Ricardo Jorge dos Santos Marques

Computer Science Graduate

Algorithmic Skeleton Framework for the Orchestration of GPU Computations

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Hervé Paulino, Prof. Assistente, Universidade Nova de Lisboa

Júri:

Presidente: Pedro Barahona

Arguente: Leonel Sousa

Vogal: Hervé Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Outubro, 2012

Algorithmic Skeleton Framework for the Orchestration of GPU Computations

Eu dedico esta tese a todos aqueles que, de uma forma ou de outra, contribuiram para que ela se torna-se possível. Esta tese não é so minha, é também vossa.

Acknowledgements

I would like to start by thanking Professor Hervé Paulino, without whom this work would not be possible. His dedication, his patience, and his constant availability were absolutely fundamental to my thesis. I would also like to thank Professor Pedro Medeiros for his assistance in numerous occasions throughout my thesis. I extend my gratitude towards FCT-MCTES for financing the equipment, namely the GPUs, I used in this project. I thank the department of computer science of Faculdade de Ciências e Tecnologias, da Universidade Nova de Lisboa, (DI FCT UNL) for providing me with conditions to carry out this project.

My thesis also had indirect contributions, by certain individuals. My thanks to Pedro Severino for all the countless hours spent discussing solutions and sharing ideas. Similarly my thanks to André Mourão for providing so many hilarious moments, that helped relive the stress and, consequently, think more clearly. A big thanks to Tania Leitão and Filipe Carvalho for providing a friendly smile, when needed most.

A huge Thank You to Soraia Assis, whose friendship got me by the rough times, and whose smile warmed my own. Without you this thesis would be but a dream.

I thank my parents for all the sacrifices that they endured, so that I would be fortunate enough to study, and eventually reach this mark in my life.

Thank you to all my family members and friends, whose names were not mention so far. They are the silent heroes, whose word is spoken throughout this thesis.

Obrigado a todos!

Abstract

The Graphics Processing Unit (GPU) is gaining popularity as a co-processor to the Central Processing Unit (CPU), due to its ability to surpass the latter's performance in certain application fields. Nonetheless, harnessing the GPU's capabilities is a non-trivial exercise that requires good knowledge of parallel programming. Thus, providing ways to extract such computational power has become an emerging research topic.

In this context, there have been several proposals in the field of GPGPU (General-purpose Computation on Graphics Processing Unit) development. However, most of these still offer a low-level abstraction of the GPU computing model, forcing the developer to adapt application computations in accordance with the SPMD model, as well as to orchestrate the low-level details of the execution. On the other hand, the higher-level approaches have limitations that prevent the full exploitation of GPUs when the purpose goes beyond the simple offloading of a kernel.

To this extent, our proposal builds on the recent trend of applying the notion of algorithmic patterns (skeletons) to GPU computing. We propose Marrow, a high-level algorithmic skeleton framework that expands the set of skeletons currently available in this field. Marrow's skeletons orchestrate the execution of OpenCL computations and introduce optimizations that overlap communication and computation, thus conjoining programming simplicity with performance gains in many application scenarios. Additionally, these skeletons can be combined (nested) to create more complex applications.

We evaluated the proposed constructs by confronting them against the comparable skeleton libraries for GPGPU, as well as against hand-tuned OpenCL programs. The results are favourable, indicating that Marrow's skeletons are both flexible and efficient in the context of GPU computing.

Keywords: Algorithmic Patterns (Skeletons), GPU Computing, OpenCL

Resumo

A Unidade de Processamento Gráfico (*GPU*) tem vindo a ganhar popularidade como co-processador à Unidade de Processamento Central (*CPU*), devido à sua capacidade de ultrapassar o último em termos de desempenho em certas classes de aplicação. No entanto, aproveitar as capacidades do *GPU* não é uma tarefa trivial, requerendo bons conhecimentos de programação paralela. Assim, fornecer maneiras de extrair tal poder computacional tornou-se um tópico de investigação imergente.

Neste contexto, tem surgido várias propostas na área do desenvolvimento *GPGPU* (Computações de uso geral em Unidades de Processamento Gráfico). Contudo, muitas destas ainda oferecem um abstração de baixo-nível ao modelo de computação *GPU*, obrigando a que os programadores tenham que adaptar as computações das aplicações de acordo com o modelo *SPMD*, bem como em orquestrar os detalhes de baixo-nível inerentes à execução. Por outro lado, as aproximações de alto-nível possuem limitações que impedem o aproveitamento dos *GPUs* quando não se pretende só executar um *kernel*.

Neste sentido, a nossa proposta baseia-se na recente prática de aplicar a noção de padrões algorítmicos (skeletons) à computação *GPU*. Propomos Marrow, uma biblioteca de padrões algorítmicos, que expande o conjunto de *skeletons* atualmente disponíveis neste campo. Os *skeletons* da Marrow orquestram a execução de computações OpenCL e introduzem optimizações que sobrepõe comunicação com computação, agrupando simplicidade de programação com aumento de desempenho em várias aplicações. Além disso, estes *skeletons* podem ser combinados (*nested*) de forma a criar aplicações mais complexas.

Nós avaliámos as construções propostas comparando-as com bibliotecas para *GPGPU* semelhantes, bem como com programas OpenCL base. Os resultados são favoráveis, indicando que os *skeletons* da Marrow são flexíveis e eficientes no contexto *GPGPU*.

Palavras-chave: Padrões Algorítmicos (Skeletons), Computação GPU, OpenCL

Contents

List of Figures

List of Tables

Listings

1

Introduction

1.1 Motivation

The future of computing befalls upon parallelism. As transistors continuously get smaller, following Moore's Law [Moo65], today's microprocessor development is mainly concentrated on the addition of execution cores, rather than on the increase of single-thread performance through higher clock speeds. Examples of this trend are the latest CPUs released by Intel, codename Sandy Bridge, that feature up to six physical cores[1].

Another form of parallel processor is the graphics processing unit (GPU) that, despite starting off as a solution to a very domain specific problem (computer graphics), has been maturing into a powerful general processing unit. Recent GPUs surpass even the CPU in parallel performance and throughput in some particular classes of applications, not necessarily related to graphics processing. Hence, the use of GPUs as co-processors to the CPU is, more and more, a popular computation strategy, that even motivated other approaches aimed at introducing heterogeneity in the CPU's architecture. For instance, the Accelerated Processing Units (APUs) are processors that have both GPU and CPU cores on the same dye. An example of such an architecture is AMD's Fusion [Bro10] processor.

Nevertheless, the GPUs were not always viewed as general purpose co-processors. In the late 1980s all the graphical computations were simple 2D vectorial operations, with great performance concerns in terms of screen drawing speed. Originally, these computations were performed by the CPU, with poor performance results, due to the inadequacy of the latter's architecture. This inefficiency dictated that the evolution of

[1] Information taken from http://ark.intel.com/products/codename/29900/Sandy-Bridge

computer architectures was directed to a path of offloading demanding and domain specific computations to a dedicated processor, giving birth to the GPU. Towards the end of the twentieth century, as 3D accelerated applications gained popularity, due to the gaming market, new graphics APIs with 3D support were proposed (e.g., OpenGL [NDW97], and DirectX [BD98]). With time, the popularity of these APIs caused a relocation of an increasing number of calculation steps to the GPU.

Nowadays, the use of the GPU has crossed the boundaries of graphical computations. Scientific researchers embraced the GPU as an important support beam on which they build complex scientific models that require TeraFlops of computational power, an area commonly referred to as High-Performance Computing (HPC). Besides scientific computations, today the GPU is gaining popularity as a possible way of improving the performance of many general-purpose applications, that run in everyone's computers (e.g., databases [BS10], anti-virus [Kas09], so on and so forth).

GPU architecture differs a lot from the usual CPU design, as it is designed to take advantage of applications that fit well in the data-parallel computing model (expatiated below), where throughput is more important than latency. However, early attempts to harness the computational power of these processors for non-graphic applications meant mapping them to a graphics API, and structuring them in terms of the graphics pipeline. In most cases these tasks were not at all trivial, driving away most inexperienced programmers. Such attempts were recognised by Mark Harris in 2002, subsequently coining this trend as general-purpose computation on graphics processing units (GPGPU) [Har05].

As the GPU evolved into the general-purpose fully programmable processors of today, there was an emergence of some general-purpose programming languages that were more programmer friendly, and allowed for a greater control over the execution of the graphics pipeline. The first popular GPGPU language was NVIDIA's CUDA [NVI]. Because of its C-like syntax, it introduced a smaller learning curve to those who wished to create applications that executed on GPUs. This made CUDA very popular among scientific researchers because it allowed the use of NVIDIA graphics cards for computing scientific models, thus, increasing their efficiency by removing limitations imposed by computational power. However, CUDA is not an industrial standard, and its code is not portable to architectures that are equipped with chips not manufactured by NVIDIA. Later on, as the need for such a standard in GPGPU programming architectures grew, Khronos (a consortium of corporations - whose members feature Intel, AMD, NVIDIA, Google, and many others - bent on creating open standard APIs) saw fit to create the first standard language for GPGPU development. The result was OpenCL [Mun+09], today's standard for GPGPU APIs. It provides a broad set of programming APIs based on past successes. Moreover, it defines core functionality, supported by all platforms, as well as optional functionalities for high-function devices.

The previous GPGPU languages provide a programming model that abstracts the underlying platform model (GPU). They divide the computation in two categories: host,

and device. The host computations, usually processed by a primary unit such as the CPU, orchestrate and issue device executions. In turn, the device runs kernel functions, which are the parallel computations executed by one or more compute units (e.g., CPU cores, cores on a stream multiprocessor). This model will be further elaborated in Chapter 2. As it stands, the execution models of these GPGPU APIs gives support to the following parallel computing models:

- **Data-parallel** – Where concurrency is expressed as instructions from a single program applied to multiple independent partitions of a data structure.
- **Task-parallel** – Where computations are expressed in terms of multiple concurrent tasks that have independent instruction locksteps.

1.2 The Challenges of GPGPU

GPGPU APIs were a major development in GPU computing, giving developers a simpler way to use the resources available in GPUs. Nevertheless, using the previously mentioned GPGPU APIs is still a challenging exercise, since not only does the parallel part of the application needs to be structured according to the SMPD model (as described in Section 2.1), but also, the developer has to oversee many low-level programming concerns. The latter range from memory/resource management, to performing data transfers between host and device memories, or even synchronizing the host execution with the auxiliary computing device. On the other hand, if the developer decides to use a non-standard API, such as CUDA, he or she should be aware of the lack of portability attached to such an API. The acknowledgement of these limitations led to the proposal of several high-level GPGPU APIs (e.g., Aparapi [AMD11], Accelerator [TPO06], RapidMind [McC06]), covered in Section 2.3.

Even though these high-level frameworks provide a good level of abstraction relative to the underline platform model (GPUs), they only offer basic building blocks with which to build parallel applications, remaining still significantly close to the underline computing model (parallel computing), and/or not being able to extract the full potential of the architecture. For instance, although Accelerator's execution model is highly oriented towards arrays, it does not support more complex operations than simple element-wise arithmetic combinations (e.g., addition, subtraction, division) of arrays. Even worse, some platforms (e.g., Aparapi) do not even guarantee that the code created by the developer will in fact be executed on a GPU, since this execution is only supported if a transformation from source code to OpenCL code is feasible. These and other particular issues are discussed in more detail in Chapter 2.

