduces the result partially within its assigned decomposition, and the CPU function further reduces to a final value, or simply define a single CPU function which reduces the whole data-set. In Marrow, its implementation is somewhat standalone, as it requires full control over the data-sets, thus not nestable. Differing from SkePU, Marrow's MapReduce decomposes the input data-set in partitions which are submitted over time to an internal Stream skeleton, enabling the communication and

computation overlap transparently.

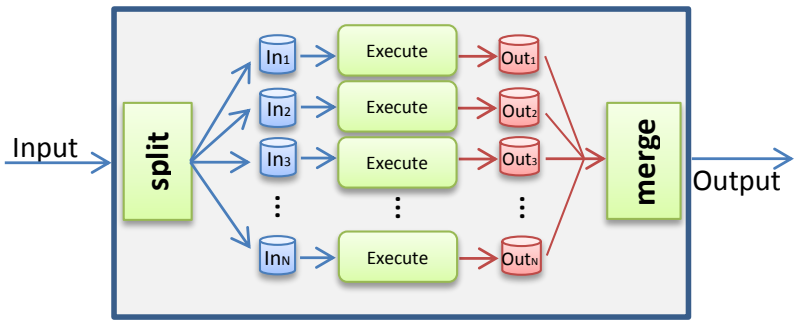

Figure 3.8: MapReduce skeleton (taken from [Mar12]).

3.5 Programming Example

This section showcases the use of the Marrow library, applying it in the programming of the simple Saxpy example, presented in Listing 3.1. Saxpy is a BLAS routine that multiplies a scalar to a Matrix and then sums the result with another matrix ($y[i] = \alpha x[i] + y[i]$). This routine is completely data-parallel as none of the operations requires more than the value to be calculated in both matrices (at the same index) as well as the scalar value itself.

Creating Marrow computations is split in two main stages: creation of the computation tree (lines 2–14) and execution requests (lines 16–28). In the first stage we start by describing the OpenCL kernel arguments (lines 15–14), followed by the creation of the KernelWrapper, to prepares the OpenCL computation for execution (line 11) and its nesting on the Stream (line 14). The use of the Stream enables the communication and computation overlap of a number of partitions (*divisions*) using a number of concurrent buffers (*numBuffers*). The second stage comprises two loops, one to decompose the dataset, creating the desired number of partitions, and the emission of the subsequent execution requests (lines 16–24). A second loop is used to wait for the results of all partitions to be available (shown in lines 26–28).

Listing 3.1: Saxpy in Marrow

```
// Stage 1: Computation tree initialization
// Define the work-size
unsigned int workSize = numberElems/divisions;
std::vector<unsigned int> globalWorkSize(1);
globalWorkSize[0] = workSize;
// Define the input arguments of the computation
std::vector<std::shared_ptr<IWorkData>> inDataInfo(3);
inDataInfo[0] = std::shared_ptr<IWorkData> (new BufferData<float>(workSize));
inDataInfo[1] = std::shared_ptr<IWorkData> (new BufferData<float>(workSize));
inDataInfo[2] = std::shared_ptr<IWorkData> (new FinalData<float>(alpha));
// Define the output arguments of the computation
std::vector<std::shared_ptr<IWorkData>> outDataInfo(1);
outDataInfo[0] = std::shared_ptr<IWorkData> (new BufferData<float>(workSize));
// Create the computation wrapper (leaf of the tree)
std::unique_ptr<IExecutable> kernel (new KernelWrapper(kernelFile, "saxpy",
                                                       inDataInfo, outDataInfo,
                                                       globalWorkSize));
// Create the root of the tree
Stream *s = new Stream(kernel, divisions, numBuffers);
// Stage 2: Execution Requests
IFuture ** futures = (IFuture**) new Future*[divisions];
// Create and submit each decomposition for execution
for(unsigned int i = 0; i < divisions; i++) {
    std::vector<void *> inputValues(2);
    std::vector<void *> outputValues(1);
    inputValues[0] = &inValues1[i*numberElems/divisions];
    inputValues[1] = &inValues2[i*numberElems/divisions];
    outputValues[0] = &outValues[i*numberElems/divisions];

    futures[i] = s->write(inputValues, outputValues);
}
// Wait for all decomposition to finish computation
for(unsigned int i = 0; i < divisions; i++) {
    futures[i]->wait();
}
```

4

Multi-GPU Support

This work's main focus is the multi-GPU extension of Marrow, within a single system to provide an increased scalability and performance potential. Additionally, the heterogeneity of the GPUs is also considered for an adaptive decomposition of the computations. Reaching this goal uncovered several challenges in Marrow's execution platform, such as performance-aware domain decomposition as well as in the operations offered to the programmer, where a trade-off between simplicity and expressiveness must be considered, to maintain usability. The former stems from the need to decompose and distribute the computation tree among the multiple devices with differing capabilities, while the latter stems from the need to provide a usable programming model to the programmer, while enabling the parametrization the platform's behavior for multi-GPU execution.

In the next section a general overview of the multi-GPU support is presented, as well as its overall architecture.

4.1 General Overview

The Marrow usage has evolved from the previous single-GPU version, by further abstracting steps which parallel execution entails, such as the data-set decomposition and their explicit submission for execution. This abstraction shifts the focus of the programmer slightly, becoming more centered on the configuration of the computation tree for implicit parallel execution, by defining data-set decomposition (and distribution) restrictions within each OpenCL computation.

The data-set decomposition within Marrow transparently creates static partitions of the data-set, taking into account the individual performance of the GPU(s) it is being

created for. These take advantage of benchmark values of each device, gathered at installation time. As the decompositions are hand-tailored for each device, there is no dynamic scheduling such as work-stealing, as each decomposition should be executed by its designated GPU for efficient execution. To take advantage of the decompositions, the computation tree is replicated, enabling their parallel computation, as depicted in Figure 4.1.

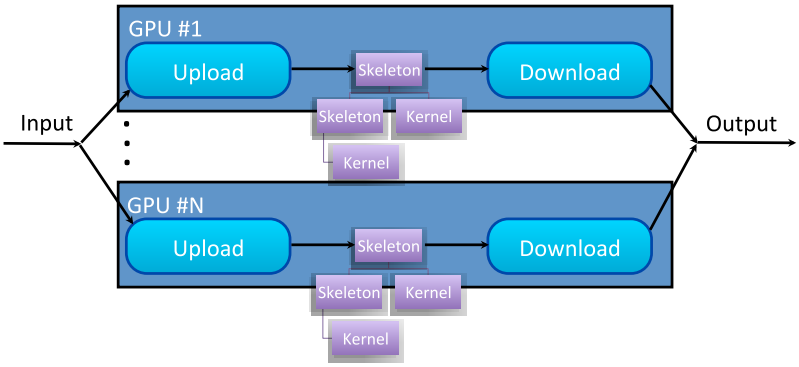

Figure 4.1: Computation tree replication for decompositions.

To increase the platform's flexibility, several new features where added to the definition of kernel arguments to express restrictions over the decompositions, such as defining whether a memory region can be decomposed (and how it must be decomposed), or must be replicated to all devices. Additionally, several argument parameters were added associate distribution-aware semantics to a computation, such as the decomposition's beginning element, size, its position (whether it is the first, middle or last partition), among others. For example, an N-Body simulation, using a direct-sum algorithm (one of our case-studies), requires that the whole data-set be replicated to all GPUs, as all elements are compared to all others. Additionally, for this case-study to work correctly, each decomposition execution has to know where is the beginning of its assigned computation, as well as its size. This allows the programmer to create more complex OpenCL computations, somewhat bypassing the limitations introduced by the opaque nature of the data-set decomposition.

Marrow's execution model remains request-driven, where the execution requests are submitted to the root of the computation tree, although, in multi-GPU Marrow, all skeletons support parallel requests over-time, akin to the Stream skeleton present in the single-GPU Marrow, with the help of the improved runtime platform. This platform contains the main components which enable the multi-GPU execution, such as the Scheduler, AutoTuner and the TaskLauncher. The cooperation between the Scheduler and TaskLauncher now enables the communication and computation overlap optimization previously offered by the Stream skeleton. This overlap's performance depends on the number of decompositions within each GPU, as these are submitted in parallel for transfer. When orchestrated correctly, this increases the usage rate of the PCIe bus, while enabling the

parallel execution of computations when their (smaller) decompositions become available on the device. Therefore we define overlap degree as the number of decompositions that are computed within each GPU device.

The set of supported skeletons has changed slightly, as the Stream skeleton has been currently swapped by a Map skeleton, because of the aforementioned universal support for data-set submissions over-time. Skeleton nesting remains a cornerstone in Marrow, enabling the creation of complex computation structures. Multi-GPU support for nesting is, however, non-trivial as the computation tree needs to be configured, in regards to several parameters, such as the desired number of GPU devices to use, as well as the number of overlapping decompositions within each. The programmer can opt to omit these values, which allows the platform to use all the GPU devices present within the system, and use a default value for the overlap partitions.

4.1.1 Architecture

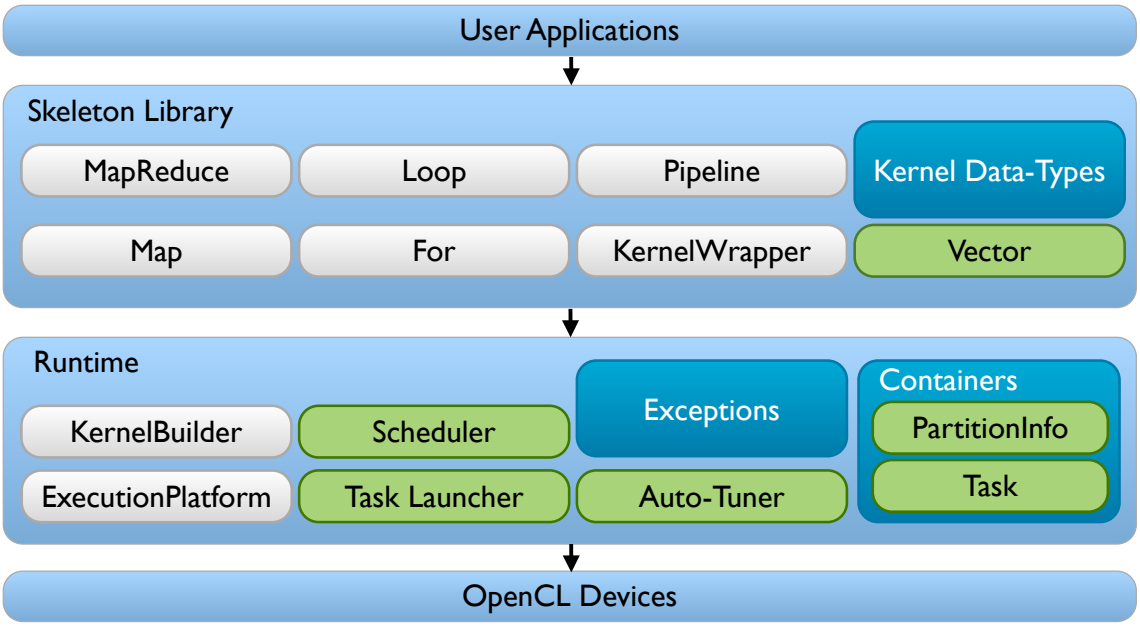

Figure 4.2: Architecture of Multi-GPU Marrow.

The multi-GPU Marrow architecture, depicted in Figure 4.2, follows an overall structure similar to its single-GPU predecessor, with a Library layer, containing the only components directly accessible by the programmer (detailed in Section 4.3). These enable the creation of computation trees, for complex computations using skeleton nesting. The Runtime layer (detailed in Section 4.4) contains the components which are used to interact with the OpenCL platform. Additionally, this layer manages the executions of computation trees, providing transparent communication and computation overlap, domain decomposition as well as resource management (i.e. command queues, memory locations among others). The highlighted components in the figure are introduced by this work,

although all components have suffered changes for multi-GPU support. Among these, the Scheduler, TaskLauncher and Auto-tuner comprise the persistent runtime platform. This platform is initialized statically when a multi-GPU Marrow program is executed, and persists until all algorithmic skeletons are de-allocated. Exceptionally, these components require access to the Skeleton Library, as these work under a paradigm similar to client-server, where the computation trees require the persistent platform for configuration (e.g. acquiring data-set decompositions), while the platform requires access to the computation tree, to orchestrate transfers and executions.

An example is presented in the next section, to enlighten the most significant changes from a programmer's point-of-view when using multi-GPU Marrow.

4.2 Programming Example of Multi-GPU Marrow

Using multi-GPU Marrow, the Saxpy example is revisited (in Listing 4.2), which applies a matrix-wise operation $y[i] = \alpha x[i] + y[i]$ with α as a scalar value. This is a data-parallel case study, given that each element only requires values in the same index of the input matrices. The programming model remains largely the same with two main stages: computation tree initialization and execution request.

Listing 4.1: Buffer default decomposition.

```
1 inDataInfo[0] = std::shared_ptr<IWorkData> (new BufferData<float>(numberElems
2                                                 IWorkData::PARTITIONABLE, 1));
```

The type of decomposition required by Saxpy (element-wise data-parallel) is expressed by the default Buffer constructor, not requiring explicit definition of decomposition restrictions Buffer objects of Listing 4.2 (lines 4–9). The default constructor is equivalent to the one shown in Listing 4.1, where it becomes explicitly defines that it can be decomposed, and that each OpenCL work-item (processing core) only requires a single element of this Buffer argument. The definition of decomposition restrictions are further detailed in Subsection 4.3.4. A KernelWrapper is then initialized, wrapping the actual computation with the same syntax as the single-GPU Marrow (line 10–12). Subsequently, a Map is used (line 13), which simply applies the nested computation to the multi-GPU platform, as Saxpy does not require additional behaviors. The computation tree is now ready to start receiving execution requests, starting the second stage of a Marrow computation. This stage has been simplified by taking advantage of the implicit data decomposition, as well as the abstraction of memory operations over the data-sets, using a Vector concept (lines 15–22). Once these are created, they are submitted for execution to the skeleton (line 23), akin to single-GPU Marrow, subsequently waiting for the results (line 24).

Listing 4.2: Saxpy in multi-GPU Marrow.

```
// Stage 1: Computation tree configuration
// Define the work size
std::vector<unsigned int> globalWorkSize(1);
globalWorkSize[0] = numberElems;
// Define the input arguments of the computation
std::vector<std::shared_ptr<IWorkData>> inputDataInfo(3);
inDataInfo[0] = std::shared_ptr<IWorkData> (new BufferData<float>(numberElems));
inDataInfo[1] = std::shared_ptr<IWorkData> (new BufferData<float>(numberElems));
inDataInfo[2] = std::shared_ptr<IWorkData> (new FinalData<float>(alpha));
std::vector<std::shared_ptr<IWorkData>> outputDataInfo(1);
// Define the output arguments of the computation
outDataInfo[0] = std::shared_ptr<IWorkData>(new BufferData<float>(numberElems));
// Create the computation wrapper (leaf of the tree)
std::unique_ptr<IExecutable> kernel (new KernelWrapper(kernelFile, "saxpy",
                                                       inDataInfo, outDataInfo,
                                                       globalWorkSize));
// Create the root skeleton fo the computation tree
map = new Map(kernel, numDevices, numBuffers);
// Stage 2: Execution request
// Create the containers with the input data-set
std::vector<std::shared_ptr<Vector>> inputData(2);
inputData[0] = std::shared_ptr<Vector> (new Vector(inputValues1, sizeof(float),
                                                   numberElems));
inputData[1] = std::shared_ptr<Vector> (new Vector(inputValues2, sizeof(float),
                                                   numberElems));
// Create the containers with the output data-set
std::vector<std::shared_ptr<Vector>> outputData(1);
outputData[0] = std::shared_ptr<Vector> (new Vector(outputValues, sizeof(float),
                                                   numberElems));
// Submit an execution request to the root of the computation tree
future = map->write(inputData, outputData);
// Wait for the results
future->wait();
```

4.3 Skeleton Library

The main changes to the Skeleton Library layer are closely related to domain decomposition. In single-GPU Marrow, this had to be explicitly defined, whereas now, it is mostly transparent to the programmer, with the exception of the definition of decomposition restrictions, if the computation so requires.

The automated decomposition introduced a need to simplify memory access within Marrow, and for the skeleton interfaces that can be extended by the programmer, such as the Loop skeleton's termination condition. To satisfy this need, a Vector concept was introduced to define a contiguous region of memory, which is submitted for computation. It abstracts memory operations over the data-sets, such as decomposition, as well as keep track of the synchronization needs with their counterparts in the GPU devices.

The multi-GPU adaptation not only required the adaptation of each skeleton's behavior to support data decomposition, but also had an impact on the actual definitions of the skeletons' usage and skeleton nesting have been modified.

4.3.1 Skeleton Interface

The ISkeleton interface, presented in Listing 4.3, is definition of an algorithmic skeleton. Its functions are used when the computation is launched at the root of the computation tree, assuming full control over the computation. Previously, in single-GPU Marrow, launching a skeleton computation was done by simply calling a single function `executeSkel`, which was in charge of data-transfers as well as launching the actual skeleton - operation that requires full control over the data-sets. Accordingly, each skeleton by itself could only execute sequentially, and was normally nested within a Stream skeleton. By becoming a nested skeleton instead of a root skeleton, the interface becomes the IExecutable, therefore delegating control to the Stream skeleton.

The development of multi-GPU marrow has entailed some modifications of the ISkeleton interface, with the purpose of handing more control to the execution platform. To the preexisting `executeSkel` function (line 2), several new functions were introduced: `uploadInputData` (line 12), `downloadOutputData` (line 20) and `finishExecution` (line 28). The `executeSkel` function triggers the execution of the computation tree over the subtrees (instances of IExecutables as detailed in the next Subsection). This function assumes that all the input data is available at the target GPU devices, an operation performed by `uploadInputData` function. The `downloadOutputData` function is called download the output data from the GPUs to make it available to the host. The `finishExecution` function is used to apply an additional computation step, before the Future is notified of the computation's conclusion. The most obvious example of the usage of this feature is the reduce step in MapReduce.

4.3.2 Executable Interface

The IExecutable interface, depicted in Listing 4.4, provides an uniform interface for nestable entities, both nestable skeletons and KernelWrappers. The first three functions, `initExecutable` (line 2), `reconfigureExecutable` (line 4), and `clearConfiguration` (line 6) are related to the way Marrow operates when preparing the resources required for execution, being the last two are new to multi-GPU Marrow. The runtime platform is initialized statically, using the available number of devices, and a default number of overlapping partitions per device. Upon initialization skeletons make use of this information to preallocate necessary memory on the GPU devices. When the programmer initializes the root of the computation tree, he can define the number of devices and overlapping partitions. When this configuration differs from the default, it requires a computation tree reconfiguration, to adhere to the new configuration. This implies the destruction of the previously reserved GPU memory and the re-allocation according to the new configuration.

Lastly, the `execute` function (line 8), applies the behavior of the current computation node (which can be a skeleton or not) to the GPU memory locations passed as arguments, while taking into account which partitions these belong to.

Listing 4.3: The ISkeleton interface.

```
class ISkeleton {
  virtual void executeSkel(const cl_command_queue &executionQueue,
              const unsigned int deviceIndex,
              const unsigned int uniqueId,
              const unsigned int partitionIndex,
              const unsigned int overlapPartition,
              std::vector<std::shared_ptr<Vector>> &inputData,
              std::vector<std::shared_ptr<Vector>> &outputData,
              std::vector<cl_mem> &inputMem,
              std::vector<cl_mem> &outputMem) = 0;

  virtual void uploadInputData(const cl_command_queue &executionQueue,
              const unsigned int deviceIndex,
              const unsigned int uniqueId,
              const unsigned int partitionIndex,
              const unsigned int overlapPartition,
              std::vector<std::shared_ptr<Vector>> &inputData,
              std::vector<cl_mem> &inputMem) = 0;

  virtual void downloadOutputData(const cl_command_queue &executionQueue,
              const unsigned int deviceIndex,
              const unsigned int uniqueId,
              const unsigned int partitionIndex,
              const unsigned int overlapPartition,
              std::vector<std::shared_ptr<Vector>> &outputData,
              std::vector<cl_mem> &outputMem) = 0;

  virtual void finishExecution(const unsigned int uniqueId,
               std::vector<std::shared_ptr<Vector>> &outputData) = 0;
};
```

4.3.3 Skeletons

The list of supported skeletons remains mostly the same except with the removal of the Stream skeleton and the inclusion of the Map skeleton. Nonetheless, the remainder had to be modified in regards to their support to multiple GPUs. The overlap of communication and computation was previously provided using execution requests over-time to the Stream skeleton, yet the new runtime platform already subsumes this overlap transparently. As a result, the Stream was excluded from multi-GPU Marrow. The upgraded runtime platform is detailed in Section 4.4.

Map enables the programmer to use multi-GPU Marrow, simply applying a computation tree without introducing new behavior. This is useful when the programmer only requires the execution of single OpenCL computation (KernelWrapper), as it is not considered a skeleton and may not be submitted for computation in a standalone fashion. This skeleton cannot be nested because it requires control over the data-set for device transfers.

Listing 4.4: The IExecutable interface.

```
class IExecutable {
  virtual void initExecutable() = 0;

  virtual void clearConfiguration() = 0;

  virtual void reconfigureExecutable() = 0;

  virtual cl_event execute(cl_command_queue executionQueue,
              unsigned int deviceIndex,
              unsigned int uniqueId,
              unsigned int partitionIndex,
              unsigned int overlapPartition,
              std::vector<cl_mem> &inputData,
              std::vector<void*> &singletonInputValues,
              std::vector<cl_mem> &outputData,
              cl_event waitEvent,
              std::vector<std::shared_ptr<Vector>> &resultMem) = 0;
};
```

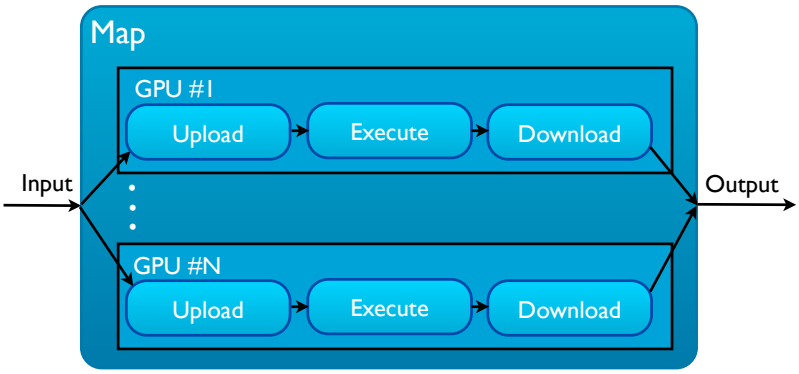

Figure 4.3: The Map skeleton with implicit decomposition.

MapReduce maintains its functionality, as shown in Figure 4.4. Its internal adaptation now takes advantage of the new Map skeleton, instead of a Stream. Additionally the data-set partitioning is now done implicitly instead of using a programmer-defined function.

Pipeline is now adapted to the new platform, supporting data-set partitions, depicted in Figure 4.5. This support entails the internal allocation of the intermediary GPU memory used between stage one and two, thus avoiding the redundant allocation of GPU memory in both stages. This feature entails that each subtree (and thus each computation) knows if the allocation of the input/output data-sets is required on the GPUs. As such, redundant communication between stages is avoided, and the memory is already in-place to be used by the second stage, after the first stage finishes.

Loop with multiple GPUs has introduced issues relating to its flexibility. Within its single-GPU execution there are already cases where the `step` function depends on the previous iteration's data-set or not (Figure 4.6).

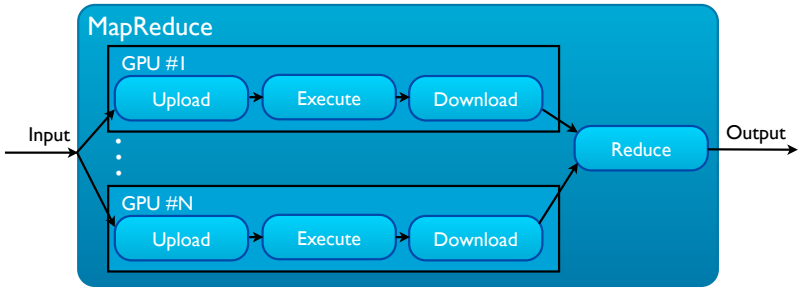

Figure 4.4: The MapReduce skeleton.

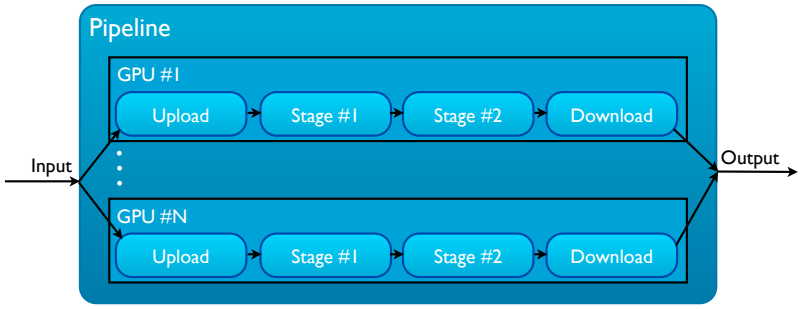

Figure 4.5: The Pipeline skeleton.

With its multi-GPU generalization, it is necessary to add a new semantic that only allows the `step` function to be computed when all other partitions have finished computation of the current iteration (Figure 4.7). Additionally this synchronization requirement normally originates from the dependency on the whole data-set for a correct `step` computation. Therefore, when using this behavior, the `step` function is only computed once per iteration by a single-thread which has access to the whole data-set.

Although this behavior could be always applied to all computations, it occurs in a performance penalty due to its synchronized behavior when it is not needed. Thus it is also possible to execute these decompositions independently, where each computes its own `step` function over their data-set.

Accordingly the step function can be computed according to one of the following dependencies:

- Requires the full data-set available (only one thread calculates the `step`);
- Requires only the current decomposition's data-set. Several parallel `step` computation per iteration;
- Does not require any data-set. As the previous mode, several parallel `step` computations.

This behavior can either be the default (completely data-parallel without using the previous iteration's data-set), or must be defined by the programmer, as it depends intimately on the computation.

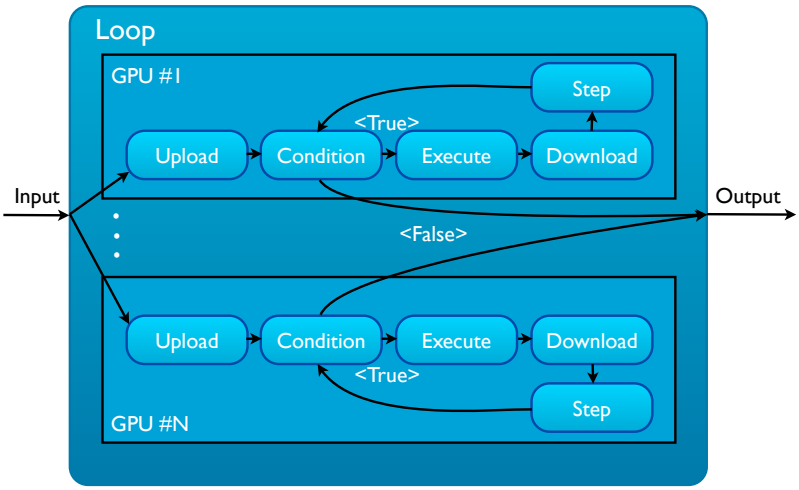

Figure 4.6: The Loop skeleton, with parallel *step* computation.

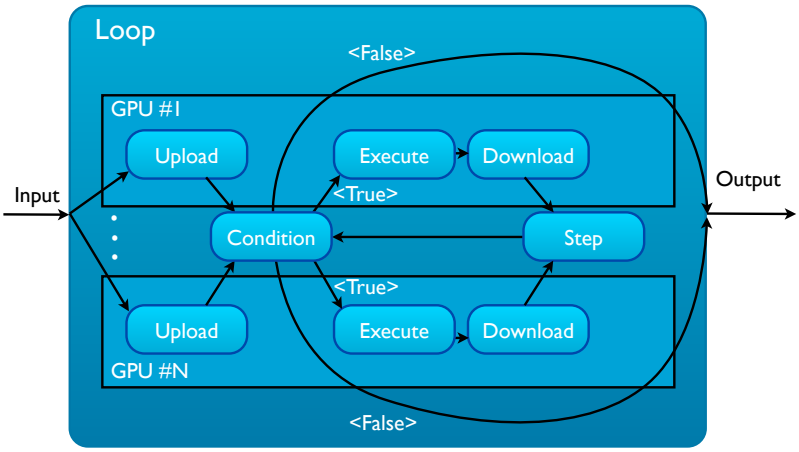

Figure 4.7: The Loop skeleton, with a globally synchronized *step* computation and condition.

KernelWrapper now supports multiple GPU execution, by embedding the information concerning the decomposition of its domain, as well as their required OpenCL memory resources.

4.3.4 Definition of Computational Arguments

Marrow uses a set of C++ data-types which translate into various OpenCL data-type counterparts, that are used to define the computation's arguments. Currently the following data-types are supported:

- Buffer represents a contiguous memory region, with single dimension, defined by the programmer when creating an execution request;
- Singleton represents a single element, which is defined by the programmer when creating an execution request;
- Final represents a single element like the Singleton, however, it is defined when the actual data-type is created, even before the assignment to a KernelWrapper;
- Local reserves a local memory location which is ready when the execution starts.

The Image2D data-type present in single-GPU Marrow, has not been included in multi-GPU version, as it introduces new challenges and significant complexity when dealing with data-set decomposition, due to its multi-dimensional access to data. Although none of our case-studies previously used this data-type, it remains an interesting exercise for future work.

Marrow's implicit decomposition somewhat limits the freedom the programmer has when designing a computation tree and its data-set dependencies, in exchange of simplicity. To help overcome this limitation, the Buffer and Final data-types have been improved with new features.

Final data-type now supports a Trait. This feature informs the execution platform that it should replace the value of the argument by a certain value regarding the current decomposition. The supported traits are:

- Default - maintains the default behavior, submitting the original value;
- Offset - replaces the value with index of the decomposition's start. This value is useful for computations which need to know the global position of the decomposition, such as a N-Body simulation where the whole data-set is replicated, and each device only computes a region, partially defined by the offset. This value is obtained from the first partitioned input argument;

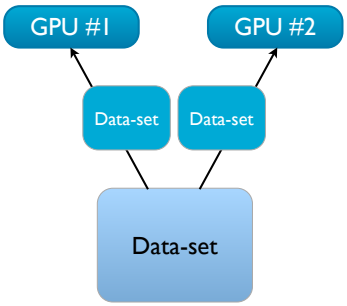

Figure 4.8: Buffer transfer with Copy mode.

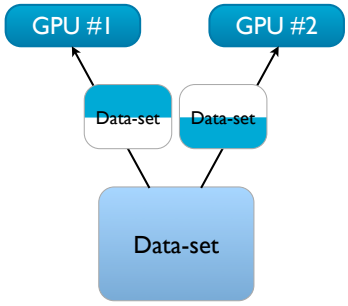

Figure 4.9: Buffer transfer with Partitionable mode.

- Size - replaces the value with the size of the decomposition. This value is important when the computation requires the number of elements of the decomposition, instead of the work-group sizes. Akin to Offset, this value is harvested from the first partitioned input;
- PartLocation - replaces the value with the location of the decomposition: Top (first), Middle and Bottom (last). This value is useful for computations which depend on vertical borders. These are defined using predefined integers within an enumeration.

Buffer data-type has also been improved with two decomposition options: Copy and Partitionable.

The Partitionable mode, depicted in Figure 4.9 allows the auto-tuner to decompose the Buffer, accordingly to the computation's restrictions. The creation of decompositions is detailed in Section 4.4.

The Copy mode, shown in Figure 4.8 transfers the whole buffer to the GPU devices without any decomposition at all. Internally, the decompositions are computed, however these are not used for data-transfer. In conjunction with the previously presented offset and size options of the Final data-type, it is possible to direct a computation to a region of the data-set. This decomposition mode is useful for computation which have a dependency on the whole data-set, but can decompose the region of the data-set to be computed, such as a n-body simulation. Furthermore there are five variants of the Copy mode, which can only be used for output arguments. These allow the programmer to merge each copy (assigned to different decompositions) by either predefined functions (sum, subtraction, multiplication and division) or a user-defined function. These functions are currently computed sequentially and hence these should be used with care as they easily bottleneck when used for large data-sets.

The implementation of the predefined functions was a challenge, as many of the data-types available in OpenCL are composite (e.g. cl_uchar4), and do not support arithmetic operations by default. Overcoming this implied the usage of C++ templates using a technique called *Substitution failure is not an error* [VJ02] (SFINAE), which enables the

detection of a data-type's operation support, as shown in Listing 4.5. The base operation-support detection structure is defined, providing the boolean value with the support of the data-type for a certain operation (line 17–18). This value is ascertained by defining two functions with the same name `sum_test`. The first function (line 11) is only called if the `decltype` is valid, in this case the application of the sum operation, returning a char data-type (one byte). If the first function fails to be called, the second function (line 13) simply returns a data-type with a different size (in this case two bytes), allowing the support detection using the return data-type size.