It would be advantageous to software developers to have a framework that would provide them with guidelines, that would steer the design process towards good design models. At the same time, the framework would take care of any concerns native to parallel programming (e.g., synchronization, communication), and still be able to harness the

full potential of the architecture. These ideas are some of the main purposes of the recent approach that centres on the application of algorithmic patterns [Col91] (also known as skeletons) to the context of GPGPU development. Skeletons are essentially abstractions of commonly used parallel patterns, like the *MapReduce* or the *master-worker* skeletons (as described in Subsection 2.4). They are provided as algorithmic skeleton frameworks (ASkFs), that contain recurring structures, and behaviours, associated to parallel programming.

To the best of our knowledge, only three platforms that support this type of abstraction on GPUs have been released, namely SkelCL [SKG11], SkePU [EK10], and Muesli [EK12]. Nonetheless, this work is still very preliminary and does not go much beyond supporting the *MapReduce* skeleton and some variants, leaving out skeletons like *Pipeline*, and *Loops*. These types of skeletons have been successfully implemented in other areas (e.g., clusters [CL07], multi-core CPUs [LP10]), and are useful in GPUs since they aggregate multiple computational steps within a persistent memory utilization scheme, not requiring memory transfers between steps. On the other hand, the mechanisms that these libraries provide for the developer to adapt a skeleton's execution are somewhat limiting – in SkePU developers use a macro language, in SkelCL executions are defined and issued as strings, and in Muesli developers are limited to pre-defined data types. In the end of Section 2.4 we give more detail about these and other issues.

When using any of the previous skeleton platforms, developers have no control over how the communication is overlapped with the computations. This obviously simplifies application development, although, if the developer intends to increase general performance by controlling this overlap, it is not possible. For example, a developer may intend to create an application that applies the same execution successively to multiple different data-sets. Given that, the decoupling of the CPU's and GPU's address spaces implies memory transfers, the best executional strategy would be to concurrently send new input data-sets to the device, as it computes upon an older data-set. This would enable the GPU to begin processing the next input immediately after finishing its previous execution, consequently reducing its idle time. However, this is not a particularly easy technique to apply and only Muesli supports this kind of optimization, although is is not parametrizable. On the other hand, none the previous platforms supports nesting of GPU skeletons.

1.3 A High-Level Skeleton Framework for GPU Computing

Our proposal builds on the idea of applying the notion of skeletons to the context of GPGPU development. We intend to solve some of the previously mentioned issues by developing an ASkF that is mainly focussed on orchestrating the execution of OpenCL kernels, and offers a varied set of data- and task-parallel skeletons. By not delving into the domain of the parallel computations (kernels) we were able to propose a framework that has a rich set of constructs, that can still support the major functionalities offered

by the OpenCL language. The set of proposed skeletons include some that are already present in the context of CPUs and clusters (e.g., *Pipeline*), and others there are completely new to this context (e.g., *Stream*), as far as we know. Also, only in the interests of completeness, our ASkF supports common GPU skeletons, like the *MapReduce*. We were primary interested in skeletons whose execution behaviour was not hampered by the logical, and physical, division between the host and device address spaces. Skeletons whose execution is based on persistent data schemes are attractive to us because they do not require data-transfers when combining distinct execution instances. In this way, they avoid the overheads associated to transfers between disjunct memory spaces. For example, consider a N-staged *Pipeline*. If executed on the GPU, the results of stage i, where $0 < i < N$, do not have to be transferred back to main memory in order to be available to the subsequent stage $(i + 1)$. There are other examples of such skeletons. The entire set supported by our ASkF is presented in Chapter 3.

We deemed as very important to allow the combination, or nesting, of skeletons as it enables developers to build complex executional structures, possibly containing very distinct behaviours, in a very simply and efficient manner. This technique is also beneficial in terms of performance, in that it is compatible with a disjoint memory scheme. An application may apply a successive collection of computations, in the form of skeletons, to an input data-set, and only carry out memory transfers when: writing the input to device memory, and reading the results to main memory. Furthermore, the nesting mechanisms helps the skeleton design to remain simple. A skeleton may only provide a very specific behaviour, since more complex structures are created by nesting multiple skeletons.

We also sought to allow the developer to seamlessly introduce performance gains by having the skeletons overlap communication and computation. By doing so, the skeletons can make better use of the parallelism qualities of modern GPUs, and increase overall productivity. Having the skeletons transparently apply this execution strategy obviously facilitates the developer's job. He or she can develop efficient applications without having a large degree of knowledge in both, parallel programming, and the OpenCL language features.

In essence, our approach is to design and implement a high-level skeleton framework for the orchestration of GPU computations that encompasses the above concerns: performance (by overlapping communication with computation) and modularity (by supporting nesting of skeletons).

1.4 Contributions

The main contribution is C++ ASkF, named *Marrow*, for the orchestration of OpenCL kernels, that introduces new skeleton constructs within that scope. Marrow is supported by a OpenCL runtime, that provides a standard execution model across multiple heterogeneous parallel architectures, despite our main focus being in GPUs. Additionally,

Marrow's constructs can be nested, which to the best of our knowledge is a functionality not supported by any GPGPU ASkF. The nesting enables the development of intricate parallel applications without having to manually connect distinct computational models, and implicitly avoiding heavy memory transfers. These resulting applications transparently use overlap between communication and computation as a way to increase overall performance. Overlap reduces the overhead introduced by constant memory transfers and gives a virtual sense of persistence to the computations. Moreover, it enables us to exploit concurrency between computations and memory data transfers, increasing the application's parallel factor.

Another key contribution is fact that Marrow has an OpenCL based implementation. This highly increases the portability of the code produced when using Marrow, considering that if the underlying platform supports OpenCL, then it is very likely that it fully supports Marrow. Naturally, there are other portability concerns derived from the use of the C++ language, namely using a compiler that fully supports C++11. However, this issue does not significantly impair our library's portability.

Yet another contribution is the comparative evaluation that we performed. We compared Marrow in terms of performance and programming model productivity, against OpenCL, SkePU, and SkelCL. We left out Muesli, since the latter enables GPU executions through CUDA, consequently distancing itself from our direct research focus.

1.5 Document Structure

The remainder of this document is structured as follows:

Chapter 2 Presents the state of the art associated to our research area. Naturally, our research area is associated to GPGPU technologies at different levels (e.g., low-level, high-level, algorithmic patterns), so the most relevant technologies at each level are described. Additionally, an overview about GPU architecture is given.

Chapter 3 Expatiates on the developed C++ ASkF, Marrow. The chapter discusses all of Marrow's core features and functionalities, namely: the supported skeletons, nesting mechanism, overlap between communication and computation, and others.

Chapter 4 Presents Marrow's architecture, as well as its most relevant implementation details.

Chapter 5 Validates the developed library, by presenting and discussing experimental results that evaluate Marrow's performance and programming model against technologies of lower and higher abstraction level.

Chapter 6 Summarises key aspects about the work carried out in this dissertation. Particularly, it reflects on the achievement of the proposed objectives; on the contribution of Marrow to the current state of the art on GPU computing; and discusses possible future research topics, within Marrow's context.

2

State Of The Art

To better understand the GPU's potential for general-purpose computing, and what kind of attempts have been made to exploit its capabilities, this chapter presents a description of the most relevant GPGPU technologies in general GPU computing, and particularly in regards to our proposal. This chapter begins by providing a brief overview about the evolution of GPU architecture (Section 2.1), based on [OHLGSP08]. Then, GPGPU technologies are presented, from lowest (Section 2.2) to highest (Section 2.3) level. Subsequently, an alternative to the common methodologies of high-level GPGPU platforms is presented, in the form of computational patterns (Section 2.4). Lastly, our final considerations about the current state of the art, in regards to GPGPU development, are given in (Section 2.5). Should be noted that only the platforms that have a direct impact in our work, either by development support or by intellectual contribution, are provided with programming examples.

2.1 GPU Architecture Evolution

It all begins at the graphics pipeline, the basis for the GPU's architecture. It is where the input, a list of geometric primitives, is shaded and mapped onto the screen. The steps in the canonical pipeline are:

1. **Vertex Operations** – Each vertex from the input primitives is transformed from a 3D position into a 2D screen coordinate (referred to as *the shading process*). Since vertices can be computed independently, this stage is well suited for parallel hardware.

2. **Primitive Assembly** – The vertices are assembled into triangles, that is the fundamental primitive in current GPU architectures.

3. **Rasterization** – This is the process of determining which screen-space pixel locations are covered by each triangle. Every triangle generates a primitive called a *fragment* at each screen-space pixel location that it covers.

4. **Fragment Operations** – Determines the final color of each *fragment* by using the color information from the vertices and/or textures. Even though the elements in this step can be computed in parallel, this is typically the most computationally demanding stage in the graphics pipeline.

5. **Composition** – Finally, fragments are assembled into a final image, with one color per pixel.

First GPUs were composed of highly specialized function units, however, as they became faster in rendering basic computer graphics, demand for more sophisticated techniques grew. Most functionalities, supported by these early GPUs, were fixed functions. That is to say, if a new technique was developed and was not supported by the graphics API, the hardware had to be directly modified so that the API could be extended to support such functionality. There was not much room for the API to grow on top of the hardware because the latter had very specific and limiting functionalities, which greatly limit the possibilities of software developers.

The overcoming of these limitations required a major architectural switch of GPUs. Highly specialized function units were replaced by smaller and simpler processors, transforming the GPU's architecture into a SIMD architecture. A SIMD (Single Instruction Multiple Data) architecture is composed of multiple processing elements capable of executing the same operation on multiple data streams simultaneously, as defined by Flynn in [Fly72]. The GPU's hardware was composed of large amounts of processors for vertex and pixel specific calculations, increasing the flexibility of the hardware and providing the vendors with a scalable architecture.

Further refinements were made to the architecture over the years, like the unification of the vertex and pixels specialized processors into a single less specialized type with only one instruction set, capable of handling all the previous tasks. This increased the complexity of the processors, but also made them more flexible. That refinement is today denominated as the Unified Shader Model 4.0 [Bly06]. Also, modern GPUs provide a SPMD (Single Program Multiple Data) computing model where multiple processing elements execute the same program, having a synchronization lockstep at program level instead of at instruction level, as happens in a SIMD model. This means that the execution path between processing elements may differ.