Once this is implemented, two template functions are defined for each operation, one when the value is true for example `SumImpl<Type, true>`, which applies the arithmetic operation over the data-set, and the other, `SumImpl<Type, false>`, which throws an exception, as there is no support for the operation. The function can simply be called using the boolean value (shown in Listing 4.6). Ideally, when an operation is not supported, there should be compilation error, which does happen by default, however, the error message itself is cryptic, as there are templates involved, and no alternatives were found to customize this error message. When the programmer defines a custom merging function, there is no compatibility detection, being his full responsibility.

Listing 4.5: SFINAE example using the sum operator.

```
1  // Defines two types which differ in size so they can be identified.
2  struct sfinae_base {
3    typedef char no[2];
4    typedef char yes;
5  };
6
7  struct supports_sum : sfinae_base {
8    template <class U>
9    // Define a function which is execution of the type U supports the
10   // sum operation.
11   static auto sum_test(const U* u) -> decltype(*u + *u, char(0));
12   // The function which is executed if the first is not.
13   static typename sfinae_base::no& sum_test(...);
14   // This value contains the boolean value which informs if the data-type Z
15   // supports the sum operation or not, by comparing the sizes of the
16   // returned data-type.
17   static const bool value = (sizeof(sum_test( (Z *)0) ) !=
18                       sizeof(typename sfinae_base::no));
19 };
```

Listing 4.6: SFINAE usage.

```
1  SumImpl<Z, supports_sum::value>::merge(v1, v2, numberElements);
```

4.3.5 Example

To showcase the usefulness of these features, we present the N-Body case-study, which without these features would be impossible to implement in Marrow.

The N-Body is a well-known simulation, where there is a set of bodies, and these affect each other according to a force that varies their velocity. As each body depends on all others for the velocity evolution, there is a full data-set dependency in-place. To deal with this dependency the Copy decomposition mode is used, replicating the whole data-sets to all GPUs that are used.

The introduction of the Copy replication, introduces another challenge, as each GPU now has to know which region of the whole data-set the computation must be applied to. To overcome this, we use the Offset final data-type option, which allows us to define the initial element of the computation. While the OpenCL work-size informs us of the decomposition's size, as each work-item is in charge of a single body.

4.4 Runtime

Marrow's runtime layer is the main target of improvement for the support of multiple GPU devices. As with the previous iteration of Marrow, communication and computation overlap is still used to achieve higher performance, although at a higher degree of overlap, as it is not only used within a single GPU, but among all, which in some cases entails overhead, as it tends to bottleneck the PCIe bus when computations are data-bound, with a low degree of computation (detailed in Chapter 5). Computations in multi-GPU Marrow, akin to the single-GPU counterpart, follows two main stages: skeleton initialization and skeleton execution request.

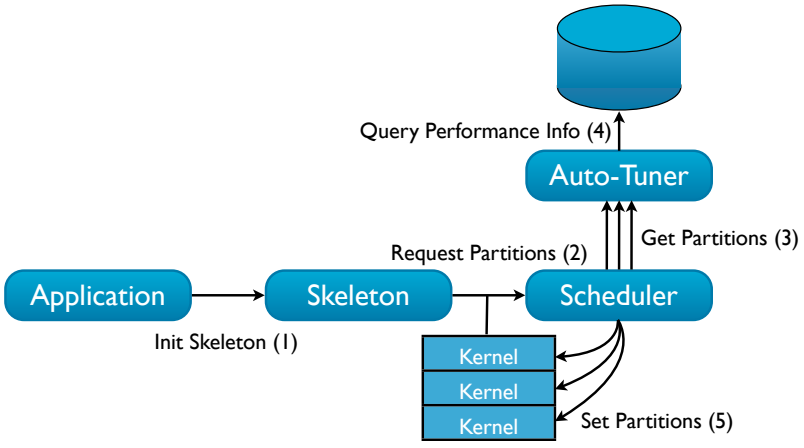

Figure 4.10: Initialization of a Marrow skeleton.

Skeleton Initialization Before any computation may take place on the multi-GPU Marrow, the programmer must first create the computation tree, and define the execution configuration (number of GPUs and decompositions) so the platform can configure itself, although these can be omitted (using all the available GPUs and a default number of decompositions). In Figure 4.10, the steps that the skeleton initialization entails are

presented. When a skeleton is initialized (step 1), in addition to the usual allocation of internal resources, a request is sent to the Scheduler, containing the list of the nested kernels (step 2). A data-set decomposition request is then forwarded, per kernel, to the Auto-Tuner (step 3). The Auto-Tuner then decomposes each kernel's input and output arguments, taking into account performance information obtained at installation-time (step 4). There are two types of decompositions, normal partitions, and overlapping partitions. The former are created by decomposing the data-sets taking into account the performance of each GPU device, while the latter are created by further decomposing partitions as to increase the degree of communication and computation overlap, in an attempt to improve the PCIe bus' usage rate. The decomposition information is then saved within each kernel object (KernelWrapper, step 5).

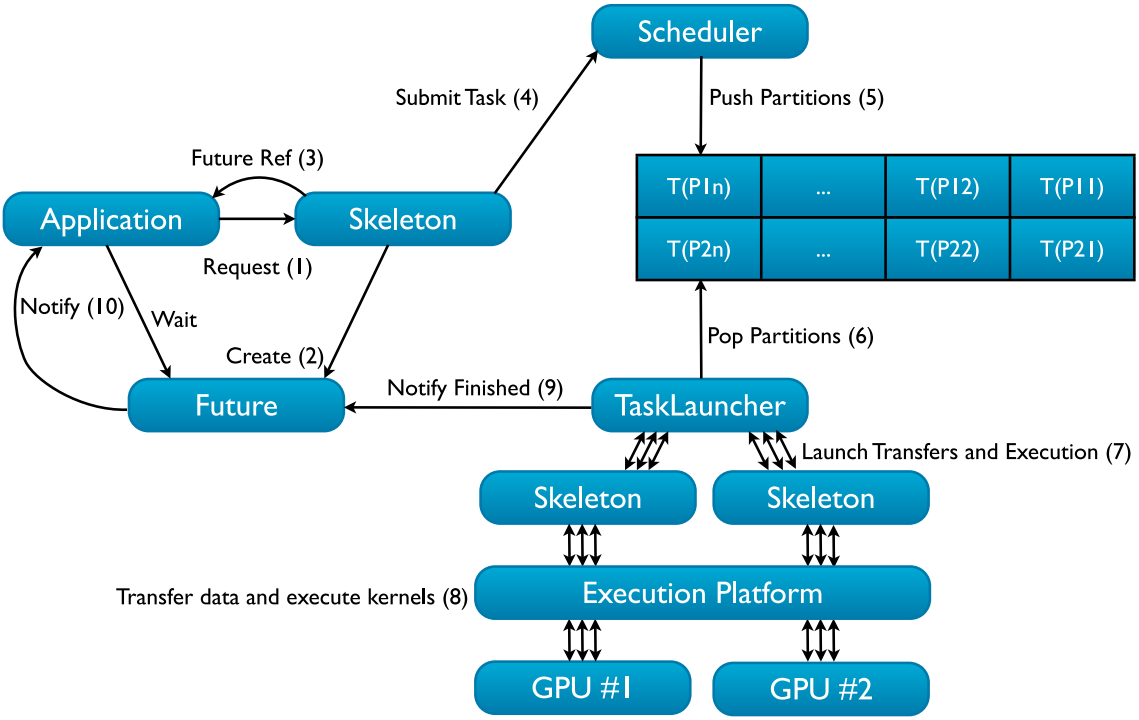

Figure 4.11: Execution request in Multi-GPU Marrow.

Skeleton Execution Request Once the skeleton initialization finishes, the computation tree is able to start handling execution requests, as shown in Figure 4.11. This stage starts when the application requests and execution from the skeleton (step 1), containing the data-sets used for the computation. A Future object is created (step 2) and its reference is returned to the application (step 3), granting the application access to the computation's state as well as access the output data. The application may then wait for the computation to finish, or continue progressing until it actually requires the output information.

Meanwhile, the execution request is forwarded to the Scheduler using a Task object containing the input and output data-sets (step 4). As the decompositions are static, and assigned to each GPU device, this component submits multiple Task instances (number

of devices $\times$ the number of the overlapping partitions). Each entry is in fact an instance of a TaskEntry, which contains a pointer to the original Task as well as the index of the partition as well as the overlapping partition index within the former. Currently the partition index is the same as the device index, however, when using different scheduling algorithm this assumption might not hold.

These objects are then submitted to each device's task queues (step 5), which are shared with the TaskLauncher. This component contains a pool of threads waiting for new TaskEntry objects to arrive to the GPU queues, concurrently pulling them from the queues (step 6). Given a TaskEntry object, its extra information is stripped, and used to obtain the correct data-set pointers (taking into account the decomposition) and the Task object is recovered, containing the reference to the root skeleton, which is accessed to trigger the steps necessary for the computation (step 7), such as the data-set transfers (`uploadInputData/downloadOutputData`) and the actual skeleton execution orchestration (using the `executeSkel` function). The execution then follows the computation tree structure, applying the nested skeletons behaviors until reaching the KernelWrapper objects located at the leaves, which launch the actual OpenCL kernel computations to the GPUs. The output data is finally downloaded to the host when the whole computation tree has executed (step 8).

When all decompositions have been computed, the task is considered finished and the Future is notified (step 9) as well as the application (step 10), awakening it as the output data is available.

4.4.1 Scheduler

The Scheduler, is the entry point to Marrow's runtime platform, as execution requests (Tasks) are initially submitted to this component. Furthermore, it has exclusive access to the Auto-tuner as the domain decomposition is closely related to the scheduler's configuration (i.e. number of GPUs to use and number of overlap partitions within each GPU). A Task is created when a execution request is made to a skeleton, and it contains the state of the computation (finished or not), as well as a pointer to the root node of the computation tree. Task submission to the GPU queues is composed by the creation of several TaskEntry objects, each representing the execution request of a single overlapping partition of the wrapped Task. The latter is only finished when all the TaskEntry have finished they computations as well as data transfers.

As Marrow uses static decompositions, hand-crafted for each GPU, the Scheduler follows static scheduling, submitting the TaskEntries for computation to their predefined GPU queues, without requiring migration as the static decompositions ensure an efficient execution, on the GPUs they were created for. The queues are organized within an array, where each queue is assigned to a GPU by its correspondence to the GPU's position in the OpenCL platform configuration.

Alternatively, the Scheduler could be implemented using dynamic scheduling, with

work-stealing with the assistance of the Auto-tuner, to help define when a Task should be stolen. This option, however, was not explored due to a limitation currently imposed by the Loop skeleton. When computing with a fully synchronized condition and *step* function, each new iteration can only begin when all decompositions of the previous one finish their computation. With work-stealing, it is necessary to either stash the intermediary results from each decomposition and sequentially steal additional ones, until all the work related to a single iteration is completed, or to use a thread per decomposition using seamless synchronization (i.e. barrier). The latter tends to be unfeasible as work-stealing commonly creates a very large number of decompositions for a fine-grained performance adaptation. Additionally there are data-migration concerns which should be addressed with this approach, as PCIe bus communication is expensive. The Auto-tuner, would have a different purpose, such as assisting the Scheduler on the decision whether a decomposition should be stolen, while taking into account migration, instead of the creation of hand-tuned partitions. This option is interesting as its dynamic nature enables a runtime-adaptation to performance discrepancies within heterogeneous GPUs, although requiring a completely different approach to auto-tuning. Therefore this exercise is left as future work.

4.4.2 Task Launcher

The TaskLauncher is responsible for the launch of data-transfers and computations. To achieve this goal, the component consumes TaskEntry objects from the GPU queues shared with the Scheduler using a pool of threads, with its size depending on the configurations of the number of devices and degree of overlap. The degree of overlap is the number of threads used for parallel data-transfers and computation within a single GPU device. Each thread is assigned to monitor a single GPU queue, greedily acquiring new work. Each GPU, however, can have multiple threads assigned to it, depending on the configured overlap degree. Therefore are $numberOverlapPartitions \times numberGPUs$ threads, ensuring that the there is a thread per overlap partition, achieving the usage of the barrier when using the Loop skeleton when using synchronized execution, avoiding stashing partial results within each iteration. Furthermore, this number of threads allows the maximum degree of parallelism within a Task, as all partitions are processed in parallel.

As this component provides such a high degree of parallelism, the overlap of communication and computation is achieved transparently, as the requests are submitted to the OpenCL platform (by the skeletons/KernelWrapper) as soon as possible, and henceforth delegated to the OpenCL platform, for scheduling over the GPU devices internally.

4.4.3 Auto-Tuner

The creation of the Auto-tuner component for multi-GPU Marrow was a challenge, due to the fact that contrary to other algorithmic skeleton frameworks that use predefined computation kernels for each skeleton, Marrow mainly introduces behaviors, while the

programmer provides the OpenCL kernels, enabling higher computation flexibility than in other algorithmic skeleton frameworks. To maintain this flexibility, the data-set decomposition must be able to effectively handle an arbitrary number of arguments, as well as their arbitrary type and finally their mapping to the computation itself (restrictions). Handling these requirements required some planning, as a common feature is fundamental for a possible mapping within all these variable configurations.

The answer is the OpenCL kernel's work-size definition, which defines the number of GPU cores (work-items) required for the kernel computation. As this configuration is tied intimately by the GPU's architecture, it is possible to use this information for decomposition, by using a relation between the work-size and the kernel arguments' size, specifically, multi-element memory regions.

As each kernel within a computation tree can have differing number of arguments (with the exception of the stages of a pipeline), as well as work-sizes, when a skeleton requests partitions, it actually gathers and submits all nested KernelWrappers to the Scheduler, so this component can then submit them to the Auto-tuner sequentially. Therefore Marrow's decomposition, is completely kernel-centric and is heavily reliant on the kernel's work-size to define compatible decompositions among all the arguments.

Currently the multi-GPU Marrow, focuses solely on the Buffer data-type, which represents a contiguous region of memory, that has single-dimensional access. Although the Buffer is a single-dimensional data-type, the work-size configuration can actually have multiple dimensions (e.g. an image can be represented with a buffer, and be configured with a two-dimensional work-group in terms of width and height, as the image can be represented contiguously line-by-line). As such, Marrow only decomposes the work-size using the last dimension, obtaining decompositions with contiguous memory regions. The correct translation between the work-size configuration and the array arguments is however, impossible in general without the assistance of the programmer. For this decomposition to work, the programmer must define how many elements each work-size element requires, multiplied by the sizes of all the dimensions except the last one (that is decomposed automatically). We name this value *indivisible size*, generally calculated using the formula:

$$indivSize = arrayElementsPerItem * \prod_{n=0}^{numDim-2} workSize[n]$$

Where $arrayElementsPerItem$ defines how many array elements are computed by each work-item (e.g. a 2D image filter which processes two pixels per work item in a line should have $indivSize = 2 * workSize[0]$, with $workSize[0]$ half the width of an actual line in the buffer, and $workSize[1]$ being the decomposed height).

Given the previously presented challenges, data-set decomposition in multi-GPU Marrow is done with two main stages, applied when an algorithmic skeleton is initialized: work-size and argument decompositions.

Work-size Decomposition This stage decomposes the last dimension of the work-size. As one of Marrow's goal is to provide an efficient execution on possibly heterogeneous GPU devices, this decomposition must take into account each GPU's performance. Currently two main performance values are evaluated: single- and double-precision performance. These values are collected at installation time using FLOPs benchmark from the Shoc benchmark suite. Two ratios are then calculated among the GPU devices, for both single and double precision. Identifying double precision computations is not perfect in Marrow as no OpenCL code is analyzed. The ratio to use is chosen with the size of the arguments data-type. This is not ideal as there are structured data-types (e.g. cl_uchar4) in OpenCL which are not trivially identified and will mis-classify.

Once the ratio is decided, the work-space is decomposed proportionally to each GPUs ratio. This distribution is memory-aware as the partition sizes adapt according to the available memory in each GPU. This feature is based on a static/dynamic memory requirement analysis within the computation tree which calculates the maximum work-items each GPU can handle. Static memory does not vary with the size of a partition, such as a Buffer with the Copy decomposition mode (and variants). The dynamic memory varies with the size of partition, such as the Buffers with the Partitionable decomposing mode, represented as the amount of memory a single work-item requires. Furthermore, when the configuration implies several partitions within a GPU (overlapping partitions), the proportional partitions created are further decomposed to the desired number of same-sized partitions within a each GPU, as shown in Figure 4.12.

Although only the kernel's work-size has been mentioned, there is also the possibility of defining a work-group size. This configuration allows the programmer to define the amount of work-items each group has, sharing among them cache memory. Normally if none is defined, the OpenCL platform picks one transparently, although these may not give the best results. As such, the programmer may define a work-group size, which has to be a multiple of the work-size. In multi-GPU Marrow, this parameter can also be defined, adding a new restriction to the decompositions, making the work-size decompositions a multiple of this configuration.

Argument Decomposition The second stage, analyses the input and output data-sets from a KernelWrapper, and while taking into account the previously calculated work-space partitions as well as the indivisible size defined within each Buffer data-type, the decompositions are created, by defining the beginning element for computation, the beginning element for data-transfer (which differs for Copy decomposition mode) and the number of elements. Additionally a new argument object is created with the same data-type, but containing the new decomposition size. All this information is then saved in a PartitionedInfo object, which contains all the decompositions of a single argument. This set of argument partitions are saved within the KernelWrapper they belong to.

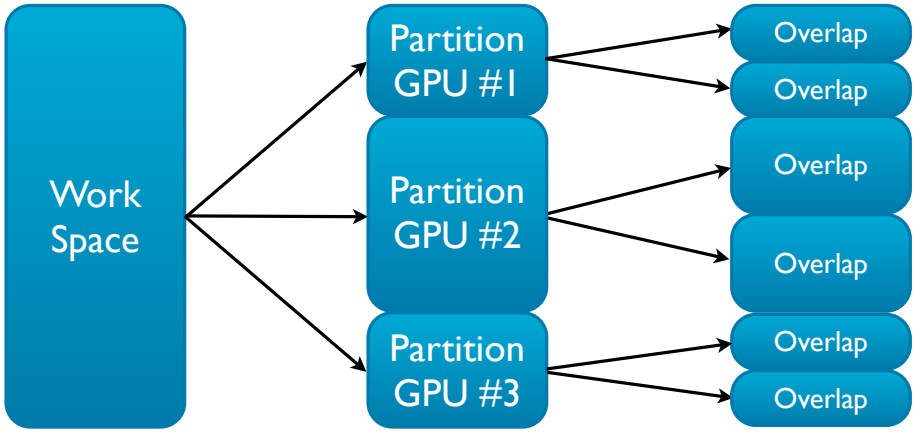

Figure 4.12: Auto-tuner work-space partitioning and overlap partitions.

4.4.4 Execution Platform

The execution platform has been improved to support the multiple GPUs, as well as the tracking of the available memory within each device. There is a limitation to OpenCL however, as it cannot query the amount of currently used memory. Consequently, memory tracking is only local to the Marrow platform, and might not work correctly as the assumption that only our platform uses the GPUs may be broken by other applications.

Communication and Computation Overlap is a feature which is, closely related on the underlying implementation of OpenCL, possibly wielding varying results depending on the hardware used. There are three possible configurations to implement this feature:

1. use a single context (shown in Figure 4.13) for all GPUs, containing several command queues;
2. using a single OpenCL context per GPU (shown in Figure 4.14, containing several command queues (depending on the degree of overlap);
3. using several OpenCL contexts per GPU, each containing a single command queue, creating an arbitrary number of contexts depending on the degree of overlap (shown in Figure 4.15).

An OpenCL Context provides a scope for allocated GPU memory, command queues as well as computations themselves. As this scope has to handle concurrent implementations, there has to be synchronized access at some level within the OpenCL platform.

Two systems were used for the development of Marrow (detailed in Chapter 5), one NVIDIA platform and one AMD platform. On our development process we have observed that both platforms have the same behavior:

- Configuration 1 does not enable any speed-up with any degree of overlap or number of devices, implying a global synchronization within a context;

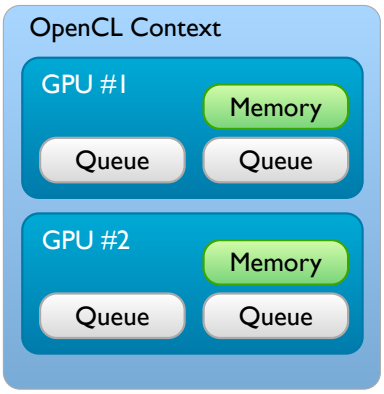

Figure 4.13: Using a single contexts for all devices and command queues.

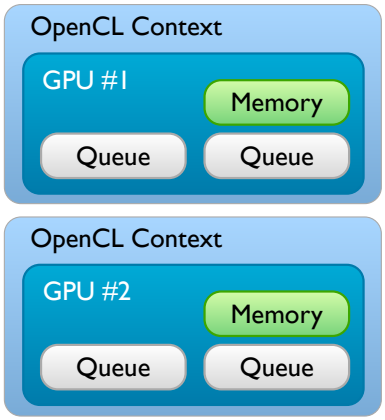

Figure 4.14: Using a context per GPU, containing multiple command queues.

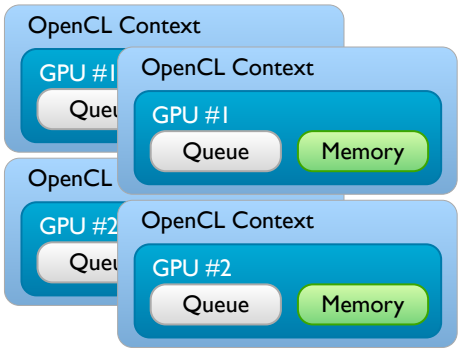

Figure 4.15: Multiple contexts per GPU, each with a command queue.

- Configuration 2 produced scalability with the usage of multiple devices, although there was no performance gains when using multiple command queues within the same GPU (and context), which is consistent with the suspicion of synchronization within a context;
- Configuration 3 finally produced performance improvements with the usage of multiple devices, as well as with the usage of multiple queues for each device.

Consequently we have organized the execution platform using the third configuration, with a context per overlap partition (with a single command queue within). However, this choice does have drawbacks, as it introduces redundant memory allocation on the GPUs when using the Copy decomposition option and multiple overlapping partition within a GPU, as the memory location cannot be shared among different contexts. An option to overcome this would be to use configuration 2, at the expense of performance when using overlap with each GPU.

4.4.5 KernelBuilder

The KernelBuilder is responsible for the compilation of the OpenCL kernels. Naturally it has been adapted for multi-GPU support by enabling the choice between which devices it should compile for, depending on the OpenCL context, as well as saving these to persistent memory to avoid redundant re-compilations.

4.5 Final Remarks

In this chapter, we have presented the multi-GPU support for the Marrow algorithmic skeleton framework.

Differing from all the other algorithmic skeleton frameworks, the programmer defines the actual OpenCL kernels, enabling a broader support, as well as the combination of these using skeleton nesting. Although the programmer has some extra burden to supply the OpenCL computation, Marrow focuses on abstracting the host-side orchestration and allocation required for GPU computations as well as to support different behaviors in regards to decomposition, to bypass the limitations of an automated decomposition system.

A statically available execution platform is used, which receives execution requests from algorithmic skeletons, akin to StarPU's interaction with SkePU, although Marrow's focus is on task-parallelism instead of data-parallelism. The platform also provides static domain decomposition, which takes into account the relative performance of the available devices, thus supporting heterogeneity. The inclusion of automated decomposition warranted new ways to increase flexibility, as many problems require decomposition restrictions because of the way the computations function. The answer to this requirement is to allow different distributions for the multi-element data-sets, akin to the ones present

in SkelCL, by either copying the whole data-set to all devices or to decompose them. Marrow's usage has evolved, by including the Vector, for the definition of data-sets, which abstracts data-set decomposition as well as data-access, while providing a vision of data persistence internally.

5

Evaluation

In this chapter we evaluate the performance of our prototype implementation from the performance, efficiency and productivity perspectives. For that purpose we have implemented several case-studies and conducted our measurements in two distinct systems with multi-GPU acceleration.

5.1 Case-Studies

To this goal, six case-studies are used: a Filter Pipeline, FFT, a N-body simulation, Saxpy, Segmentation and Hysteresis.

Filter Pipeline This case-study comprises several image filters pipelined, namely Gaussian Noise, Solarize and Mirror. All these filters have an horizontal line dependency, thus only allowing decomposition of its height, as each work-item affects two non-contiguous pixels within the same line. This restriction is expressed by defining an indivisible size equal to the size of a line. These kernels have been taken from AMD's OpenCL Samples[1].

The computation tree is composed of two nested pipelines, thus obtaining three stages configured to pipeline the execution of the filters in the following order: Gaussian Noise $\Rightarrow$ Solarize $\Rightarrow$ Mirror.

FFT This case-study comprises a set of single-precision, Fast-Fourier transformations (with 512kB each). The computational kernel has been adapted from a benchmark from

[1] http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk

the Shoc Benchmark Suite. The computation is structured using a pipeline, where the first is the FFT and the second is its inverse.

The input set of FFTs is decomposed in smaller sets, which are parallelized among the various GPUs. The work-size is defined as the number of FFTs to be computed, therefore, the input arguments have a indivisible size of $numFFTs \times numElementsPerFFT$.

N-Body This case-study performs the very well-known N-Body simulation, where the position and velocity of a set of elements (bodies), are calculated based on the force interactions amongst them.

The kernel follows the classical direct-sum algorithm, of complexity $O(N^2)$, where a single body is affected by all the other bodies in the set. Therefore, a dependency on the whole data-set is imposed, requiring full replication to all GPUs in our implementation. Although the access to the whole data-set is required, it is possible to assign computation regions to each decomposition, entailing a synchronization step after each iteration amongst all decomposition.

This functionality is achieved with a specialized Loop skeleton. While it is similar to a For, as the `step` function simply increments a counter until a certain value is reached, before each `step` computation, the decompositions are read from the GPUs, and afterwards written to all devices so they obtain a unified vision of the current iteration's input.

Akin to the image filters, the N-Body computation kernel was taken and adapted from AMD's OpenCL Samples.

Saxpy This case-study is a matrix operation which is part of BLAS (Basic Linear Algebra Subroutines), which has already been extensively presented within the previous chapters. It performs the multiplication of a matrix with a scalar value, and sums the result with a second matrix ($y[i] = \alpha x[i] + y[i]$). Both these operations are data-parallel as they only require the elements at the current position in both matrices in its computation. This computation is single Map skeleton with the Saxpy computation kernel nested within, without any decomposition restrictions.

Segmentation The segmentation is a tomography image enhancing algorithm. A tomography image is a set of pixel within a three dimensional space, where each pixels' value is a gray-scale that represents the amount of absorbed radiation at that position. This image is affected by the algorithm, changing its gray-scale value into either white, gray or black, depending on a value threshold.

A single Map skeleton is used with the nested computation kernel. Although this computation presents pixel-wise parallelism, it presents a three-dimensional workload, thus the Auto-Tuner decomposes only within the last dimension (coarse-grained decomposition).

Table 5.1: Systems used for the evaluation of Multi-GPU Marrow.

	CPU	Motherboard	Drivers	GPU #1	GPU #2	GPU #3
S_1	Xeon(R) E5506 @ 2.13GHz	ASUS P6T7	319.21	C1060	FX 3800	C1060
S_2	i7-3930K @ 3.20GHz	ASUS P9X79 PRO	1113.2	HD7950	HD7950	N/A

Hysteresis This is another tomography image enhancing algorithm [Cad+10]. Akin to segmentation, its input is a three dimensional image and hysteresis affects it by attempting to eliminate redundant gray values, classifying each pixel as either black or white.

This algorithm consists of three iterative stages (Loop skeletons), where each of their stop conditions depend on the previous iteration's output. These would be commonly structured using two Pipeline skeletons, yet these have incompatible arguments among the several loops, impeding this option. Therefore, the implementation sequentially applies three Loop skeletons.

Akin to segmentation, there is a three-dimensional workload, thus, only the last dimension is decomposed by the Auto-Tuner. This algorithm is commonly used after segmentation is applied, to further reduce the number of redundant gray values.

5.2 Systems

Two systems are used to evaluate multi-GPU Marrow, described in Table 5.1. The first system, S_1, equipped with a NVIDIA platform containing three heterogeneous GPUs configured in the OpenCL platform as depicted in the table. The ratios calculated for data-set distribution within each GPU are: $0.36 :: 0.27 :: 0.37$. The second system, S_2, has a AMD platform containing two homogeneous GPUs, where the distribution is simply split in half to each GPU. Both systems use Ubuntu 12.04.3 with a 3.8.0-32 kernel.

There is also an important feature in system S_2 as its motherboard is equipped with dual PCIe x16 lanes (dedicated to each GPU), enabling near-linear speedup when transferring data to one GPU versus transferring the same amount of data but partitioned equally between the two GPUs. This behavior has been confirmed with our data-transfer benchmark. When applying this benchmark to S_1, it becomes evident that there is a single PCIe x16 lane shared amongst the three GPUs, exhibiting a 24% improvement when using two GPUs and a 36% when using three GPUs versus a single GPU, meaning there is an improvement of the PCIe usage rate, but no dedicated lanes.

5.3 Metrics

The evaluation begins by measuring the library's overhead, in Section 5.5, versus its pure OpenCL counterpart, in both single- and multi-GPU executions.

In Section 5.6, Marrow's performance is compared against the original, single-GPU version, evaluating execution performance, when both are using a single GPU, at which the latter excelled.

Having evaluated the single-GPU performance, the multi-GPU scalability is studied in Section 5.7, by presenting the possible multi-GPU configurations, and their performance results versus the single-GPU performance. Additionally, in this section it is also studied the impact of the overlap of communication and computation when using multiple GPUs.

The domain composition is then evaluated in Section 5.8, by comparing the execution times within each GPU, observing the degree of work imbalance amongst them, using the discrepancy on their execution times.

Finally, we attempt to measure Marrow's productivity gain when programming on it versus pure, multi-GPU, OpenCL in Section 5.9, using both case-studies' code-sizes.

5.4 Methodology

The case-studies presented in this chapter are the result of 500 runs, where these are ordered, and two thirds of the edge cases are ignored, thus only taking into account the middle third of the values (167 runs). The standard deviation is calculated solely from the middle third of the values, by applying the using an sample standard deviation (using an unbiased estimator of the variance).

The execution times presented in this chapter only take into account the actual execution times of the computation tree, excluding platform initialization. Although the decomposition is not included within this execution time, it is relevant to mention that its overhead is negligible, being $< 0.5ms$, as these are simply arithmetic operations and the creation of the decomposition structures.

The usage of the communication and computation overlap optimization varies according to the metric to evaluate. It is switched off when measuring overhead versus pure OpenCL implementations and distribution quality. However, when studying scalability, against the original, single-GPU Marrow, as well as among multi-GPU configurations, we pick the best result among the overlap configurations. This configuration was varied for each measured GPU configuration using $\times 2/\times 4$ decompositions per GPU device. The overall impact of this optimization is presented, in Subsection 5.7.1.

Hysteresis This case-study consists of three `Loop` skeletons where each computation depends on the borders of the decomposition, yielding different results with different decompositions. Additionally each `Loop` has a stop condition which depends on the output of the previous iteration, thus introducing overhead on the CPU side of the execution, which varies with the number of iterations.

The OpenCL and original Marrow versions can be compared because they both explicitly assume the decomposition duties. This is, however, not true in the multi-GPU

version of Marrow as these are created automatically yielding very different results. In both OpenCL and single-GPU Marrow versions, five predefined decompositions are created (four in the second Loop), and these are submitted for execution, sequentially in the former, and in parallel in the latter (as well as in multi-GPU Marrow).

Although multi-GPU Marrow features automated decompositions, we have tried to force the creation of five overlapping decomposition, however, we are unable to achieve the same decomposition within the second Loop, as well as, the single-GPU Marrow case-study currently does not support five concurrent execution, and due to time constraints, was not possible to fix for this document, rendering the comparison invalid. As such, this case-study has been omitted from comparisons with OpenCL and original Marrow.

5.5 Overhead against OpenCL

The library's overhead is evaluated versus a pure OpenCL implementation, using the aforementioned case-studies and systems.

5.5.1 Single-GPU Overhead

We begin by measuring the overhead of the framework in a single GPU environment, having as baseline the execution time of the OpenCL case-studies presented in Appendix A, Table A.1. The charts presented in this subsection show the speedups relative to these values. The evaluation of this overhead does not take into account overlapping executions as the OpenCL case-studies do not have this capability. The graphical representation of the speedup values for Filter Pipeline, FFT and N-Body can be found in Figure 5.1 while the values Saxpy and Segmentation can be found in Figure 5.2.