Modern GPUs feature a big parallel pipeline with a high data throughput, also having programmable components as opposed to the original fixed function components. A good example of a unified and massively parallel programmable unit is the architecture of the NVIDIA Fermi, depicted in Figure 2.1. It features sixteen stream multiprocessors of thirty-two cores each, summing up to five-hundred and twelve processing cores. Each

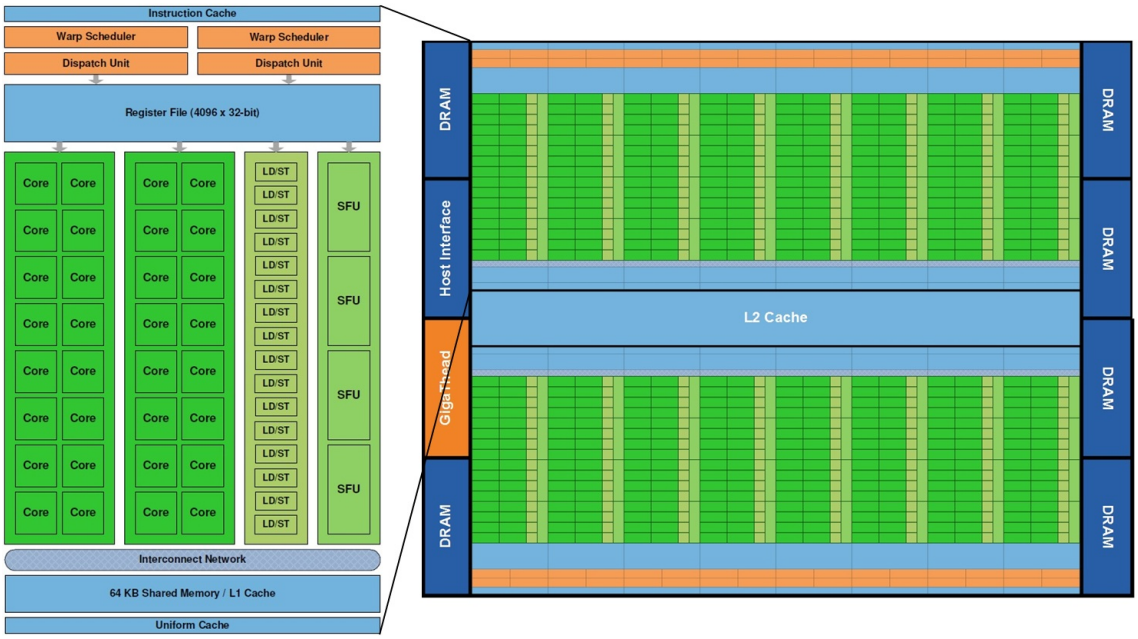


Figure 2.1: The architecture of the NVIDIA Fermi. Edited from figures in [Cor09]

stream multiprocessor has its own 64 KB L1 cache, shared by its thread processors, as well as a 16 KB register file. There is also a L2 cache that is shared by the stream multiprocessors.

Fully utilizing the potential of the modern GPU architecture is complex endeavour. The differences from the usual parallel programming methodology (CPU) are such, that there is a high probability of creating an inefficient GPU program, if the developer does not pay close attention to specifications of the underlying hardware. Considering only NVIDIA GPUs, and their terminology, one such difference is the fact that the smallest executable unit of parallelism on a device, a warp, comprises 32 threads. Stating that every modern NVIDIA GPU can support a minimum of 768 threads per multiprocessor [NVI09], a device that has 30 multiprocessors can have more than 30000 active threads. This parallel capability can be easily wasted if the number of threads is not sufficient to fill every available multiprocessor. Thus, executing a very small number of threads may present more overheads than actual performance benefits.

Typically, hundreds of threads are queued up for work (in warps). These GPU threads are extremely lightweight, in contrast with CPU threads. If a processor must wait on one warp of threads, it simply begins executing work on another. Because register are allocated to active threads, no swapping of registers and state occurs between GPU threads, and resources stay allocated to a thread until it completes its executions. This efficient warp swapping is usually beneficial, although very sensitive to the memory access patterns of threads. If the accesses to principal GPU memory (global memory) by threads within a warp are not coalesced, then general performance will be significantly reduced.

GPU global memory loads/stores by threads of a half warp (16 threads) are coalesced by the device in as few as one transaction, when certain access requirements are met. To

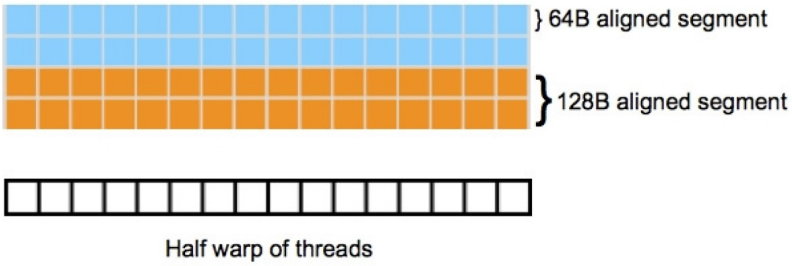


Figure 2.2: Linear memory segments and threads in a half warp, taken from [NVI09]

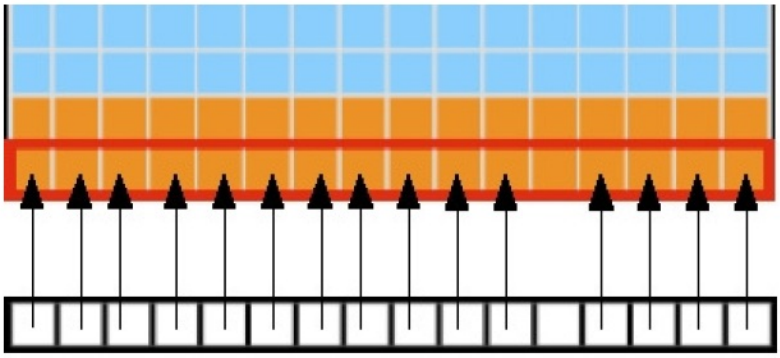


Figure 2.3: Coalesced access in which all threads but one access the corresponding work in a segment, taken from [NVI09]

understand these memory access recommendations, global memory should be viewed in terms of aligned segments of 16 and 32 words, as in Figure 2.2. It shows global memory as of 64-byte aligned segments. Two rows of the same color depict a 128-byte aligned segment. At the bottom of the figure is indicated a half warp of threads that accesses the global memory.

A simple coalesced access pattern is represented in Figure 2.3. The k-th thread accesses the k-th word in a segment, though it is not necessary that every thread participates. This access pattern results in a single 64-byte transaction (highlighted by the red rectangle). Even the unnecessary word is fetched from global memory. If just one of these threads, for example the one that did not participate, accessed a data value that was not stored in the first 16 words, then the half warp would required two full transactions before beginning its execution. Logically, increasing the number of transactions reduces performance, since the warps are left waiting rather then executing.

2.2 General Purpose Programming on GPUs

As the computational power of GPUs (and CPUs) grew it was noticed that for certain domains of computation, like data-parallel oriented applications, the use of GPUs outperformed even the most extremely optimized CPU versions of the algorithms. This reality fostered the proposal of new APIs, which were not exclusively for graphics programming: Instead, they allowed the development of more general-purpose applications. The

most relevant GPGPU APIs are subsequently presented.

2.2.1 Brook

Brook [BFHSFHH04] is a language that was designed to provide programmers with native support for data-parallelism, while still being efficient enough to allow the development of arithmetic intensive applications. Furthermore, it had portability concerns, enabling programs to run in various heterogeneous architectures. To accomplish these goals Brook provides various features, of whom the most important are: Streams, Kernels and Reductions.

Streams are the basic building blocks of Brook. They define a collection of data which can be operated in parallel. Each stream is composed of elements, whose domain falls in the primitive types provided by the language, such as floats. A similarity can be traced between streams and C arrays, although, access to a stream is restricted to a kernel.

Kernels are functions which are applied to every element of a stream. In order to obtain a data-parallel application of the kernels to the streams, programmers are forced to distinguish between the input data of a kernel, used as read-only, and the respective output data, used as read-write.

Reductions are the inverse of a kernel, that is, they provide a mechanism for calculating a single value from a collection of records. Reductions accept a single stream and output either a smaller stream of the same type, or a single element value. An example of reduction is summing all the integer elements of a stream. Reductions are still data-parallel.

Brook was the first effective approach to GPGPU development. However, due to the release of other GPGPU APIs, namely CUDA, it was never really adopted in the industry, being remitted as a case study for research activities.

2.2.2 Compute Unified Device Architecture

Compute Unified Device Architecture (CUDA) [NVI] was created by NVIDIA and released in 2007. It is a hardware and software architecture for issuing and managing data-parallel oriented computations on the GPU, without the need of mapping them to a graphics API.

Architecture overview

The CUDA software stack is composed of three layers, as shown in Figure 2.4. An important aspect of CUDA is that it provides general memory addressing, that is, from a programming perspective the application can read and write data at any memory address, equivalent to a CPU and system RAM.

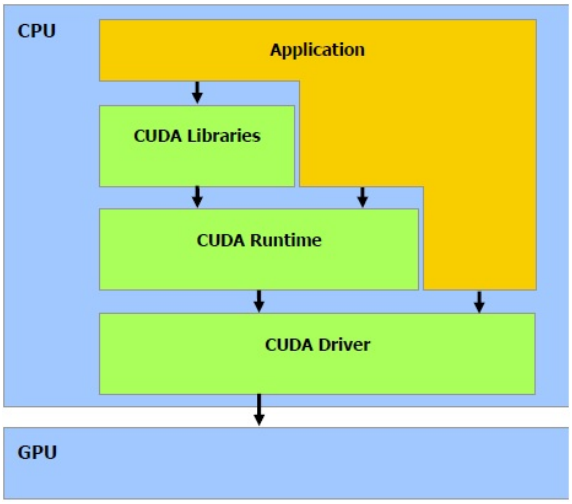


Figure 2.4: CUDA's software stack, taken from [NVI]

1. **Two high-level mathematical libraries** – CUFFT and CUBLAS.
2. **API and its runtime** – The API comprises an extension to the C programming language.
3. **Hardware driver** – Being designed by NVIDIA, CUDA relies on hardware specific drivers, created for their own graphics cards.

Programming Model

In CUDA, the GPU is viewed as a data-parallel co-processor, capable of executing a huge number of threads in parallel. To get the desired effect, a function that operates on each of the data independent partitions is compiled to the instruction set of the device. At runtime the resulting program, a kernel, is downloaded to the device and mapped onto one or more threads. Both the host (CPU) and the device (GPU) maintain their own memory, allowing the programmer to issue data transfers between them as needed.

To organize the different threads running the same kernel in the device, CUDA joins together batches of threads, as thread blocks, allowing them to cooperate and synchronize by sharing data through fast shared memory. Each thread is assigned a thread identifier, which is the thread's number within its block.

There is a limited maximum number of threads that a block can contain. Nevertheless, blocks of equal dimensionality and size that execute the same kernel can be grouped together into what is known as a grid of blocks. This is done at the expense of thread cooperation, because threads in different blocks inside a grid cannot communicate or synchronize. Yet, this mechanism allows devices to run all the blocks of a grid sequentially, in parallel, or even a combination of both, boasting the parallel factor of the device. The block's identifier is its number within the grid.

CUDA's memory management coordinates the memory access per-thread, per-block and per-grid, dividing the memory between grids, and structuring the memory spaces as illustrated in Figure 2.5. In this memory coordination scheme, the host has full access to the global, constant, and texture memories.

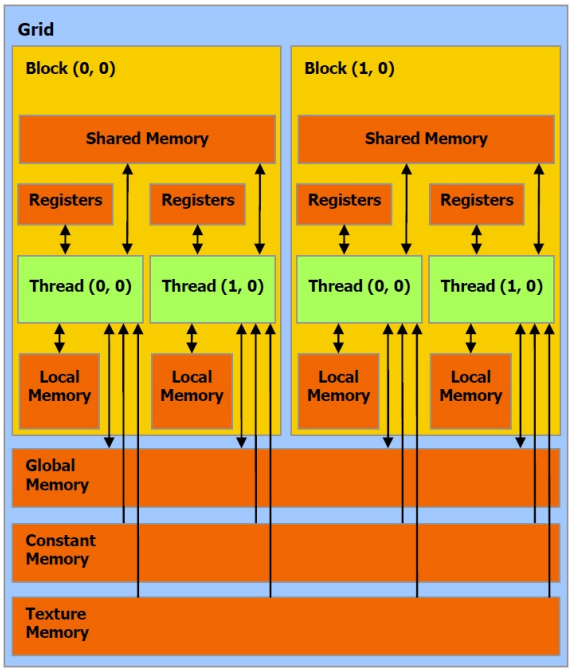


Figure 2.5: CUDA's memory management, taken from [NVI]

- **Registers** – Read-write per-threads.
- **Local memory** – Read-write per-threads.
- **Shared memory** – Read-write per-block.
- **Global memory** – Read-write per-grid.
- **Constant memory** – Read-only per-grid.
- **Texture memory** – Read-only per-grid.

2.2.3 Open Computing Language

As the GPUs evolved into programmable parallel processors, it became important to provide software developers the means to easily use these powerful processing platforms. Furthermore, the need for software developers to take full advantage of heterogeneous processing platforms (e.g., servers, desktop computers, hand-held devices) has grown.

This lead to the proposal of some platforms on which software developers could build applications that used both CPUs and GPUs as main computational devices, giving developers a simple yet efficient way to control and use them. However, these platforms were usually associated to a specific vendor or hardware and were not standard from an industry perspective. As a result, they hampered the development of applications that harnessed the computational power of these processors, from a single or multi-platform source code base.

With these prospects in mind, a consortium of corporations, known as Khronos, joined efforts to create an industrial open standard language for general-purpose parallel programming, across multiple hardware platforms. The result is a language known as OpenCL [Mun+09].

> "OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms." [Mun+09]

Being launched in 2008, it supports a wide range of applications, from consumer software, to HPC solutions. It is particularly suited to play a key role in emerging interactive

graphics applications that combine graphics rendering pipelines with general parallel compute algorithms.

Structurally, OpenCL is a cross-platform programming language, featuring an API that enables the coordination of parallel computations across multiple heterogeneous processing devices. However, it is more than just a language. It is a framework on which software developers can build general-purpose programs that execute on GPUs without the need to map their algorithms into a 3D graphics API such as OpenGL [NDW97] or DirectX [BD98]. Its architectural details will be subsequently outlined.

Platform Model

OpenCL's platform model if fairly straightforward. Nevertheless, first we must define a common terminology that is going to be used throughout the rest of the thesis.

- **Host** – Formally, the host is the entity that uses the OpenCL API. Usually, it is a processing unit such as a CPU.
- **OpenCL device** – An OpenCL device is a collection of compute units, that execute commands issued by the host. Typically, these devices correspond to GPUs, multi-core CPUs, or other forms of parallel architectures.
- **Compute Units** – A compute unit is composed of one or more processing elements. In most modern GPUs, for example, a correspondence to a compute unit is a stream multiprocessor, composed of a very large number of small processing units.
- **Processing Elements** – It is a virtual scalar processor. It is the basic computational structure, on which the commands issued by the host are computed. For example, it is and ALU (arithmetic logic unit) inside a stream multiprocessor, in a GPU.

OpenCL's platform model consists of a host connected to one or more OpenCL devices, which in turn are divided into one or more compute units. These compute units are further divided into one or more processing elements.

An OpenCL application runs on a host according to its platform model. The application submits commands from the host to the device, prompting the execution of computations on the processing elements. On the other hand, these processing elements execute a single stream of instructions as SIMD units.

Execution Model

Firstly we must expand the terminology defined previously.

- **Context** – The environment within which the kernels execute. It includes, for example, a set of devices, the memory objects accessible to those devices, and one or more command-queues associated each device.

- **Command-queue** – An object that holds commands that will be executed on a specific device, of a particular context. Commands to a command-queue are queued in-order but may be executed in-order or out-of-order.
- **Kernel** – A kernel is a function declared in an OpenCL program. It is executed by an OpenCL device.
- **Work-item** – It is an instance of a kernel, executed by one or more processing elements.
- **Work-group** – It is a collection of work-items that execute concurrently on the processing elements of a single compute unit. All the work-items inside a work-group execute the same kernel and share local memory.
- **Program Objects** – The program source and executable that implements a specific kernel.
- **Memory Objects** – A set of objects visible to both the host and the OpenCL devices.

The execution of an OpenCL program is divided into two parts: kernels that execute on OpenCL devices, and host program that executes on the host. This model is very close to CUDA's execution model, differing mostly in the terminology - CUDA has threads instead of work-items, and thread blocks instead of work-groups.

When the host submits a kernel for the device to execute an index space is defined. This index space is called NDRange and it is a N-dimensional index space, where N is either one, two, or three. It is defined by an integer array of length N, that specifies the length of each dimension. Each point of this index space is called a work-item, and each work-item is assigned a global identifier that constitutes a N-dimensional tuple, whose components range from zero to the number of elements in that dimension minus one, that corresponds to its position within the index.

In order to provide a more coarse-grained decomposition of the index space, work-items are joined together to form work-groups. To define the number of work-groups in each dimension, an array of length N is created. Each work-group is then assigned a unique global identifier with the same dimensionality as the work-items index space. Additionally, work-items are assigned a local identifier in order to uniquely identify them within a work-group. Again, we have similarities to CUDA's execution model since a NDRange index space is analogous to a grid of thread blocks.

To exemplify the manner in which a work-item can be identified within the index, lets consider Figure 2.6. The host inputs the index space for the work-items (G_x, G_y) and the number of work-items per work-group (S_x, S_y), in each dimensions. This defines a global G_x by G_y index space where the number of work-items is $G_x \times G_y$, and a work-group S_x by S_y local index space where the number of work-items per work-group can be given by $S_x \times S_y$. Hence, the number of work-groups can be given by $(G_x/S_x) \times (G_y/S_y)$.

A work-item global identifier (g_x, g_y) is obtained directly form its position within the index space, or by combining its local identifier (s_x, s_y) and work-group identifier (w_x, y_y) as: $(g_x, g_y) = (w_x \times S_x + s_x, w_y \times S_y + s_y)$

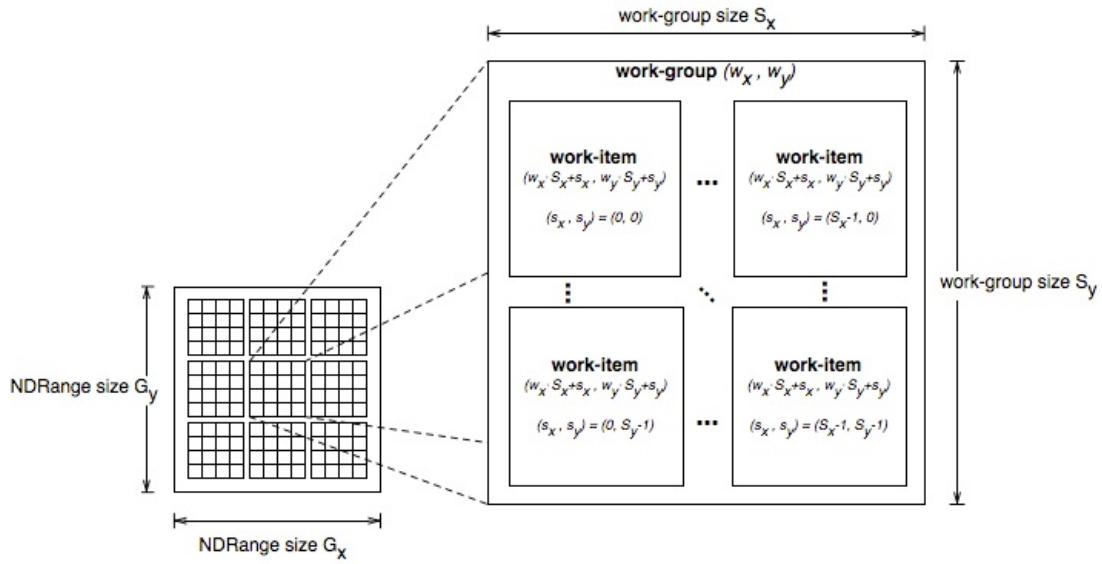


Figure 2.6: An OpenCL 2-dimensional index space, taken from [Mun+09]

The OpenCL execution model explicitly supports both data-parallel and task parallel programming models. To do so, a host program must define a context for the execution of the kernel and its resources (e.g., devices, command-queues, kernels, program objects, memory objects). This context is manipulated by the host through the use of functions from the OpenCL API. The host then places commands into a data structure called a command-queue, which are then scheduled onto the respective device.

Memory Model

In OpenCL's architecture the memory is divided into four distinct regions:

- **Global Memory** – This memory region allows read/write access to all the work-items in the index space, as well as to the host.
- **Constant Memory** – This region allows read/write access to the host, but only read access to the work-items.
- **Local Memory** – A memory region local to work-group, therefore it can be used to allocate memory objects shared by all the work-items of that specific group. The host has no access to it.
- **Private Memory** – A region of memory private to a work-item. Logically, variables stored in one work-item's private memory are not visible to another work-item. The host has no access to it.

This memory scheme is parallel to CUDA's, except in the absence of texture memory. The host's memory is usually independent from the OpenCL device's memory, however, they often need to interact. This interaction occurs either by direct copy between memories, or by mapping and unmapping regions of a memory object. To map objects to the device, the host's program uses the OpenCL API to create memory objects in global memory, and also to enqueue memory commands in order to operate on these objects. In similar fashion, to copy data explicitly the host enqueues commands to transfer data between memories.

When it comes to memory consistency, OpenCL uses a relaxed consistency model. So, the state of memory visible to a work-item may not be consistent across a collection of work-items at a given point. Only private memory has load/store consistency, the local memory is only consistent across work-items in a single work-group at a work-group barrier (a barrier is a standard synchronization mechanism in parallel computing). This is also valid for global memory, but there are no guarantees of memory consistency between work-items in different work-groups.

Programming Model

To better understand the fundamentals of the OpenCL language, a simple OpenCL program, that illustrates its basic functionalities and properties, is depicted in Listing 2.1. This example presents a program that multiplies two square matrices, A and B, and stores the results in matrix C. Nonetheless, to simplify this exemplification, the code only contemplates the essential parts related to OpenCL. Therefore, all the error checks, possible optimizations, and auxiliary functions are left out.

Before the kernel is executed by the OpenCL device some aspects relative to the execution environment must be configured, by following some common steps. First the devices that are able to run the kernel must be obtained (line 19). The type of the desired device can be selected, provided it exists in the given platform. A context must be created and associated with the devices obtained previously (line 21). It is possible to choose and use more than one device, although, in this example only one is used. Nonetheless, the need to submit orders to the device still remains. This can be accomplished by creating a *command-queue*, associated to the device and the context (line 22).

Now that the configuration of the platform is complete, the memory objects that the kernel can access must be initialized, along with their respective access permissions (lines 23 to 28). In this example, the memory objects are initialized and, at the same time, filled with the values from the matrices A and B.

Subsequently, the kernel function source (Listing 2.2) must be read, compiled, and used to create a kernel object (lines 29 to 31). There are several ways to obtain the source, so the procedure used in this example is not important. Now that kernel has been created it must bet pointed to needed arguments, for example the matrices (lines 33 to 37).