Filter Pipeline This case-study, shows a very low overhead in S_1, with a maximum of 3% on the smallest data-set. However, in S_2, there is a significant overhead, varying from 11% for the smallest data-set, and peaking at 38% on the medium sized one, while having a 22% in the largest data-set. It is currently unknown why this high overhead occurs solely in S_2, although it hints to some internal nuance within AMD's OpenCL platform.

FFT Within both systems the FFT shows very low overhead with a maximum of 1%, as there is a bottleneck in the data-transfers, due to its very large data-set sizes, overshadowing library-related overheads.

N-Body Due to the high-computation nature of the N-Body case-study, the overhead of tends to be low, as its main bottleneck is within the computation itself. However, the overhead is indeed noticeable due to the fact that Marrow has extra orchestration (within the Loop) for the multi-GPU support, while a single-GPU implementation does not required it.

In S_1 this overhead is not very noticeable, peaking at 2% due to the fact that the GPUs performance is bottlenecking the whole case-study.

In S_2 however, the GPU is more powerful, lowering the bottleneck shadow, thus presenting a higher overhead peaking at 11% at the smallest data-set but reverting once again to 2%, akin to S_1 in the largest, as the computation bottleneck becomes once again noticeable.

Saxpy In the smallest data-set of the Saxpy case-study, there is a 5% overhead, while in S_2 there is an actual speedup, instead of overhead. The cause for this is currently unknown, due to time constraints and OpenCL's opaque nature. Additionally, there were no anomalies detected when analyzing with AMD's CodeXL Profiler. In S_1, the remaining data-sets have $16 - 17\%$ overhead. In S_2 there is a very small overhead of approximately 1%.

Segmentation This case-study has speedup in S_1 of $2 - 11\%$, instead of overhead. Once again the reason for this behavior is unknown, as these are not consistent to the results in S_2, hinting once again to the OpenCL platform. In S_2 there is a 44% overhead on the smallest data-set, although this value is inflated as the case-study takes less than a millisecond to execute per run. On the medium-sized data-set there is a 17% overhead and no overhead at all in the largest.

Remarks Overall, Marrow presents a relatively low overhead, although with some inconsistent results between, and within both systems, among each case-study. On the contrary to results in S_1, our AMD platform S_2, presents considerable overhead in some of the case-studies. In S_1 there was an overhead between $0 - 17\%$ while in S_2, where there are the most inconsistent results, between $0 - 44\%$ overhead. These values are acceptable within S_1, however in S_2 these are occasionally unexpectedly high, and further study of its OpenCL platform is warranted.

5.5.2 Multi-GPU Overhead

To measure the overhead related to multi-GPU execution, a OpenCL Saxpy case-study was implemented with multi-GPU support. Additionally it is important to note that this implementation is completely different from the previously presented one. As multi-GPU support entails a whole new execution platform, we avoided using this version when comparing in other single-GPU sections. As OpenCL has a low-level programming model, and due to time constraints, Saxpy was the only case-study adapted to have multi-GPU execution. However, this case-study does not implement any auto-tuning, as the data-set is split evenly among all decompositions (in both systems).

The execution values referring to the multi-GPU OpenCL case-study are presented in Appendix A, Table A.2, and the overhead representation of Multi-GPU Marrow is depicted in Figure 5.3.

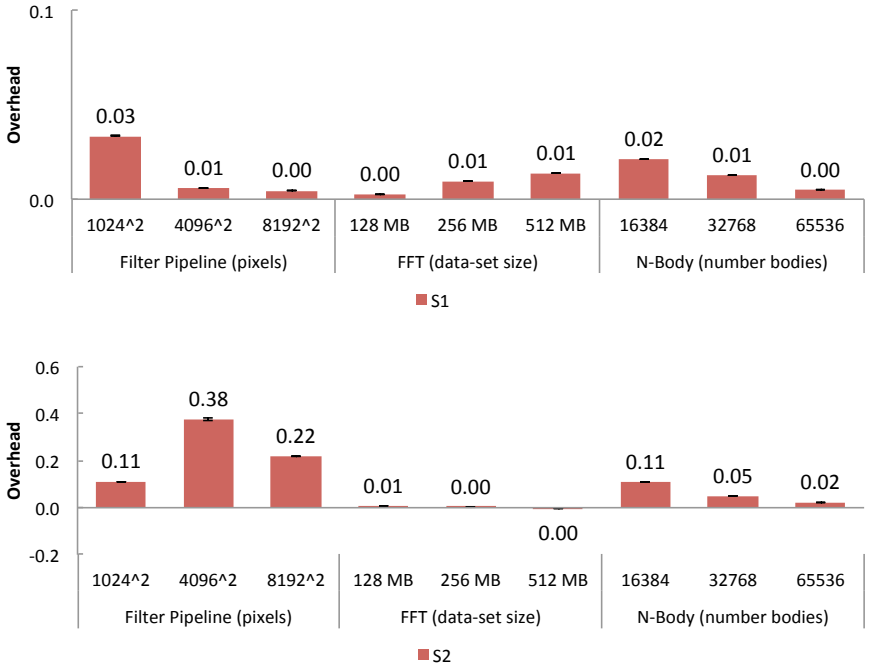

Figure 5.1: Single-GPU Overhead of Filter Pipeline, FFT and N-body in S_1 (top) and S_2 (bottom).

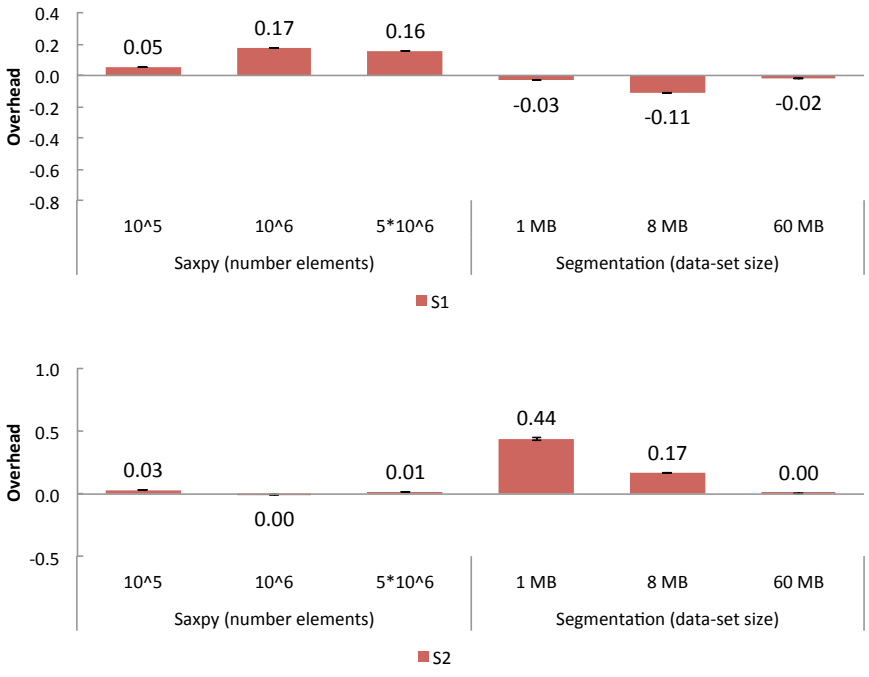

Figure 5.2: Single-GPU Overhead of Saxpy and Segmentation in S_1 (top) and S_2 (bottom).

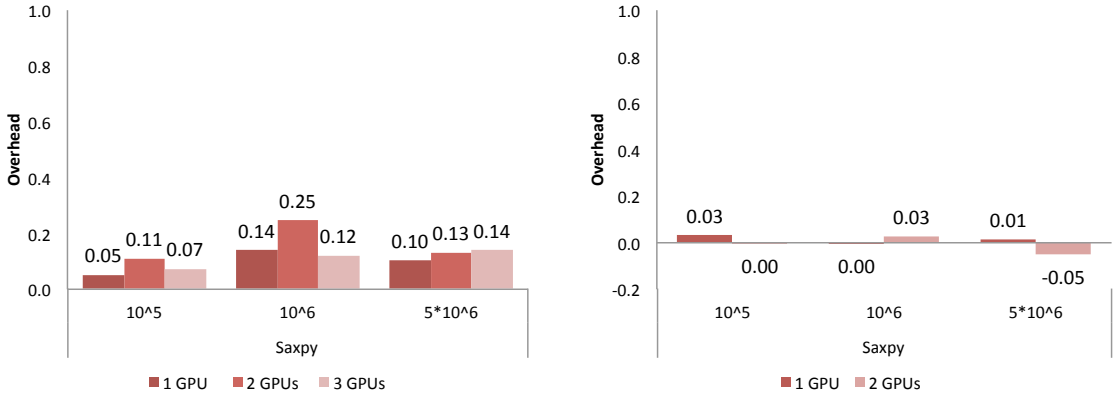

Figure 5.3: Multi-GPU Saxpy overhead in S_1 (left) and S_2 (right).

System 1 We can observe that in S_1 there is considerable overhead. However, this overhead is a static value across all data-set sizes, independently of the number of GPUs used: $\approx 0.3ms$ for smallest, $\approx 5ms$ for medium and $\approx 20ms$ for the large data-set. These static overheads imply that Marrow does introduce some overhead in its runtime platform, albeit it is independent of the number of GPUs that are used. Although the OpenCL implementation does not feature auto-tuning, as it splits evenly, this only incurs on a 6% discrepancy in the workload adaptation, which is not significant enough in this case to avoid the system's overhead.

System 2 In S_2, it is possible to observe very little overhead across all configurations and sizes. The differing overhead behavior from the previous system can be attributed to the CPU, as the superior performance of S_2's CPU shadows this overhead. In comparison with S_1's CPU, this system boast of a much newer architecture (almost three years apart) as well as supports a higher number of threads (6 cores with 12 threads versus S_1's 4 cores with 4 threads).

5.6 Comparison against Original Marrow

In this section, the performance of Marrow is evaluated in comparison to the original Marrow, using the best overlap results with a single GPU in both platforms. The original Marrow's was evaluation was performed by myself, and has contributed to a published paper in Euro-Par 2013.

The graphical representation of the speedups in relation to the execution times in original Marrow for Filter Pipeline, FFT and N-Body can be found in Figure 5.4 while the speedups for Saxpy and Segmentation are presented in Figure 5.5. The execution times of the original Marrow can be found in Appendix A, Table A.3.

Filter Pipeline This case-study presents differing behavior depending on the platform, presenting mainly slight overhead in S_1 and slight speedup in S_2, although we consider

these to be platform differences as they do not present a major impact in performance.

FFT In both S_1 and S_2, FFT shows a speedup, varying in the former between $\times 1.12 - 1.14$ and in the latter $\times 1.03 - 1.10$. This speedup is most likely connected to the already discussed organization of the execution platform in Subsection 4.4.4, where we use a single OpenCL context per command queue, whereas the original Marrow used a single, global context, which in these systems imply lesser performance.

N-Body There is a consistent overhead in this case-study in both systems, although decreasing with data-set sizes as it has a bottleneck in its computation, as previously mentioned. In S_1 this overhead is negligible, peaking at $\times 1.02$, while in S_2 it is more significant, having a peak of $\times 1.11$. This overhead is partially due to the fact that the original marrow uses a single For skeleton which does not contain as much orchestration as the multi-GPU version.

Saxpy This case-study shows a very good speedup over the original Marrow in both systems, where in S_1 this varies from $\times 1.28 - 1.76$ while in S_2, this maintains a lower degree of speedup varying from $\times 1.56 - 1.79$. We believe this speedup is due once again to the re-implemented execution platform which allows better overlap, using the new context configuration.

Segmentation There is a very low overhead in general in both systems, varying from $\times 1.01 - 1.04$ in S_1 and $\times 1.03$ in S_2, with the exception of the smallest data-set which consistently with the previously presented overhead study with pure OpenCL, a high overhead of $\times 1.44$ is observed. As the execution time is very low (lower than one millisecond), this overhead comes from our execution platform, as the original Marrow did not have such a complex platform, thus avoiding most of the overhead with very small data-sets.

Remarks This evaluation presented consistent results within each case-study, however, there are differing results among them, presenting overhead in some case-studies (Segmentation, N-Body), and sometimes consistent speedup (FFT). Saxpy presents some very high speedup which we suspect has to do with the new OpenCL context organization within our execution platform, adopted for better overlapping performance, but we are unable to confirm in useful time-frame for this document.

5.7 Multi-GPU Performance

While the previous evaluations focused in single-GPU performance (with or without overlap), this section is centered on multi-GPU scalability, to evaluate Marrow's viability as a multi-GPU platform.

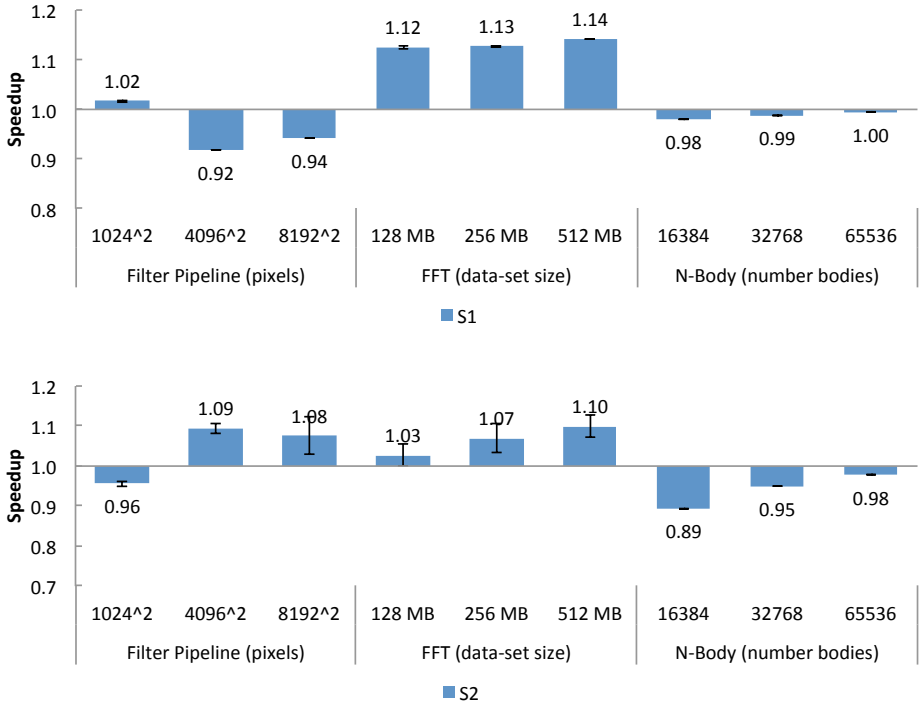

Figure 5.4: Single-GPU Speedup versus original Marrow of Filter Pipeline, FFT and N-body in S_1 (top) and S_2 (bottom).

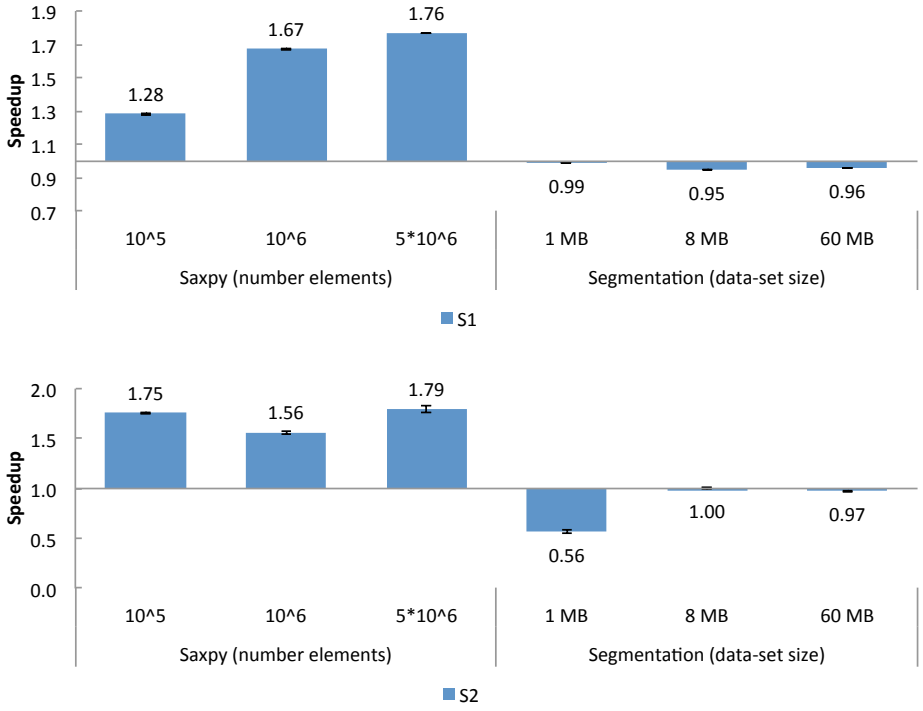

Figure 5.5: Single-GPU Speedup versus original Marrow of Saxpy and Segmentation in S_1 (top) and S_2 (bottom).