A NDRange index space is defined in order for OpenCL to map the kernel onto the

work-items. This initialization requires the number of dimensions, of work-items per dimension, and optionally, of work-items per work-group (line 38). Then, the OpenCL device is ordered to run the kernel (line 39).

Lastly, the results are read, and the resources are freed (lines 42 to 50). Note that this example is not at all optimized, since our focus in this description is readability and ease of understanding.

Listing 2.1: OpenCL code that multiplies two square matrices

```
1  int main(int argc, char** argv) {
2    int matrix_width = atoi(argv[1]);
3    int matrix_size = matrix_width*matrix_width;
4    int mem_size = sizeof(int)*matrix_size;
5    int *A = (int*) malloc(mem_size);
6    int *B = (int*) malloc(mem_size);
7    int *C = (int*) malloc(mem_size);
8    randomInit(A, matrix_size); randomInit(B, matrix_size);
9    cl_platform_id platform_id;
10   cl_device_id device_id;
11   cl_context clContext;
12   cl_command_queue clCommandQueue;
13   cl_program clProgram;
14   cl_kernel clKernel;
15   cl_int errcode;
16   cl_mem cl_A, cl_B, cl_C;
17   clGetPlatformIDs(1, &platform_id, NULL);
18   int is_gpu = atoi(argv[2]);
19   errcode = clGetDeviceIDs(platform_id, is_gpu ? CL_DEVICE_TYPE_GPU :
20     CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
21   clContext = clCreateContext(0, 1, &device_id, NULL, NULL, &errcode);
22   clCommandQueue = clCreateCommandQueue(clContext, device_id, 0, &errcode);
23   cl_C = clCreateBuffer(clContext, CL_MEM_READ_WRITE, mem_size,
24     NULL, &errcode);
25   cl_A = clCreateBuffer(clContext, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
26     mem_size, A, &errcode);
27   cl_B = clCreateBuffer(clContext, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR,
28     mem_size, B, &errcode);
29   clProgram = getKernelSource(argv[3],clContext);
30   errcode = clBuildProgram(clProgram, 0, NULL, NULL, NULL, NULL);
31   clKernel = clCreateKernel(clProgram,"matrixMul", &errcode);
32   int cl_matrix_width = matrix_width;
33   errcode = clSetKernelArg(clKernel, 0, sizeof(cl_mem), (void *)&cl_C);
34   errcode |= clSetKernelArg(clKernel, 1, sizeof(cl_mem), (void *)&cl_A);
35   errcode |= clSetKernelArg(clKernel, 2, sizeof(cl_mem), (void *)&cl_B);
36   errcode |= clSetKernelArg(clKernel, 3, sizeof(int),
37     (void *)&cl_matrix_width);
38   size_t globalWorkSize[2] = {matrix_width,matrix_width};
39   errcode = clEnqueueNDRangeKernel(clCommandQueue, clKernel, 2,
40     NULL, globalWorkSize, NULL, 0, NULL, NULL);
41   clFinish(clCommandQueue);
```

```
42    errcode = clEnqueueReadBuffer(clCommandQueue, cl_C, CL_TRUE, 0,
43      mem_size, C, 0, NULL, NULL);
44    free(A);free(B);free(C);
45    clReleaseMemObject(cl_A); clReleaseMemObject(cl_B);
46    clReleaseMemObject(cl_C);
47    clReleaseContext(clContext);
48    clReleaseKernel(clKernel);
49    clReleaseProgram(clProgram);
50    clReleaseCommandQueue(clCommandQueue);
51    return 0;
52  }
```

Listing 2.2: The kernel function

```
1  __kernel void matrixMul(
2    __global int* C,__global int* A,__global int* B, int width){
3     int px = get_global_id(0);//x-dimension
4     int py = get_global_id(1);//y-dimension
5     int value = 0;
6     for (int i = 0; i < width; i++) {
7      int elementA = A[py * width + i];
8      int elementB = B[i * width + px];
9      value += elementA * elementB;
10    }
11    C[py * width + px] = value;
12 }
```

OpenCL features two domains of synchronization, not used in the example: work-items in a single work-group, and commands enqueued to command-queue(s) in a single context. Synchronization between work-items of a single work-group is done by using a work-group barrier. Every work-item, in a work-group, must execute the barrier before any are allowed to continue subsequent computations. Still, there is no mechanism for synchronization between work-groups.

In regard to command queues there are two synchronization points: command-queue barriers, and events. The first ensures that all previously queued commands have been completed and any resulting memory updates are visible to subsequent commands. The latter, are returned by every API function that enqueues commands. A next command waiting on that event certainly views consistent memory objects.

2.3 High Level GPGPU

GPGPU APIs, like OpenCL and CUDA, are popular among software developers that intend to make use of the GPU's parallel computing qualities. Regardless, these APIs do not provide a good abstraction from the underlying execution model since, for instance,

they require an explicit management and interconnection of both host and device memories by the programmer, as well as other low-level responsibilities. It comes without surprise that efforts were made towards equipping software developers with higher-level GPGPU development tools, mostly driven towards an object-oriented paradigm. These tools provides an additional abstraction layer to the underline platform, letting developers focus on the data-parallel structures and operations, rather than on how to translate the computations to a lower level syntax.

Subsequently are presented the high-level GPGPU platforms and APIs most relevant to our work.

2.3.1 Libraries

RapidMind

RapidMind [McC06] was a C++ platform, inspired in SH [MQP02], for expressing data-parallel computations on GPUs. It was released in 2006, being the real first high-level GPGPU API.

It provided three main C++ types: Value<N,T>, Array<D,T> and Program. All the three types are containers: the first two for data and the last for operations. Parallel computations are invoked by applying either programs or parallel collective operations (reductions) to arrays, which generate new arrays or scalar values, respectively.

The Value<N,T> type is a N-tuple, holding N values of type T. T can be a basic numerical type (e.g., single/double-precision floating point, signed/unsigned integers).

The Array<D,T> type is a data container like Value<N,T>, although, it is multidimensional and variable in size. Thus, D is the dimensionality (can take the value of one, two or three), and the type T gives the type of the element, similarly to T of Value<N,T>.

Lastly, the Program type stores a sequence of operations on specific structures. Its declaration starts by the BEGIN macro and ends with the END macro, encapsulating variable declaration, and operations. The variables declared inside a Program are declared either as input or output parameters, using the IN<T> and OUT<T> template wrappers.

RapidMind was acquired by Intel in 2009, being used in the development of the Intel ArBB platform [NSLMGTWDCW+11]. At the time of writing of this document, Intel's ArBB did not support GPU executions, however, according to Intel, efforts were being made in that direction.

Aparapi

Aparapi [AMD11] is AMD's solution for high-level GPGPU. It was presented in June 2011, as an API for expressing data-parallel workloads in Java, that may execute either on OpenCL or on the Java Thread Pool. The developer creates an application where the data-parallel executions are implemented inside an extension of a Kernel base class. Upon the invocation of the Kernel's execute method the runtime decides on which platform should the the operations be executed.

If its the first execution of the kernel, Aparapi tries to convert the Java Bytecode of the kernel function into OpenCL. If the platform supports OpenCL, and the Bytecode is convertible, the conversion is carried out and the kernel executes on an OpenCL device. Otherwise, the kernel executes on a Java Thread Pool. This conversion is only done once, since the consequent executions will skip this process and go immediately to execution on the available back-end, either OpenCL or Java Thread Pool depending on the previous conclusions.

The Aparapi source code is part of an open source project and it is currently only capable of executing kernels on recent AMD GPUs. Therefore is is still bound to a specific set of hardware devices.

Accelerator and PeakStream

Accelerator [TPO06] is a C++ library with a managed API wrapper, created by Microsoft and currently in its second version, for implementing array-processing operations on multi-threaded systems, such as multi-core processors or GPUs.

It features a parallel array (PA) namespace that offers various API functions, that carry out operations (e.g., subtractions, divisions) in, and in between, arrays in parallel. These parallel executions are transparent to the programmer, and carried out on a CPU or a GPU. Nonetheless, the types of the arrays received as argument in the API functions have to be associated to the Accelerator runtime (e.g., IntParallelArray, FloatParallelArray).

To select the hardware on which the operations are executed, an appropriate target must first be created. For example, a DirectX 9 or a multi-core target. From this point on, array operations can be submitted for evaluation. These are performed in the declared order, and execute on the respective target.

In order to execute on a GPU, Accelerator automatically translates the structures and operations issued by the programmer into GPU machine code, by using the DirectX 9 API. Hence, GPUs that do not support the DirectX 9 API are unable to execute Accelerator workloads.

On the other hand, if the developer issues a multi-core target, Accelerator takes care of thread creation and synchronization. The number of threads is automatically determined, depending on the available CPUs and workload size. One limitation is that only CPUs that have an 64 bit instruction set are compatible to Accelerator.

PeakStream [Pap07] is somewhat similar to the Accelerator platform, as it also provided a C++ array-oriented data-parallelism abstraction. It uses C++ operator overloading, instead of API functions, in order to execute operations between arrays. It was bought by Google in 2007, and is currently only for internal usage.

Critical Analysis

The semantics gap between these high-level APIs and the underlying GPU programming model is considerable. Thus, they offer a decent abstraction when developing applications that are to be computed, either totally or partially, on the GPU. However, these platforms are not free of issues. In fact, they have a varied set of limitations, in regards to distinct concepts. These include, portability concerns, an over simplified execution model, and others problems.

Aparapi's approach has the advantage of providing a high degree of abstraction, although the developer is oblivious of the code's sustainability for GPU execution until it is just-in-time compiled to OpenCL, at runtime. Its programming model does not provide the same capabilities of OpenCL's, since programmers are reduced to using unidimensional arrays in kernel executions, besides normal singleton values (e.g., integers, floats). In addition, at the time of writing, it only supported GPU executions in AMD chips, analogously to CUDA. Therefore, this is a convenient tool to leverage from the existence of a GPU, when the required conditions are met, but not for actual GPGPU development.

More often than not, the scope of the application context is narrowed to a set of operations known to be supported by the target programming model. For instance, Accelerator, is a good tool for applying a sequence of element-wise arithmetic operations to a set of arrays, but is not powerful enough for general-purpose computing. On top of that, it has problems related to portability, specifically, it may only execute on GPUs that support, at minimum, DirectX 9.

2.3.2 Programming Language Support

GPGPU Programming Languages

Lime is a Java-compatible language, proposed in 2012, that exports a high-level GPU programming model. It provides a high-level object-oriented paradigm, that offers task, data, and pipeline parallelism. It extends Java with several constructs designed for programming heterogeneous architectures with GPU accelerators.

Lime represents the main computation as a task graph data structure, in which values flow between tasks over edges in the graph. To compose a task graph, the programmer uses Lime-specific operators, such as the `task` operator, or the `=>` (connect) operator. The former creates a computational unit equivalent to a OpenCL kernel, while the latter represents the flow of data between tasks. Another operator is the `finish`, that initiates the computations and forces completion.

A Lime `task` repeatedly applies a *worker* method, as long as the input data is presented to the task through an input port, and enqueues its output (the result of a method application) to an output stream. These methods are task agnostic, i.e. they may be invoked as conventional static or instance methods, only becoming worker methods by applying the task operator. Moreover, tasks are either *isolated* or *non-isolated*. An isolated task (a filter)

has its own address space and may not access mutable global state. Consequently, the *worker* method of an isolated task inputs immutable (value types) arguments, and must return values.