In Figure 5.6 the results versus Single-GPU for Filter Pipeline, FFT and N-Body are represented, while the results from Saxpy and Segmentation can be found in Figure 5.7. The values of single-GPU execution are (picking the best overlap result) are presented in Appendix A, Table A.4.

Filter Pipeline In both systems, this case-study presents scalability as expected. However in S_1 this scalability is very low considering there are 3 GPUs due to the PCIe bus bottleneck, with a speedup of $\times 1.37$ on the smallest data-set and $\times 1.55 - 1.57$ on the remaining data-sets. In S_2 there is a more considerable, as two GPUs present a speedup of $\times 1.27$ in the smallest data-set and $\times 1.56$ to $\times 1.67$ in the remaining data-sets respectively, due to the dedicated PCIe buses.

FFT As previously mentioned, this case-study has the largest data-sets. As such, it is not surprising to observe the lack of scalability in S_1 because of the shared PCIe bus among all three GPUs, which is acting as a bottleneck to the whole computation. In S_2, however, as there is a dedicated PCIe bus to each GPUs, speedup is achieved, varying from $\times 1.51$ in the smallest data-set, to $\times 1.88$ in the largest, as the computation is heavily focused in data-set size.

N-Body This case-study shows decent scalability in S_2 when dealing with larger number of bodies, peaking at $\times 1.76$. In S_1, however, we can observe super-scalability with a performance peak at $\times 5.68$ when using three GPUs. This behavior is attributed to the overlap, as without it, the speedup is $\times 1.65$ and $\times 2.19$ using two and three GPUs respectively, for the biggest data-set (65536 elements). It is however very important to note that the actual execution times in S_1 are considerably higher than S_2's, as the latter is equipped with GPUs with a much higher throughput. Specifically the high speedup in S_1 translates to $158.28ms$ with three GPUs, while in S_2, although there is a lower speedup, the execution time is still relatively close, with $205.97ms$. The high standard deviation shown in this case-study when using three-GPUs in S_1 has to do with the fact that there are a high number of threads working concurrently, and synchronizing after each iteration, with a high number of transfers between the host/GPU, inherent to the Copy replication per overlapping decomposition, which is used to its full effect in these configurations ($\times 4$ per GPU).

Saxpy When testing Saxpy within S_1 with two GPUs we have observed that this there was a lack of scalability when dealing with larger data-sets. We believe this is due to the fact that the actual Saxpy computation kernel, as it is naïvely implemented, without taking advantage of caching. Thus global memory is accessed every time an element is required. In S_1, as previously mentioned the middle GPU is a FX 3800, which on the contrary to C1060's 512-bit memory bus, only has 256-bit. The 256-bit bandwidth is bottlenecking the computation with the high number of accesses to global memory,

slowing down the whole computation. When using three GPUs we can observe that we do get some speedup, peaking at $\times 1.31$, although still limited by the FX 3800, as it does not vary from two to three GPUs.

In S_2 it is possible to observe consistent scalability, which becomes close to linear with a large data-set, with a value of $\times 1.89$.

Segmentation In both systems we can observe and inflated speedup in the smallest data-set, as its execution time is very low. On the other data-sets both systems show consistent scalability, having a maximum of $\times 1.55$ in S_1 and $\times 1.71$ in S_2

Hysteresis This case-study takes advantage of the implicit parallelization of the computation of the loop's condition, when using multi-GPU as well as overlapping computations, as such it is not surprising to observe scalability in both systems consistently, as more CPU threads are used. In S_1, this scalability maxes at $\times 2.0$, while in S_2 it has a maximum value of $\times 1.90$

Remarks Overall, Marrow presents an expected scalability, specially in S_2, with the help of its dual PCIe buses. In S_1 the shared PCIe bus entails reduced scalability in filter pipeline or even the complete lack of scalability in FFT, due to its large data-sets. Additionally its heterogeneity might be the origin of some unexpected behaviors, such as in Saxpy, which is slowed down when using the FX 3800, which features a reduced memory access bandwidth, as this computation accesses global memory multiple times for each element directly.

5.7.1 Impact of Communication and Computation Overlap

Having observed the results of multi-GPU scalability, using the best communication and computation overlap results, it is necessary to also study the actual behavior this optimization has within the case-studies.

The charts representing the impact of this optimization can be found in Figure 5.8 for Filter Pipeline, FFT and N-Body while for Saxpy and Segmentation these are depicted in Figures 5.9. In Appendix A, Table A.5, there are the execution values of the non-overlapped execution using the maximum number of GPU devices in both systems, as the remaining results would be cumbersome to present and discuss within this chapter. As previously mentioned, there are two main configurations used: $\times 2$ and $\times 4$ overlapping decompositions per GPU.

In both systems, there is a common category of systems, where the overlap behavior is similar: Filter Pipeline, Saxpy and Segmentation. It is possible to observe a consistent overhead introduced by overlap when the data-sets are too small, being that in segmentation this is more obvious as it has the smallest data-set of all three. When a certain data-set size is achieved, the overlap begins to show significant speedups consistently on all three case-studies.

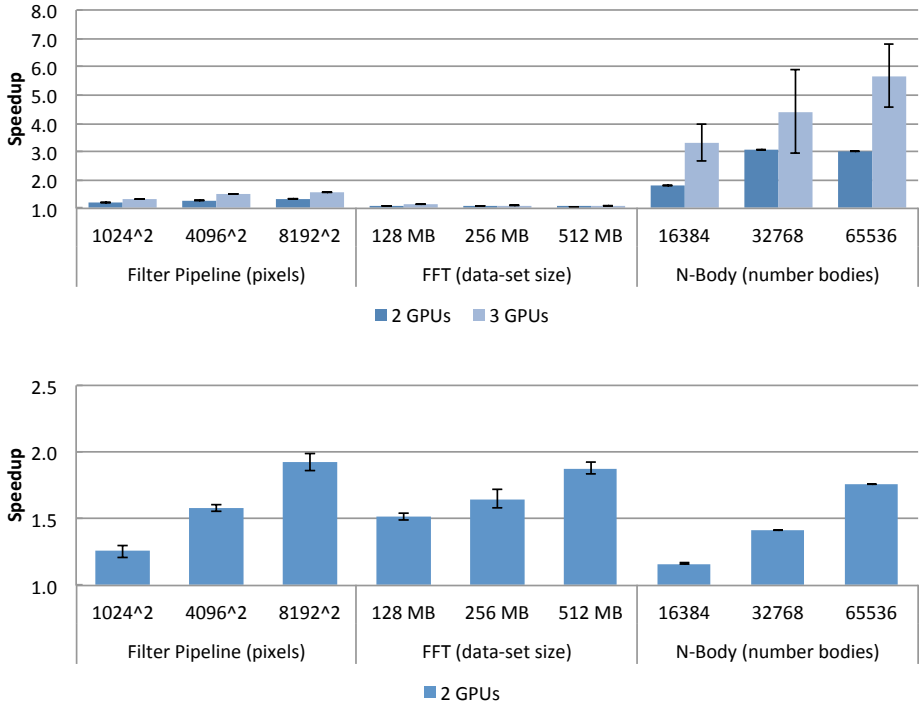

Figure 5.6: Multi-GPU scalability versus Single-GPU of Filter Pipe, FFT and N-Body in S_1 (top) and S_2 (bottom).

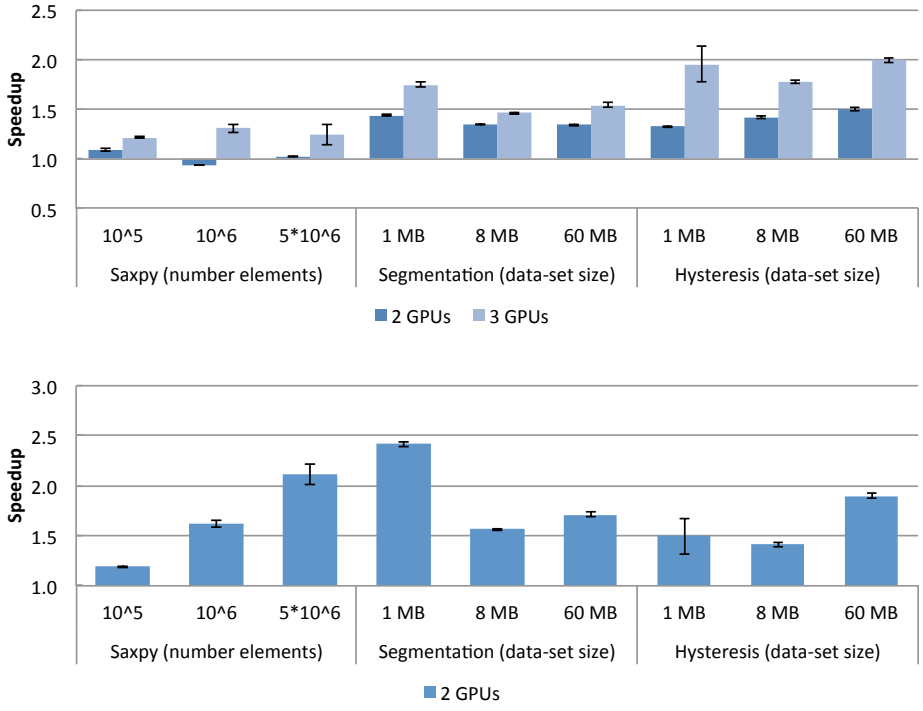

Figure 5.7: Multi-GPU scalability versus Single-GPU of Saxpy, Segmentation and Hysteresis in S_1 (top) and S_2 (bottom).

FFT Within S_1 there is a predictable lack of scalability due to the shared PCIe. In S_2 there is indeed scalability, although, when more than two overlap decompositions are used, the performance degrades, due to its large data-sets. Although not obvious at first, this case-study is the extreme of the previously presented behavior, where all the system overheads are hidden, and the bottlenecks in the transfers become obvious.

N-Body This case-study presents differing behaviors in both systems. This is mainly due to the differing performance between the GPUs present in both systems, where in S_2 there is no gain from overlap, as a single kernel execution is able to perform faster than an overlapped execution in S_1, for example, S_2 obtains $205.96ms$ in its with 65536 bodies using two GPUs, while S_1 takes $290.09ms$ using two GPUs with its best overlap configuration. When using three GPUs, however, S_1 becomes indeed faster versus S_2's two as expected.

Hysteresis As previously mentioned, this case-study takes advantage of multiple CPU threads, which in our system varies with the number of GPUs as well as overlapping decompositions. As such there is a consistent speedup when dealing with more overlap, as more CPU threads are introduced implicitly.

Remarks In regards to overlap, it is possible to identify a common behavior within some of the case-studies (Filter Pipeline, Saxpy and Segmentation), where in both systems, these have some overhead when dealing with small data-sets ($< 50MB$), but gain performance when above this threshold. Although this behavior is common between the systems, there is a much lower degree of speedup in S_1 due to its shared PCIe bus. The FFT is the extreme of this behavior, where the data-sets are so big, that all overhead is hidden, and the bottleneck within S_1's shared PCIe becomes embarrassingly obvious, while S_2, is still capable of speedup (although not as high as some of the other case-studies). Hysteresis shows a predictable scalability due to the usage of more CPU threads for the computation of the iteration's evolution condition.

Adopting an automated choice of overlap degree, requires several parameters which have to be taken into account, namely the aforementioned data-set size, the PCIe bus configuration (whether there is a linear speedup when transferring to multiple GPUs or not), and finally, computation's weight, to get an overview of its impact in relation to its data-set size. The first two features can be easily harvested, from the computation's configuration and using a benchmark at installation-time respectively. The last parameter however, is harder to obtain/estimate, as it either requires code analysis, empiric data (which is not always available) or user-submitted information. Although it is possible to automate this choice, implementing it correctly requires a great deal of effort.

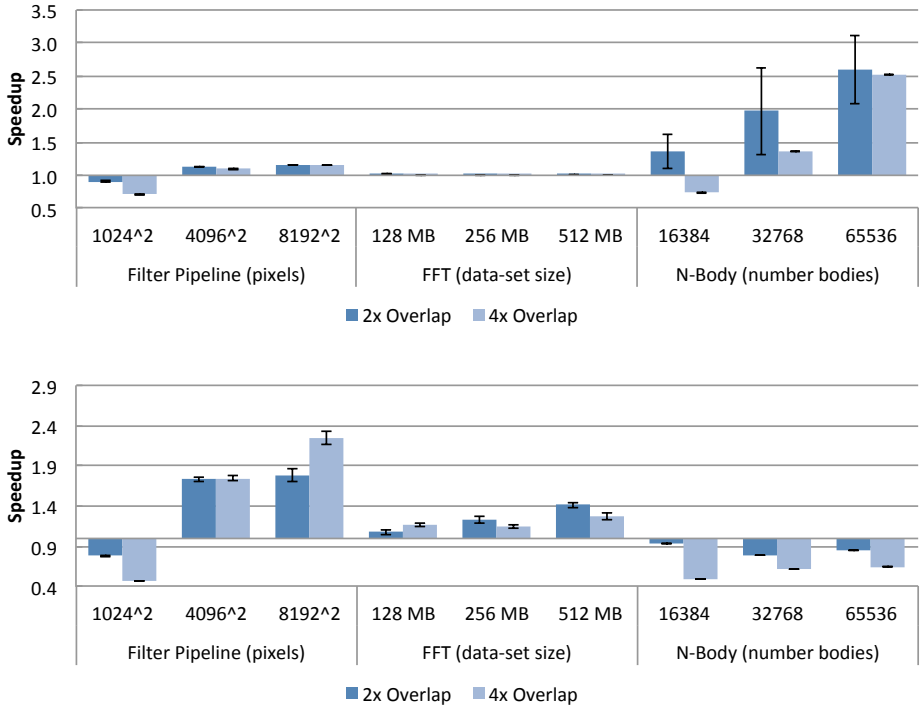

Figure 5.8: Overlap behavior with Filter Pipeline, FFT and N-Body in S_1 (top) and S_2 (bottom).

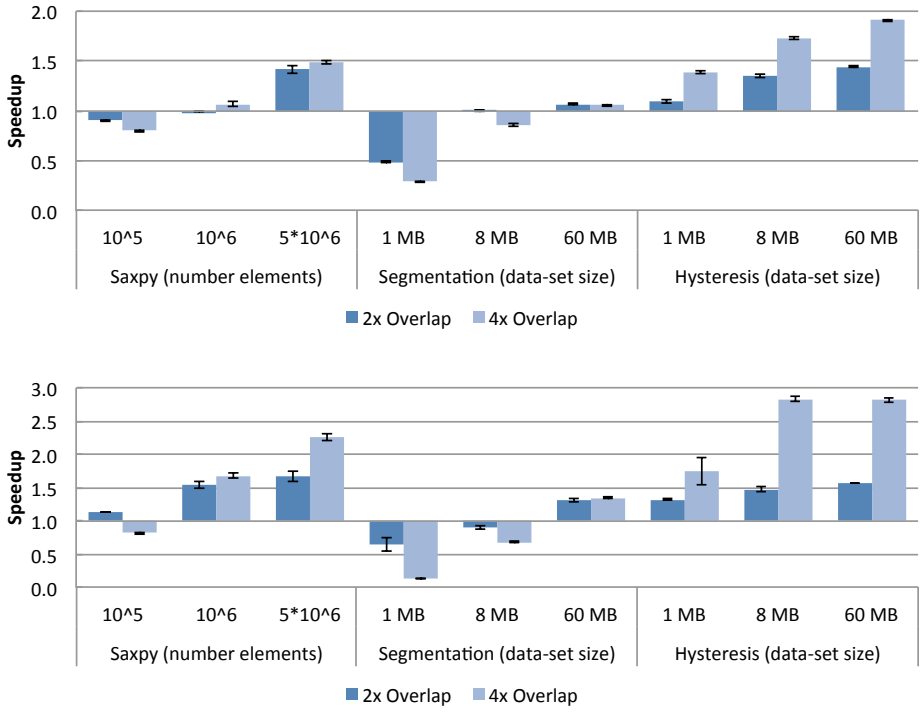

Figure 5.9: Overlap behavior with Saxpy, Segmentation and Hysteresis in S_1 (top) and S_2 (bottom).