A value type, represents a deeply immutable object (e.g., array, or a data structure) declared using the value modifier on a type. As a result, a float bi-dimensional value array (matrix) would be declared as float[[] [N]], where $N > 0$. This would instantiate a matrix in which the outer dimension is unbounded, and the inner dimension is bounded to size N.

The => operator is used to connect two tasks, when the output type of the upstream task matches the input type of the downstream task. This is explicitly exposed in order to enable the compiler, and runtime, to automatically optimize the I/O, as well as synchronization, between tasks.

Lime offers a *map* and *reduce* model for fine-grained data parallelism. The map operator is represented by the @ token. It applies a worker function to each element of a value type, and returns the respective output (depending on the input value type). In its turn, the reduction is expressed using an operator, or method, followed by !. This indicates to the compiler that the method should be treated as a combinator. Lime permits instance or static methods to serve as reduction operators, as long as they apply the computations to two arguments of the same type, and consequently produce a result of that type.

The Lime compiler performs several optimizations to the applications, such as in regards to kernels, or even to data memory placement. However, it does not optimize how the communication is overlapped with the computation.

X10 [CGSDKEPSS05] is an instantiation of the APGAS [SABCCGKPT10] programming model on top of a base sequential language, with Java-style productivity. X10 was developed by IBM in the last few years. The design of the APGAS model was aimed at programming for clusters of multi-core nodes. X10's instantiation of the previous model, according to IBM, can be used to write efficient code for some heterogeneous parallel architectures (e.g., multi-cores, SMPs, Cell-accelerated nodes). To achieve GPU execution X10 has to support GPU programming idioms such as threads, blocks, barriers, constant memory, etc [CBS11]. Is accomplishes these challenges with an extension to the X10-to-C++ compiler, that recognizes such idioms and generates CUDA kernel code.

Before we can understand the mechanisms by which X10 allows the development of GPU executable code we must introduce some key concepts of the APGAS model. Firstly, APGAS is organized into four simple and independent concepts: *locality*, *asynchrony*, *conditional atomicity* and *order*.

- **Places** – A place is an computational entity on which executions take place (e.g., x86 core, SMP, GPU). Places can be joined together and are reified, i.e. can be stored in variables, passed into functions, etc. In this model, a unit of serial execution

(thread) can be denominated as an *activity*, and an activity is only located in a specific place for its lifetime. Given a place p the statement at(p) S can be used to request the execution of S at place p.

- **Asynchrony** – An activity may utilize the statement async S to launch a new activity to execute S. S may reference variables in the surrounding lexical environment.

- **Conditional Atomicity** – An activity may use the statement when (c) S, where c is a boolean valued expression. The execution of this statement in a state s terminates in a single step, and yields the state s' if and only if the condition c is true in s, and that the execution of S in s yields s'.

- **Order** – The statement finish S imposes partial order on the state changes provoked by individual activities. After S is executed it waits for all activities spawned during the execution of S to terminate. Thus, ensuring that subsequent activities see the effect of the state changes that resulted from the execution of finish.

The async, finish and at statements can be nested arbitrarily, providing an increased flexibility to the APGAS model.

Using APGAS' constructs a programmer is able to create applications that use one or more GPUs as their execution platform. Though APGAS offers a higher abstraction of the GPU programming model than low-level GPGPU APIs, its execution model is not unlike CUDA's or OpenCL's. Since X10 compiles CUDA code, we will use CUDA's terminology for the rest of the description. In that sense, it is natural that APGAS' constructs are organized in such way that clearly separates memory domains. For example, global memory is represented as heap memory for a place, since it outlives the execution of a kernel. Local and private memories are finer-grain memory locations, associated to either a block, or a thread, respectively. Given that the latter two do not outlive kernel execution, in APGAS they are described as stack memory. This memory hierarchy is more easily unified with the notion that an individual GPU is a single place, rather than viewed as several interconnected places.

In X10, remote memory allocation is accomplished with the new construct, analogous to Java. However, the GPU programming model did not allow memory allocation inside a kernel until very late in the development process of X10, so this functionality was left out. Rather, the host part of the application calls API functions (in similar fashion to CUDA or OpenCL) to allocate desired memory spaces. An example of this mechanism is the following primitive: `CUDAUtilities.makeRemoteArray[T](p, sz, single_value);`. This creates an array of type T in GPU at place P, with its corresponding size and values.

To copy data to/from the GPU X10 provides an API that mirrors Java's System.arrayCopy, although being asynchronous, and allowing one of the arrays to be a remote reference. The following example depicts the copying of data between arrays r and l, remote and local respectively: `finish Array.asyncCopy(r, r_off, l, l_off, len);`.

The APGAS model includes built-in synchronization constructs that impose executional order at a block level. These constructs are analogous to synchronization barriers common in parallel computing, but naturally applied to GPU programming. These barriers are useful when a kernel uses local shared memory for its blocks, letting threads inside a block synchronize at will. In APGAS, the block level synchronization construct is called clock. In turn, GPU kernel parameters are represented in APGAS as the capturing of local variables within the at construct. Finally, when using pointers APGAS creates an object graph on the GPU.

Compiler Directives

OpenACC [Ope11] is a directive specification for GPU programming, recently proposed by a consortium that includes the Portland Group. It is, thus, with no surprise that it grows from the work, on this same topic, previously developed on the PGI compiler. The directives allow for the identification of blocks of code as potential GPU kernels. By using the #pragma acc kernels ... !$acc end kernels directive the programmer can define operations that are to be executed on the auxiliary accelerator, or GPU. This directive can be extended with clauses to specify a particular execution behaviour, for instance the loop clause naturally specifies a loop of operations.

The programmer can also define what data must be transferred to, and from, GPU global, and local, memory. The #pragma acc data copy(...) prompts the program to pre-emptively copy the desired data to device memory, before it is actually processed. Moreover, the programmer can delimit data regions. These regions represent a persistent memory zone, where the data is transferred to device memory as the execution enters it, and is transferred back to host memory just before exiting the zone. This allows for a systematic application of kernels, distinct or otherwise, without having to perform data transfers between them. To declare such a persistent memory zone, the developer should use the !$acc data ... !$acc end data directive.

Another interesting functionality is the reduction. The reduction clause specifies a reduction operator and one or more scalar variables. For each variable, a private copy is created for each parallel gang, or thread-group, and initialized for that operator. At the end of the region, the values for each gang are combined using the reduction operator, and the result combined with the value of the original variable. Also, the result is stored in the original variable, and becomes available after the region.

Compiler and Runtime Systems Support for GPGPU

StreamIt [TKA02; UGT09] programming language, originally proposed in 2001, aims to express streaming applications in a platform independent fashion by exposing task, data, and pipeline parallelism in a natural way. This approach is advantageous since an optimizing compiler for the target platform can exploit parallelism on the latter in the most efficient manner, without significant effort by the programmer.

In StreamIt, the most basic unit of computation is a Filter. Within a filter resides the user-defined computational steps, in some way analogous to a coarse grained OpenCL kernel. Each filter holds two communication channel, input and output, used to communicate with its neighbour filters. These channels are typed FIFO queues that support three basic queue operations: pop, peak, and push. Additionally, each filter has production and consumption rates associated to its channels.

Filters can be combined into a data communication low-level graph, denominated a Stream construct. Individually these constructs do not provide an execution behaviour, instead they organize filters into a sequential composition of executions, that is, an execution pipeline.

Beside Streams (pipelines) there are two other stream constructs: SplitJoin, and FeedbackLoop. The former is used to specify independant parallel streams that diverge from a common *splitter* and merge into a common *joiner*. The latter, provides a way to create cycles in the stream graph. Accordingly, a StreamIt program is expressed as a hierarchical composition of the three previous simple stream structures, forming a data flow or stream flow graph.

The compilation process in StreamIt is composed of a series of stages, namely: *profiling and execution configuration selection*, *software pipelined scheduling*, and *final code generation*. Usually the optimal execution configuration varies depending on the application and the GPU itself. Therefore, to achieve a close to or optimal execution configuration for the corresponding application, on the target platform, StreamIt profiles the source program using several distinct configuration settings in regards to number of threads and number of registers per thread. The profiling code is generated using NVIDIA's NVCC compiler, creating four versions of the profile executable per filter. In the end, the best execution configuration, performance-wise, is chosen.

In the *software pipelined scheduling* the compiler generates constrains on the scheduling and synchronization of filters, based on the application's data flow or stream flow graph. These constrains assert that the consumption and production of data between stream constructs are accomplished correctly, and efficiently. Furthermore, given the particular characteristics of GPU memory (e.g., latency, bandwidth), the set of constrains include some that take into account the necessity for coalesced memory accesses. To obtain the highest possible memory bandwidth StreamIt has its own buffer layout scheme.

After all the constrains are generated, they are necessarily solved by the compiler. Subsequently, in the *final code generation* stage, CUDA runtime code is generated, and the application becomes ready to execute on a CUDA device.

Java – Rootbeer [PSFW12] is a platform that offers GPU executions, via the Java language. It presents itself as an alternative to Java bindings to GPGPU languages. Its execution model is fairly straightforward. The developer simply writes Java code, leaving all the details associated to the execution to the platform.

The kernel functions are represented by a Kernel interface, not unlike Aparapi's approach. The programmer must declare a class that implements the interface, and program the void gpuMethod(); function. The latter is the main point of entry of the developers code on the GPU. The class' private fields express the data that is to be transferred from/to GPU memory. Rootbeer automatically finds all reachable fields and objects from the gpuMethod and copies these to the GPU. Equivalently, on the GPU results are saved to fields and Rootbeer copies them back to the CPU.

To transform a regular Java program to a GPU accelerated one, the developer runs the command $java -jar Rootbeer.jar InputJar.jar OutputJar.jar, where InpurJar.jar and OutputJar.jar, are the original Java application jar and the GPU accelerated application jar, respectively. The original Java Bytecode suffers several successive trasnformations, before the final jar is created. These processes are particular to Rootbeer's implementation and, therefore, will be omitted. Regardless, CUDA is used as part of the transformation process.

Rootbeer supports all features of the Java language, with the exceptions of: dynamic method invocation, reflection, and native methods. The main structures of a typical Java program (e.g., objects, variables, methods) are serialized and appropriately expressed in GPU memory. Rootbeer automatically manages the entire available GPU memory, in order to not only optimize the executions, but also, allow dynamic memory allocation. Consequently, it is possible to allocate new objects on the GPU, while executing a kernel.

Critical Analysis

Of the studied frameworks, StreamIt is the one with features closer to our goals. Its a language that enables the creation of disciplined graphs of filters. All orchestration and kernel code is generated by the compiler, and subjected to a subsequent profiling phase. Nevertheless, is is not clear which are the restrictions imposed on filter implementation. Comparing it to our goals, StreamIt has a narrower scope, given that it its a streaming language that is limited by the algorithms that can be effectively implemented in such a paradigm. In turn, Marrow is a library that can be used to offload computations to GPUs, such as an image filter, or a matrix operation, regardless of the application's remaining purposes.

Lime provides constructs for the composition of task graphs, in a way similar to StreamIt. Data transfers between the Lime runtime and the GPU devices are very expensive, due to the serialization step required to cross the Java-C frontier. This penalty is only overcome by the JVM compatibility of the Lime byte code, since the remainder features have been mostly explored in other languages.

Regarding X10, the programming model is very imperative, forcing the developer to be also aware of the underlying execution model. In order to be suitable for GPU execution, a X10 asynchronous task must begin with two for loops: one for the distribution of

the work per blocks, and a second for the distribution of the work within a block. Moreover, the enclosed code must fulfil several restrictions, some not very understandable, such as the absence of method invocations.

In its turn, Rootbeer's memory management technique reduces the serialization overhead for a single kernel offload, found for instance in Lime. Yet, this is done at the expense of allocating all of the GPUs memory for each kernel execution. This approach mines the efficient composition of GPU computations that builds on the persistence of data across multiple kernel execution, for instance a pipelined kernel execution.

The compiler directives for GPUs solution, offered by OpenACC is an interesting proposal. It enables the seamless offloading of data-parallel executions, or kernels, to a GPGPU environment. Additionally, it enables fine tuning of kernel interoperability, via defining persistent memory zones. Still, compiler directives are a solution to a very specific niche of computations. It enables the programmer to efficiently build data-parallel schemes, but does not delve into the task-parallel domain.

2.4 Algorithmic Patterns and GPGPU

According to the Oxford Dictionary[1] a pattern is:

> "A regular and intelligible form or sequence discernible in the way in which something happens or is done"

In turn, when it comes to computer science, patterns are usually associated to two major schools: architectural patterns [BMRSS96] (or architectural styles [SG96]), and design patterns [Gam95]. The first define types of elements and relationships that together form the backbone of a software system, abstracting the developers from low level design problems and easing the creation of complex applications. The latter are general and focussed solutions to commonly recurring problems within a given context, contrary to architectural patterns that have a larger design scope.

In computer science, many new challenges are just specializations or subsets of problems that have already been resolved, or can be tackled with the solutions of similar problems. Bearing this in consideration, patterns are particularly useful because they encapsulate a proven and tested solution to a recurring problem, facilitating the job of current software developers.

Design patterns, in particular, are an important line of investigation of today's research, and one of the most important contribution to that work is [Gam95]. The author identifies three classes of design patterns:

- **Creational Patterns** – These abstract the instantiation process, helping to make the system independent of how its objects are created. For example, the Abstract Factory pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

[1] http://oxforddictionaries.com

- **Structural Patterns** – These are concerned with how classes and objects can be composed to form larger structures. They use inheritance to compose interfaces and/or implementations, resulting in a more flexible design. For example, the Proxy pattern provides a surrogate, or placeholder, for another object, thus controlling external accesses to the latter.

- **Behavioural Patterns** – These are concerned with algorithms and the attribution of responsibilities among the objects. Additionally, these describe the flow of communication between objects. An example of this kind of pattern is the Iterator pattern, that provides a way to access the elements of an aggregate object sequentially, without exposing its underlying representation.

2.4.1 Overview of Patterns for Parallel Computing

Many other patterns have been identified by other authors. In the particular context of parallel computing, patterns are used to facilitate the development of applications that gain performance through the parallel execution of some or all of its components. They are very relevant to the development of parallel applications since it is not always easy to understand where the execution could be parallelized, and where to use synchronization mechanisms. Even worse, debugging parallel applications is all but trivial, which can cause some bugs, like race conditions, to pass without checking into the final application. The more one can abstract developers from parallel programming details, the easier it will be for them to create parallel applications.

It has been found that fully abstracting the developer from all the parallel programming issues, like in compiler directives (e.g., OpenMP [DM98]) or languages (e.g., Fortran90 [EPL94]) that generate parallel code from sequential one, is a very limiting and complex solution. An alternative is to provide tools that guide the programmer throughout the development process. Patterns are particularly useful in this context since they abstract the development process from most parallel programming concerns, yet, still providing a good level of flexibility to it.

In the context of patterns for parallel computing there are two main lines of investigation: parallel patterns and algorithmic patterns (skeletons). We are particularly interested in the latter but provide a brief overview of an important parallel patterns technology.

Parallel Patterns

Parallel patterns are an approach proposed by the Par lab group [Ber] of the University of Berkeley. They have been developing a pattern language called PALLAS [KMMS10] (Parallel Applications, Libraries, Languages, Algorithms, and Systems) that identifies and categorizes many design patterns, most of them parallel ones. These categories make up a layered hierarchy of patterns, usable by the developers to systematically address each portion of the design problem. However, developers are encouraged to bounce

around between layers as they see fit, since a top-down or bottom-up analysis is not always the most appropriate.

There are five categories of patterns identified in PALLAS, though the first two are often merged:

1. **Structural and Computational** – The first describe the overall organization of the application and how its computational elements interact. The latter describe the class of computations that make up the application. Computational patterns are responsible for defining what computations occur inside the components defined by the structural patterns.

2. **Algorithmic Strategy** – These patterns define high-level strategies to exploit concurrency in computations, for execution on parallel devices.

3. **Implementation Strategy** – These are the structures that support how the program itself is organized and what are the common data structures specific to parallel programming.

4. **Parallel Execution** – These are the approaches usually used to support the execution of a parallel algorithm.

These categories, as well as their corresponding patterns, are depicted in a hierarchy of patterns in Figure 2.7. Some of these patterns will be subsequently expatiated, therefore in order to get a detailed description over all of them the reader is remitted to [KMMS10].

Should be noted that PALLAS is a pattern language and not a programming language. It is merely a formal way to describe a common taxonomy over the fundamentals of an area of computer science, in this case parallel programming. This allows a common and universal comprehension of particular aspects of the subject, defining terms and trains of thought so as to have mutual understanding between fellow developers.

Skeletons

Algorithmic patterns, or skeletons, were introduced by Cole in [Col91] and are essentially abstractions of commonly used parallel patterns of computation, communication and interaction. The computational constructs manage the logic and control flow, while communication and interaction mechanisms control data exchange between processes, as well as their creation and synchronization. Skeletons provide a top-down design throughout the program structure, therefore, structured parallel programs are expressed by combining (nesting) parametrized skeletons, analogous to the way in which structured sequential programs are constructed.

Skeletons are usually made available to developers as an algorithmic skeleton framework (ASkF), which offers a set skeletons with generic functionalities. These skeletons

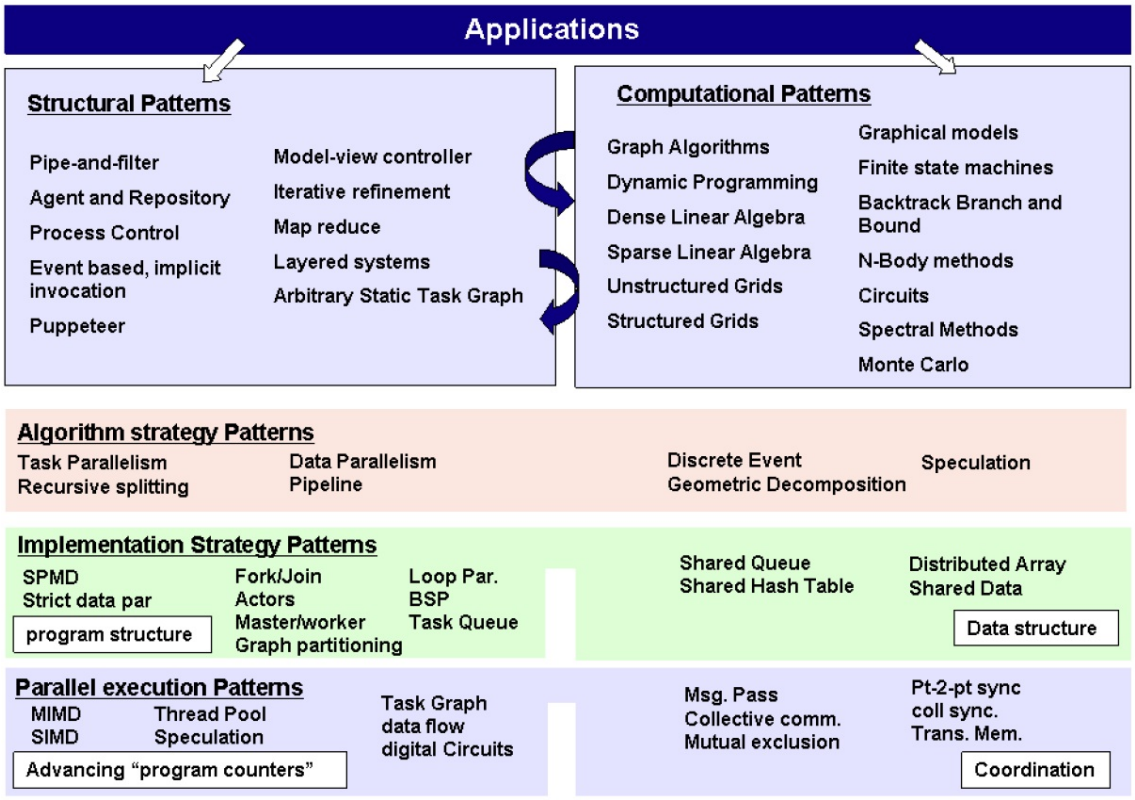


Figure 2.7: Pallas layered hierarchy of patterns, taken from [KMMS10]

can be parametrized by the programmer in order to achieve the desired parallel program. ASkFs are analogous to common software libraries, so they are accessible through well defined application programming interfaces, although they usually dictate certain aspects of the resulting program (e.g., control flow, resource monitoring, portability).

This high-level parallel programming methodology is known as *structured parallelism*, allowing an abstract description of a program that focuses on its algorithmic structure rather than its detailed implementation. This approach greatly benefits the development of complex parallel applications since the programmer has only to determine what computations are to be executed, leaving the parallel programming details up to the ASkF.

In [GVL10], skeletons are categorized based on their functionality into: data-parallel, task-parallel and resolution. Resolution skeletons delineate an algorithmic method to undertake a given family of problems. Their structure may encompass different computation, communication, and control primitives. These categories include some popular skeletons such as:

- **Data-parallel**
  - **Map** – Specifies that a function or a sub-skeleton is simultaneously applied to many independent data-sets, achieving data-parallelism. Hence, the *Map* skeleton can be regarded as SIMD parallelism.
  - **Reduce** – Is employed to compute prefix operations in a data-set, by traversing

it from left to right and then applying a function to each pair of elements. Partial results are aggregated as the reduction systematically decreases the data length, until only one element is available. That element is the final result of the *Reduce* skeleton.

- **Task-parallel**
  - **Master-worker** – Provides the ability to issue the execution of independent tasks across multiple computing nodes (workers). The *master* manages the task poll, issuing new tasks and collecting their respective results. On the other hand, *workers* take tasks from the task poll, execute them, send the results back to the *master*, and then grab the next tasks. This process continues until no other task is available.
  - **Pipeline** – Enables staged computations, where parallelism is achieved by computing different stages simultaneously on different inputs. The number of stages in the *Pipeline* may vary and the execution of the first stage is always sequential. From there on out, the execution is parallelized by feeding the output from one stage as input to the next one, in an assembly-line like manner.
- **Resolution**
  - **Divide & Conquer (D&C)** – A generalization of the *Map* skeleton, where the function is recursively applied until a condition is met within an optimization search space. Initially, a condition component is invoked on the input resulting in one of two actions: either the data is passed onto a sub-skeleton, or the input is split with the split component. Then, for each list element the same process is applied recursively until no further recursion is needed. Finally the results are combined at each level using a merge skeleton until the merge yields one result, which corresponds to the final result of the *D&C* skeleton.