5.8 Work Distribution

To evaluate the quality of the decompositions, we measured the execution times within each GPU, using the profiling option present in OpenCL's command queues. The system S_2 is omitted from this evaluation as it presents a homogeneous GPU configuration. From the six case-studies presented throughout this chapter, only three of them are presented in this evaluation: Filter Pipeline, Hysteresis and Segmentation. The remaining presented inconsistent/corrupt results and those are detailed in the end of this section.

The results are depicted in Figure 5.10, with the percentage of the total computation time each GPU took. Considering there are three GPUs, the optimal result is the one in which each takes $33.(3)\%$ to execute, implying a perfect adaptation to each GPU's performance, which can be seen in Filter Pipeline. Some work imbalance can be observed in both Hysteresis and Segmentation. As segmentation presents a lighter workload in comparison to Hysteresis, its imbalance only shows at the largest data-set with a 4% discrepancy from the optimal. In the latter, this imbalance becomes evident right away varying from 3% in the smallest data-set to 8% in the largest. This imbalance is due to the fact that both Hysteresis and Segmentation work on a three-dimensional workload. Recalling the Auto-Tuner's implementation, only the last dimension is decomposed, implying a coarse-grained adaptation of the decompositions, although it presents good results when using only two dimensions, as the Filter Pipeline shown.

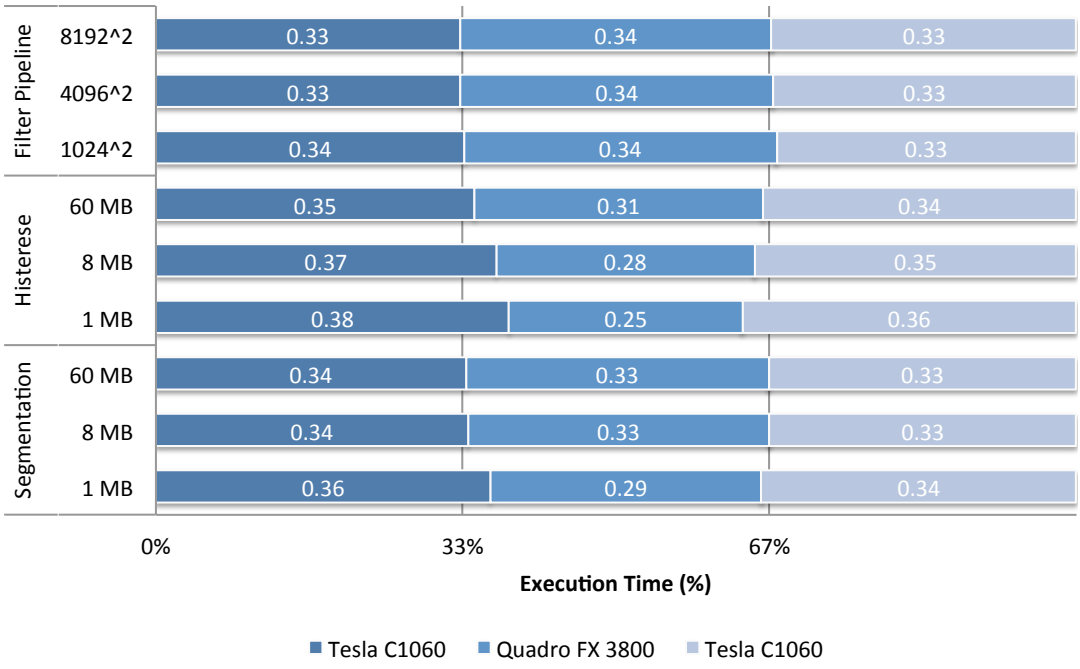

Figure 5.10: Filter Pipe, Hysteresis and Segmentation execution times within each GPU in S_1

Table 5.2: Multi-GPU Saxpy code size in OpenCL and Marrow.

Implementation	Init/Finish	Orchestration	Total
OpenCL	104	94	467
Marrow	18	38	188

Inconsistent Results As already mentioned, the remaining three case-studies, presented inconsistent/corrupt results, within the values obtained from the OpenCL profiling platform, therefore these were ignored from this evaluation. While execution times of the case-studies remained consistent, the actual timestamps obtained from the platform occasionally presented impossible results ($beginTimestamp > endTimestamp$), which we have explicitly ignored when obtaining the GPU execution times. This explicit measure was not enough, however, as there were still inconsistent results, as occasionally very high duration values continued to appear, throwing off the execution time within each GPU as well as the standard deviation to values higher than the average execution time. Furthermore, this timestamp corruption became more apparent when attempting to evaluate overlapping executions, hinting to a problem within the OpenCL profiling platform when dealing with concurrent kernel executions in both overlapping executions within each GPU, as well as to a lesser degree, in non-overlapping, multi-GPU executions.

5.9 Productivity

To attempt to measure productivity, we look at the code size of Marrow and the previously used multi-GPU Saxpy implementation using pure OpenCL. This case-study represents the simplest computation, in both OpenCL and Marrow, entailing a application of a computation, without any composed computations. Using skeleton nesting, composed computations favor Marrow even more than the values presented in this sections, as shown by Marques [Mar12] for single-GPU implementations.

In this section we consider productivity as the degree of detail a programmer is required to know as well as the degree of abstraction provided within the framework's constructs.

Although code-size does not translate directly into productivity, having presented the Marrow's implementation of this case-study and its semantics, this metric should provide a good indicator nonetheless.

The number of lines of code in initialization/finalization and execution orchestration stages, as well as the total number in each case-study (in each C++ source file) can be found in Table 5.2. It is possible to observe the significant gains inherent to the abstractions the algorithmic skeletons provide in all stages.

Init/Finish In the initialization and finalization stages, the algorithmic skeletons abstract all the verbose resource allocation/deallocation, management and error checking,

enabling a $\times 5.7$ less code written, in relation to its OpenCL counterpart, while providing an higher-level programming model which allows the creation of structured computations.

Orchestration In the orchestration stage of the execution, the algorithmic skeletons abstract over the underlying platform, while introducing implicit features and optimizations, such as multi-GPU support and communication and computation overlap, although the OpenCL example does not include the latter. It is possible to see a $\times 2.4$ less code, as there is no need to orchestrate concurrent execution for multi-GPU (and overlap), as well as data-decomposition which is explicitly defined in OpenCL implementation due to its low-level programming model.

Total Size In regards to total lines of code within the source file, Marrow does require more header inclusions to access the skeletons, although, still containing $\times 2.4$ reduction in overall code size, providing a higher-level abstraction to the cumbersome OpenCL programming model, while offering implicit scalability.

5.10 Final Remarks

Marrow's performance has been evaluated in this chapter, considering single-GPU overhead, multi-GPU scalability, workload distribution as well as programming productivity. Overall, Marrow presents good results, although at the expense of some expected, albeit occasionally heavy overhead.

Overhead There is an expected amount of overhead when using a library which provides such a high-level abstraction as Marrow. Within our study we have found some case-studies which do not present much overhead, due to their focus in either data-transfers, or heavy computation, such as FFT and N-Body. Additionally there are some issues with inconsistent results between both platforms, which we were not able to discern their origin in a useful time-frame for this document.

Scalability Depending on the architecture, all case-studies present consistent scalability, except for the FFT when using a shared PCIe bus (S_1). The absolute performance of the GPUs within each system influences the scalability results, as well as their overlap behavior, such as the N-Body case-study in S_2, which does not take advantage of overlap due to the powerful GPUs in its platform, while in S_1, this overlap makes all the difference.

Work Distribution This study has shown that the current Auto-tuner performs correctly when dealing with two dimensional workloads, although it does introduce some work

imbalance when dealing with three, or more dimensional workloads, as the decomposition only deals with the last dimension. This coarse-grained distribution introduces in our case-studies a maximum imbalance of 8%, which we consider acceptable.

Productivity Marrow enables excellent abstraction to the OpenCL's programming model, by handling most allocation/deallocation and management of resources implicitly ($\times 5.4$ less code) while enabling the creation of complex computations. Additionally, the execution of computations within Marrow is simpler, by introducing an implicit execution platform which further reduces code within this stage by $\times 2.4$. To this extent there is an overall reduction in approximately $\times 2.4$ of the code required for the creation of a multi-GPU Saxpy computation.

6

Conclusion

This chapter presents a summarized overview of this thesis, highlighting its goals in comparison with other algorithmic skeleton frameworks. The performance results are then presented briefly, as to enrich the this overview, from its effectiveness perspective. Finally we provide some guidelines for prospective future work.

6.1 Goals and Results

As a result of this thesis we have a functional prototype of Marrow enabled for multi-GPU execution, fulfilling our main goal. It enables the creation of skeleton computation trees which can be computed among multiple, and possibly heterogeneous, GPUs.

To this extent the computation trees are replicated to the GPUs, implying a data-centric parallelism with the usage of data decompositions within each of the replicated trees. This entails a new auto-tuning process, introduced in this prototype that adapts the size of the data decompositions accordingly to the GPUs' relative performance. An advantage of this process is that the programmer now enjoys a (mostly) automated decomposition while obtaining a performance-aware execution. This model also enabled the seamless transition of the previously data-parallel and task-parallel skeletons to the multi-GPU context, as most of the skeleton semantics remain the same.

The study of SkelCL and SkePU was essential on the adaptation of Marrow to a multi-GPU execution, as some of their concepts have been adopted by Marrow, such as the `Vector`, present in SkelCL. A runtime model akin to SkePU's runtime model has also been adopted, where the skeletons submit tasks, and these are delegated to a runtime platform for execution (in SkePU's case, StarPU).

The evaluation of our prototype showed a potential for scalability, although dependent on the underlying architecture. When faced with a shared PCIe bus, the speed of data-transfer among the GPUs falls considerably. This introduces a bottleneck within the whole performance, specially obvious in when applying overlap with large data-sets. As expected we can observe a high scalability in most case-studies when there is a dedicated PCIe bus, not only among the multiple GPUs, as well as when using overlap. In both cases the overlap should not be applied when applying computation smaller data-sets due to the overhead it entails. The only exception to this is the N-Body simulation which contains an exceptionally high computation requirements although having a smaller data-set.

Our static approach to auto-tuning presented good results, although degrading its quality slightly depending on the number of dimensions a computation uses, as it only takes into account the last dimension, entailing a coarser-grained adaptation.

This multi-GPU platform does entail some of overhead, in comparison with OpenCL and original Marrow, as we present a higher-level abstraction to the underlying platform, with implicit decomposition, and a new execution platform.

Overall we have accomplished all the goals set for this thesis, with generally good results.

6.2 Future Work

The distinguishable features of Marrow open several interesting research and implementation topics, such as cluster support, higher-level expressiveness of the kernel computations, dynamic scheduling strategies and new skeletons.

Generalizing Marrow to a cluster environment is a necessary step for additional scalability although, requires very careful planning and engineering, as introducing network communication entails a very high overhead when not used correctly (i.e small-sized problems, scheduling).

Currently the programmer has to supply Marrow with the OpenCL computation and configure it for a compatible execution. To boost productivity, ideally the programmer should express both computation and composition in C++, delegated on the framework the subsequent generation of the OpenCL code to run on the GPU side.

Finally the addition of new skeletons is important to improve Marrow's flexibility. In this context the Stencil skeleton is probably on of the most interesting constructs, which deals with the well-known challenges of dealing with shared borders within a data-set.

Bibliography

[Aco+10] A. Acosta, R. Corujo, V. Blanco, and F. Almeida. "Dynamic load balancing on heterogeneous multicore/multiGPU systems". In: *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS'10, Caen, France, June 28 - July 2, 2010*. IEEE, 2010, pp. 467–476.

[ADT03] M. Aldinucci, M. Danelutto, and P. Teti. "An advanced environment supporting structured parallel programming in Java". In: *Future Generation Comp. Syst.* (2003), pp. 611–626.

[AMP13] F. Alexandre, R. Marques, and H. Paulino. "Esqueletos Algorítmicos para Paralelismo de Tarefas em Sistemas Multi-GPU". In: *INForum 2013: Proceedings of INForum Simpósio de Informática*. Universidade de Évora, 2013.

[ATN09] C. Augonnet, S. Thibault, and R. Namyst. "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures". In: *Proceedings of the 15th Parallel Processing Workshops, HPPC, HeteroPar, PROPER, ROIA, UNICORE, VHPC, Euro-Par'09, Delft, Netherlands, August 25-28, 2009*. Springer, 2009, pp. 56–65.

[Aug+09] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Proceedings of the 15th International Parallel Processing Conference, Euro-Par'09, Delft, Netherlands, August 25-28, 2009*. Springer, 2009, pp. 863–874.

[BCJ10] R. Babich, M. A. Clark, and B. Joó. "Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics". In: *Proceedings of the 22nd ACM/IEEE Conference on High Performance Networking and Computing, Storage and Analysis, SC'10, New Orleans, LA, USA, November 13-19, 2010*. IEEE, 2010, pp. 1–11.

[Bac+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. "P^3L: A structured high-level parallel language, and its structured support". In: *Concurrency - Practice and Experience* 7.3 (1995), pp. 225–255.

[Bin+11] A. P. D. Binotto, C. E. Pereira, A. Kuijper, A. Stork, and D. W. Fellner. "An Effective Dynamic Scheduling Runtime and Tuning System for Heterogeneous Multi and Many-Core Desktop Platforms". In: *Proceedings of the 13th IEEE International Conference on High Performance Computing & Communication, HPCC'11, Banff, Alberta, Canada, September 2-4, 2011*. IEEE, 2011, pp. 78–85.

[Blu+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming, PPOPP'95, Santa Barbara, California, USA, July 19-21, 1995*. ACM, 1995, pp. 207–216.

[Bou+12] V. Boulos, S. Huet, V. Fristot, L. Salvo, and D. Houzet. "Efficient implementation of data flow graphs on multi-gpu clusters". In: *Journal of Real-Time Image Processing* (2012), 1–16.

[BH03] I. Buck and P. Hanrahan. *Data Parallel Computation on Graphics Hardware*. Tech. rep. 2003–03. Stanford University Computer Science Department, 2003.

[Cad+10] T. Cadavez, S. C. Ferreira, P. D. Medeiros, P. Quaresma, L. A. Rocha, A. J. Velhinho, and G. L. Vignoles. "A Graphical Tool for the Tomographic Characterization of Microstructural Features on Metal Matrix Composites". In: *International Journal of Tomography & Statistics* 14.S10 (June 2010), pp. 3–15.

[CAP] CAPS Enterprise and CRAY Inc. and The Portland Group Inc (PGI) and NVIDIA Corporation. *OpenACC*. URL: http://www.openacc-standard.org.

[CL07] D. Caromel and M. Leyton. "Fine Tuning Algorithmic Skeletons". In: *Proceedings of the 13th International Parallel Processing Conference, Euro-Par'07, Rennes, France, August 28-31, 2007*. Springer, 2007, pp. 72–81.

[CK88] T. L. Casavant and J. G. Kuhl. "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems". In: *IEEE Trans. Software Eng* 14.2 (1988), pp. 141–154.

[Che+09] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing". In: *Proceedings of the 4th IEEE International Symposium on Workload Characterization, IISWC'09, Austin, TX, USA, October 4-6, 2009*. IEEE, 2009, pp. 44–54.

[Che+10] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads". In: *Proceedings of the 5th IEEE International Symposium on Workload Characterization, IISWC'10, Atlanta, GA, USA, December 2-4, 2010*. IEEE, 2010, pp. 1–11.

[CCZ12] D. Chen, W. Chen, and W. Zheng. "CUDA-Zero: a framework for porting shared memory GPU applications to multi-GPUs". In: *SCIENCE CHINA Information Sciences* (2012), pp. 663–676.

[CA12] L. Chen and G. Agrawal. "Optimizing MapReduce for GPUs with effective shared memory usage". In: *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC'12, Delft, Netherlands, June 18 - 22, 2012*. ACM, 2012, pp. 199–210.

[CPK09] P. Ciechanowicz, M. Poldner, and H. Kuchen. *The Münster Skeleton Library Muesli–A Comprehensive Overview*. Tech. rep. ERCIS – European Research Center for Information Systems, 2009.

[CK10] P. Ciechanowicz and H. Kuchen. "Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures". In: *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications, HPCC'10, Melbourne, Australia, September 1-3, 2010*. IEEE, 2010, pp. 108–113.

[Col91] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.

[Col04] M. Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". In: *Parallel Computing* (2004), pp. 389–406.

[Comst] CompuGreen, LLC. *The Green500 List*. last visited in September 2013. URL: http://www.green500.org/lists/green201306.

[Dan+10] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. "The Scalable Heterogeneous Computing (SHOC) benchmark suite". In: *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU'10, Pittsburgh, Pennsylvania, USA, March 14, 2010*. ACM, 2010, pp. 63–74.

[Dan+12] A. Danner, J. Baskin, A. Breslow, and D. Wilikofsky. "Hybrid MPI/GPU Interpolation for Grid DEM Construction". In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems (formerly known as GIS), SIGSPATIAL'12, Redondo Beach, CA, USA, November 7-9, 2012*. ACM, 2012, pp. 299–308.

[Dar+93] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. "Parallel Programming Using Skeleton Functions". In: *Proceedings of the 5th International Parallel Architectures and Languages Europe Conference, PARLE'93, Munich, Germany, June 14-17, 1993*. Springer, 1993, pp. 146–160.

[DEK11] U. Dastgeer, J. Enmyren, and C. Kessler. "Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems". In: *Proceedings of the 4th International Workshop on Multicore Software Engineering, IWMSE'11, Waikiki, Honolulu, HI, USA, May 21, 2011*. ACM, 2011, pp. 25–32.

[Dub+12] C. Dubach, P. Cheng, R. M. Rabbah, D. F. Bacon, and S. J. Fink. "Compiling a high-level language for GPUs: (via language support for architectures and compilers)". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'12, Beijing, China, June 11-16, 2012*. ACM, 2012, pp. 1–12.

[EAH77] K. A. El-Ayat and J. A. Howard. "Algorithms for a self-tuning microprogrammed computer". In: *Proceedings of the 10th annual workshop on Microprogramming, MICRO'77, Niagra Falls, New York, USA, September, 1977*. IEEE Press, 1977, pp. 85–91.

[Els+07] E. Elsen, V. Vishal, M. Houston, V. S. Pande, P. Hanrahan, and E. Darve. "N-Body Simulations on GPUs". In: *CoRR* (2007).

[EDK10] J. Enmyren, U. Dastgeer, and C. Kessler. "Towards a Tunable Multi-Backend Skeleton Programming Framework for Multi-GPU Systems". In: *Proceedings of the 3rd Swedish Workshop on Multi-Core Computing, MCC'10, Göteborg, Sweden, November 18-19, 2010*. 2010, pp. 1–6.

[EK10] J. Enmyren and C. W. Kessler. "SkePU: a multi-backend skeleton programming library for multi-GPU systems". In: *Proceedings of the 4th international workshop on High-level parallel programming and applications, HLPP'10, Baltimore, Maryland, USA, September 25, 2010*. ACM, 2010, pp. 5–14.

[EK12] S. Ernsting and H. Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters". In: *Int. J. High Perform. Comput. Netw.* (2012), pp. 129–138.

[Fan+04] Z. Fan, F. Qiu, A. E. Kaufman, and S. Yoakum-Stover. "GPU Cluster for High Performance Computing". In: *Proceedings of the 16th ACM/IEEE Conference on High Performance Networking and Computing, SC'04, Pittsburgh, PA, USA, November 6-12, 2004*. IEEE Computer Society, 2004, p. 47.

[GVL10] H. González-Vélez and M. Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers". In: *Softw., Pract. Exper* (2010), pp. 1135–1160.

[GLS10] L. Gu, X. Li, and J. Siegel. "An empirically tuned 2D and 3D FFT library on CUDA GPU". In: *Proceedings of the 24th International Conference on Supercomputing, ICS'10, Tsukuba, Ibaraki, Japan, June 2-4, 2010*. ACM, 2010, pp. 305–314.

[Hag+11] A. Hagiescu, H. P. Huynh, W.-F. Wong, and R. S. M. Goh. "Automated Architecture-Aware Mapping of Streaming Applications Onto GPUs". In: *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS'11, Anchorage, Alaska, USA, 16-20 May, 2011*. IEEE, 2011, pp. 467–478.

[Hua+06] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. "Design of High Performance MVAPICH2: MPI2 over InfiniBand". In: *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid, CCGrid'06, Singapore, 16-19 May, 2006*. IEEE Computer Society, 2006, pp. 43–48.

[Huy+12] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh. "Scalable framework for mapping streaming applications onto multi-GPU systems". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'12, New Orleans, LA, USA, February 25-29, 2012*. ACM, 2012, pp. 1–10.

[Kes+12] C. W. Kessler, U. Dastgeer, S. Thibault, R. Namyst, A. Richards, U. Dolinsky, S. Benkner, J. L. Träff, and S. Pllana. "Programmability and performance portability aspects of heterogeneous multi-/manycore systems". In: *Proceedings of the 14th Design, Automation & Test in Europe Conference & Exhibition, DATE'12, Dresden, Germany, March 12-16, 2012*. IEEE, 2012, pp. 1403–1408.

[Kha+03] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owens, and B. Towles. "Exploring the VLSI Scalability of Stream Processors". In: *Proceedings of the 9th International Symposium on High-Performance Computer Architecture, HPCA'03, Anaheim, California, USA, February 8-12, 2003*. IEEE Computer Society, 2003, pp. 153–164.

[Khr12] Khronos Group. *The OpenCL Specification v1.2*. 2012.

[KMN12] T. Komoda, S. Miwa, and H. Nakamura. "Communication Library to Overlap Computation and Communication for OpenCL Application". In: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW'12, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, pp. 567–573.

[LBB11] M. E. Lalami, D. E. Baz, and V. Boyer. "Multi GPU Implementation of the Simplex Algorithm". In: *Proceedings of the 13th IEEE International Conference on High Performance Computing & Communication, HPCC'11, Banff, Alberta, Canada, September 2-4, 2011*. IEEE, 2011, pp. 179–186.

[Le+12] D. T. Le, H. D. Nguyen, T. A. Pham, H. H. Ngo, and M. T. Nguyen. "An Intermediate Library for Multi-GPUs Computing Skeletons". In: *Proceedings of the 9th International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future, IEEE RIVF'12, Ho Chi Minh City, Vietnam, February 27 - March 1, 2012*. IEEE, 2012, pp. 1–6.

[LM87] E. A. Lee and D. G. Messerschmitt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing". In: *IEEE Trans. Computers* (1987), pp. 24–35.

[LP10] M. Leyton and J. M. Piquer. "Skandium: Multi-core Programming with Algorithmic Skeletons". In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP'10, Pisa, Italy, February 17-19, 2010*. IEEE Computer Society, 2010, pp. 289–296.

[Lue+05] D. P. Luebke, M. Harris, N. K. Govindaraju, A. E. Lefohn, I. Buck, J. Krueger, T. Purcell, and C. Woolley. "Course on General-Purpose Computation on Graphics Hardware". In: *32nd International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'05, Los Angeles, USA, July 31 - August 4, 2005*. 2005.

[Lue+06] D. P. Luebke, M. Harris, N. K. Govindaraju, A. E. Lefohn, M. Houston, J. D. Owens, M. Segal, M. Papakipos, and I. Buck. "GPGPU: general-purpose computation on graphics hardware". In: *Proceedings of the 18th ACM/IEEE Conference on High Performance Networking and Computing, SC'06, Tampa, FL, USA, November 11-17, 2006*. ACM Press, 2006, p. 208.

[LHK09] C.-K. Luk, S. Hong, and H. Kim. "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'09, New York, New York, USA, December 12-16, 2009*. ACM, 2009, pp. 45–55.

[Mar+03] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. "Cg: a system for programming graphics hardware in a C-like language". In: *ACM Trans. Graph* (2003), pp. 896–907.

[Mar12] R. Marques. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". MA thesis. Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2012, pp. 1–131.

[Mar+13] R. Marques, H. Paulino, F. Alexandre, and P. D. Medeiros. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Springer, 2013, pp. 874–885.

[Meu+st] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. *The Top500 List*. last visited in September 2013. URL: http://www.top500.org/list/2013/06/.

[Mou+11] H. Mousazadeh, B. Marami, S. Sirouspour, and A. Patriciu. "GPU implementation of a deformable 3D image registration algorithm". In: *Proceedings of the 33rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC'11, Boston, Massachusetts, USA, 2011*. IEEE, 2011, pp. 4897–4900.

[NC11] C. Nugteren and H. Corporaal. *A modular and parameterisable classification of algorithms*. Tech. rep. ESR-2011-02. Eindhoven University of Technology, 2011, pp. 1–18.

[NC12] C. Nugteren and H. Corporaal. "Introducing 'Bones': a parallelizing source-to-source compiler based on algorithmic skeletons". In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU'12, London, England, 2012*. ACM, 2012, pp. 1–10.

[NVI08] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. 2008.

[NVI09] NVIDIA Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. 2009. URL: http://www.nvidia.co.uk/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[Pot+12] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda. "Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication". In: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW'12, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, pp. 1848–1857.

[RS11] M. Repplinger and P. Slusallek. "Stream processing on GPUs using distributed multimedia middleware". In: *Concurrency and Computation: Practice and Experience* (2011), pp. 669–680.

[SK09] D. Schaa and D. R. Kaeli. "Exploring the multiple-GPU design space". In: *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS'09, Rome, Italy, May 23-29, 2009*. IEEE, 2009, pp. 1–12.

[Sid+12] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzarán, and D. A. Padua. "Performance Portability with the Chapel Language". In: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium, IPDPS'12, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, pp. 582–594.

[SMV10] K. Spafford, J. S. Meredith, and J. S. Vetter. "Maestro: Data Orchestration and Tuning for OpenCL Devices". In: *Proceedings of the 16th International Parallel Processing Conference, Euro-Par'10, Ischia, Italy, August 31 - September 3, 2010*. Springer, 2010, pp. 275–286.

[SKG11] M. Steuwer, P. Kegel, and S. Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming". In: *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing Workshop, IPDPSW'11, Anchorage, Alaska, USA, 16-20 May, 2011*. IEEE, 2011, pp. 1176–1182.

[SKG12] M. Steuwer, P. Kegel, and S. Gorlatch. "Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library". In: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPSW'12, Shanghai, China, May 21-25, 2012*. IEEE Computer Society, 2012, pp. 1858–1865.

[Sto+07] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. "Accelerating molecular modeling applications with graphics processors". In: *Journal of Computational Chemistry* (2007), pp. 2618–2640.

[Str+08] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl. "CUDASA: Compute Unified Device and Systems Architecture". In: *Proceedings of the 8th Eurographics Symposium on Parallel Graphics and Visualization, EGPGV'08, Crete, Greece, 2008*. Eurographics Association, 2008, pp. 49–56.

[SO11] J. A. Stuart and J. D. Owens. "Multi-GPU MapReduce on GPU Clusters". In: *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS'11, Anchorage, Alaska, USA, 16-20 May, 2011*. IEEE, 2011, pp. 1068–1079.

[The10] The Portland Group. *PGI Accelerator Programming Model for Fortran & C, v1.3*. 2010.

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: *Proceedings of the 11th International Compiler Construction Conference, CC'02, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'02, Grenoble, France, April 8-12, 2002*. Springer, 2002, pp. 179–196.

[Tom+10] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. "Dense linear algebra solvers for multicore with GPU accelerators". In: *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing Workshop, IPDPSW'10, Atlanta, Georgia, USA, 19-23 April, 2010*. IEEE, 2010, pp. 1–8.

[VJ02] D. Vandevoorde and N. Josuttis. *C++ Templates: The Complete Guide*. Pearson Education, 2002. ISBN: 9780672334054.

[VD08] V. Volkov and J. Demmel. "Benchmarking GPUs to tune dense linear algebra". In: *Proceedings of the 20th ACM/IEEE Conference on High Performance Computing, SC'08, Austin, Texas, USA, November 15-21, 2008*. IEEE/ACM, 2008, p. 31.

[Yan+12] Y. Yang, P. Xiang, M. Mantor, and H. Zhou. "Fixing Performance Bugs: An Empirical Study of Open-Source GPGPU Programs". In: *Proceedings of the 41st International Conference on Parallel Processing, ICPP'12, Pittsburgh, PA, USA, September 10-13, 2012*. IEEE Computer Society, 2012, pp. 329–339.

[ZRL12] Z. Zhong, V. Rychkov, and A. L. Lastovetsky. "Data Partitioning on Heterogeneous Multicore and Multi-GPU Systems Using Functional Performance Models of Data-Parallel Applications". In: *Proceedings of the 1st IEEE International Conference on Cluster Computing, CLUSTER'12, Beijing, China, September 24-28, 2012*. IEEE, 2012, pp. 191–199.

[Zuc+11] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. “Using a "codelet" program execution model for exascale machines: position paper”. In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT'11, San Jose, California, USA, June 5th, 2011*. ACM, 2011, pp. 64–69.

Evaluation Tables

Table A.1: Single-GPU OpenCL execution times in milliseconds.

Case-study	Input Size	$S_1 - ExecTime(ms)$	$S_2 - ExecTime(ms)$
Filter Pipeline (pixels)	1024^2	3.66	3.02
	4096^2	56.14	22.60
	8192^2	209.50	116.34
FFT (MB)	128	74.37	61.25
	256	148.02	123.18
	512	294.84	251.44
N-Body (particles)	16384	73.07	33.72
	32768	242.64	99.09
	65536	871.13	353.89
Saxpy (elements)	10^5	3.79	3.87
	10^6	29.14	34.66
	$5 * 10^6$	145.74	143.31
Segmentation (MB)	1	1.50	0.66
	8	7.64	3.98
	60	43.85	31.59
Hysteresis (MB)	1	66.57	103.27
	8	532.04	581.89
	60	4661.55	2604.73

Table A.2: Multi-GPU Saxpy OpenCL execution times in milliseconds.

Case-study	Input Size	$S_1 - ExecTime(ms)$	$S_2 - ExecTime(ms)$
1 GPU	10^5	3.79	3.87
Saxpy (elements)	10^6	30.20	34.66
	$5 * 10^6$	145.74	143.31
2 GPUs	10^5	3.25	2.48
Saxpy (elements)	10^6	24.42	14.10
	$5 * 10^6$	119.88	75.56
3 GPUs	10^5	3.05	N/A
Saxpy (elements)	10^6	21.94	N/A
	$5 * 10^6$	140.91	N/A

Table A.3: Original Marrow execution times in milliseconds.

Case-study	Input Size	$S_1 - ExecTime(ms)$	$S_2 - ExecTime(ms)$
Filter Pipeline (pixels)	1024^2	3.85	2.63
	4096^2	40.72	21.52
	8192^2	167.70	75.58
FFT (MB)	128	70.83	48.10
	256	138.63	96.73
	512	275.65	193.96
N-Body (particles)	16384	73.15	33.84
	32768	242.77	99.20
	65536	871.25	353.99
Saxpy (elements)	10^5	5.12	5.28
	10^6	52.99	21.77
	$5 * 10^6$	254.07	108.92
Segmentation (MB)	1	1.44	0.67
	8	6.51	4.77
	60	41.21	27.934
Hysteresis (MB)	1	45.25	73.64
	8	330.45	314.66
	60	2796.17	1762.06

Table A.4: Multi-GPU Marrow execution times in milliseconds using a single GPU (best overlap results).

Case-study	Input Size	$S_1 - ExecTime(ms)$	$S_2 - ExecTime(ms)$
Filter Pipeline (pixels)	1024^2	3.79	2.75
	4096^2	44.33	19.65
	8192^2	178.04	70.30
FFT (MB)	128	63.00	46.86
	256	123.06	90.42
	512	241.51	176.54
N-Body (particles)	16384	74.65	37.83
	32768	245.75	104.36
	65536	875.45	361.95
Saxpy (elements)	10^5	3.99	2.94
	10^6	30.49	13.97
	$5 * 10^6$	142.35	60.79
Segmentation (MB)	1	1.45	1.18
	8	6.88	4.77
	60	42.88	28.68
Hysteresis (MB)	1	34.11	33.68
	8	172.42	67.12
	60	1609.07	577.95

Table A.5: Multi-GPU Marrow execution times in milliseconds using a single GPU (no overlap).

Case-study	Input Size	$S_1 - ExecTime(ms)$	$S_2 - ExecTime(ms)$
Filter Pipeline (pixels)	1024^2	3.79	3.40
	4096^2	56.46	36.29
	8192^2	210.47	148.88
FFT (MB)	128	74.56	61.67
	256	149.45	123.43
	512	298.98	250.66
N-Body (particles)	16384	74.65	37.83
	32768	245.75	104.36
	65536	875.45	361.95
Saxpy (elements)	10^5	3.99	4.01
	10^6	35.15	34.52
	$5 * 10^6$	172.69	145.28
Segmentation (MB)	1	1.45	1.18
	8	6.88	4.77
	60	42.88	31.74
Hysteresis (MB)	1	49.91	47.39
	8	350.84	195.86
	60	3304.87	1697.65