Due to the heterogeneity of today's computational environments many ASkFs have been proposed in order to permit the utilization of skeletons across many platforms. For example, Calcium [CL07] is an ASkF designed specifically for clusters, while Skandium [LP10] targets shared memory architectures. The list is quite extensive, so in Table 2.1 a summarized view of some ASkFs is given.

Skandium

We use Skandium [LP10] to illustrate in more detail the functionalities provided by an ASkF that executes on a high-level language, in this case Java. Skandium is a complete reimplementation of Calcium, mainly inspired by Lithium [ADT03] and Muskel [DD06] ASkFs. It is relevant to our work since it is a relatively recent ASkF with some impact in scientific community, and whose code is available to everyone. Skandium differentiates two concepts: skeleton patterns and muscle blocks. The first are obviously the skeletons

	Programming language	Execution language	Distribution library	Type safe	Skeleton nesting	Skeleton set
Alt	Java	Java	Java RMI	Yes	No	map, zip, apply, scan, sort, reduce, replicate
ASSIST	Custom control lang.	C++	TCP/IP + ssh/scp	Yes	No	seq, parmod
Calcium	Java	Java	ProActive	Yes	Yes	seq, pipe, farm, for, while, map, d&c, fork
Eden	Haskell (extension)	C	PVM / MPI	Yes	Yes	map, d&c, pipe, iterUntil, torus, ring
eSkel	C	C	MPI	No	Yes	pipe, farm, deal, Butterfly, hallowSwap
HDC	Haskell (subset)	C	MPI	Yes	Yes	map, red, scan, filter, dca, dcB, dcD, dcE, dcF
HOC	Java	Java	Globus	No	No	farm, pipe, wavefront
JaSkel	Java	Java	RMI	No	Yes	farm, pipe, heartbeat
Lithium	Java	Java	RMI	No	Yes	pipe, map, farm, reduce
Mallba	C++	C++	NetStream MPI	Yes	No	exact, heuristic, hybrid
Muesli	C++	C++	MPI OpenMP	Yes	Limited	array, matrix, farm, pipe, parallel comp.
Muskel	Java	Java	RMI	No	Yes	farm, pipe, seq, custom
P^3L	Custom control lang.	C	MPI	Yes	Limited	map, reduce, seq, comp, pipe, farm, scan, loop
QUAFF	C++	C	MPI	Yes	Yes	seq, pipe, farm scm, pardo
SAC	Custom	C-like	Threads	No	No	genarray, modarray, fold
SCL	Custom control lang.	Fortran	*Ad hoc* Tools	Yes	Limited	map, scan, farm, fold, SPMD, iterateUntil
Skandium	Java	Java	Threads	Yes	Yes	seq, pipe, farm, for, while,map, d&c, fork
SKELib	C	C	MPI	No	No	farm, pipe
SkeTo	C++	C++	MPI	Yes	No	list, matrix, tree
SkIE	GUI / Custom control lang.	C++	MPI	Yes	Limited	farm, pipe, map reduce, loop
Skil	C (subset)	C	—	Yes	No	pardata, map, fold
SkiPPER	CAML	C	SynDex	Yes	Limited	scm, df, tf, intermem

Table 2.1: Comparative table of the algorithmic skeleton frameworks, taken from [GVL10]

available to the programmers, while the latter are sequential blocks of code that provide the logic needed by a skeleton, in order to transform it into a specific application.

The muscles can be classified into one of four types, with the following nomenclature:

1. **Execution** – $f_e : P \rightarrow R$
2. **Split** – $f_s : P \rightarrow \{R\}$
3. **Merge** – $f_m : \{P\} \rightarrow R$
4. **Condition** – $f_c : P \rightarrow boolean$

P is a parameter type, R is a result type, and $\{X\}$ represents a list of elements of type X. Muscles are viewed as black boxes, invoked during the computation of a skeleton, that may execute sequentially or in parallel, depending on the skeleton. In either case,

the result of a muscle is passed as a parameter to other muscles until no further one is needed, finally delivering the result to the user.

In order to avoid programming errors, and simplify the programming model, Skandium skeletons have the following hypotheses:

- **Single input/output** – Skeletons can only receive/produce single inputs/outputs. This hypotheses simplifies skeleton nesting and as a result low-level skeletons (e.g., *reduce*, *split*) are embedded directly into higher-level ones (e.g., *map*, *D&C*).
- **Passive Skeletons** – Each skeleton output is directly related to a previously received input. Therefore, skeletons that can produce outputs without receiving inputs, *heart-beat skeletons*, are not allowed.

Skandium supports the following skeletons: *Seq*, *Farm* (*Master-worker*), *Pipe* (*Pipeline*), *If*, *For*, *While*, *Map*, *Fork*, and *D&C*. For further information about these skeletons the user is remitted to [GVL10] and [LP10]. These skeletons allow nesting in the following manner, where Δ stands for a nested skeleton pattern:

$$\begin{aligned} \Delta \quad ::= \quad & seq(f_e) \mid farm(\Delta) \mid pipe(\Delta_1, \Delta_2) \mid while(f_c, \Delta) \mid if(f_c, \Delta_{true}, \Delta_{false}) \\ \mid \quad & for(i, \Delta) \mid map(f_s, \Delta, f_m) \mid fork(f_s, \Delta_i, f_m) \mid d\&c(f_c, f_s, \Delta, f_m) \end{aligned}$$

- **Seq** – Terminates a recursive nesting of skeletons, since it wraps execution muscles which are then nested into the skeleton program as terminal leaves of the skeleton nesting tree.
- **Farm** – Receives a set of skeletons that are replicated for task parallelism.
- **Pipe** – Can have a variable or fixed number of stages, but it is worth noting that fixed staged pipes can be nested inside other fixed staged pipes to create a pipe with any number of stages.
- **If** – Receives two sub-skeletons along with a condition muscle. Depending on the result of the condition, either one or the other sub-skeleton is executed.
- **For** – Receives a sub-skeleton and an integer i as parameters. The sub-skeleton is executed i times and the result of one invocation is passed as arguments to the next one, until the last iteration provides the final result.
- **While** – Is analogous to the *For* skeleton, but instead of iterating a fixed number of times, a condition muscle decides whether the iteration stops or continues.
- **Map** – Receives a split muscle, a sub-skeleton, and a merge muscle. The input is split with the split muscle, generating a list of elements. Then, the sub-skeleton is applied to each list element in parallel. When the results are ready, they are combined with a merge muscle and the final result is returned to the user.

- **Fork** – Behaves like *Map*. However, a different sub-skeleton is applied to each data-set.
- **D&C** – A generalization of the *Map* skeleton, where *Maps* are recursively applied while a condition is met. Its execution is analogous to standard *D&C* skeleton, as previously explained. In Skandium the *condition*, the *split*, and the *merge* components are given by muscles.

2.4.2 Skeletons Libraries for GPGPU

One of the main goals behind skeletons is portability, so it comes as no surprise that attempts of utilizing the data-parallel executional qualities of GPUs were made. It is particularly interesting when skeletons embrace GPGPU technologies such as CUDA and OpenCL, since they provide a greater detachment between implementation and underline platform. In particular, SkelCL [SKG11] and SkePU [EK10] are ASkFs that support to GPU execution through GPGPU APIs, and will be the next points of discussion. Additionally, an overview about the ASkF Muesli [EK12] is given. The latter only supports GPU execution via CUDA, therefore it is not as relevant to us as SkePU and SkelCL.

SkePU

SkePU is a C++ ASkF, released in 2010, that can be used to build parallel applications that execute not only on the CPU, but also on the GPU. SkePU supports the following five skeletons: *Map*, *Reduce*, *MapReduce*, *MapOverlap*, and *MapArray*. The latter two are variants of the *Map* skeleton.

Skeleton execution in SkePU is accomplished by running in a back-end appropriate to the execution environment, decided at compilation time by the programmer. On one hand, it supports standard sequential CPU executions and multi-core CPU execution via OpenMP. On the other hand, GPU executions can be attained by utilizing either CUDA or OpenCL. These multi-platform execution models are seamlessly available to the programmer by simply changing the application's compilations options. Skeleton code is transformed into the appropriate type of machine code, based on the execution platform chosen by the programmer.

SkePU generates executable code that depends upon user-defined functions. The definition of these user-defined functions is based on macros that are expanded by SkePU into C structs. These structs have member functions for CUDA and strings for OpenCL. When using the CUDA runtime, device and host code are separated by compiling the source code with NVIDIA's NVCC compiler. In OpenCL's case, device code is stored as strings in the program so that it can be used as arguments to API functions, that compile and upload it into OpenCL devices at runtime.

The macros define the behaviours of the skeletons and are used as arguments upon skeleton instantiation, though, not all macros are compatible with all skeletons. The programmer can utilize the macros shown in Listing 2.3.

Listing 2.3: SkePU macros, taken from [EK10]

```
1  UNARY_FUNC( name, type1, param1, func )
2  UNARY_FUNC_CONSTANT( name, type1, param1, const1, func )
3  BINARY_FUNC( name, type1, param1, param2, func )
4  BINARY_FUNC_CONSTANT( name, type1, param1, param2, const1, func )
5  TERNARY_FUNC( name, type1, param1, param2, param3, func )
6  TERNARY_FUNC_CONSTANT( name, type1, param1, param2, param3, const1, func )
7  /* Particular to the mapoverlap skeleton. */
8  OVERLAP_FUNC( name, type1, over, param1, func )
9  /* Particular to the maparray skeleton. */
10 ARRAY_FUNC( name, type1, param1, param2, func )
```

The data values that are used in the macros are contained in a special sort of structure called vector, analogous to a C array. However, it implements particular memory management techniques in order to optimize data transfers between host and device memory. If a computation changes a vector in the GPU memory, it is not directly transferred back to the host memory. Instead, transfers are suspended until an element is accessed on the host side, only then is any copying done. This process is known as *lazy copying*. Recently a matrix data structure, analogous to a vector but two-dimensional, was added to SkePU.

SkePU's skeleton have the following characteristics:

- **Map** – Receives up to three vectors, of length N, and returns a new vector, also of length N, that corresponds to the applications of function f to the corresponding elements of the input vectors.
- **Reduction** – A scalar result is computed by applying a commutative associative binary operator $\oplus$ between each element in the input vector.
- **MapReduce** – A simple combination of the previous skeletons. It proceeds in the same way as if one applied a *Map* to k vectors, $k \leq 3$, and a subsequent *reduction* to the resulting vector. As expected, this skeleton receives two macros upon its instantiation, one for the *Map* operation and another for the *Reduction*.
- **MapOverlap** – In *MapOverlap* each element r_i of the result vector is a function of several adjacent elements of element i in the input vector. The number of these values is controlled by the overlap parameter. This skeleton can only be instantiated with the OVERLAP_FUNC macro.
- **MapArray** – It receives two input vectors, v_1 and v_2, and returns an output vector where each one of its elements, r_i, is a combination of the corresponding element of the second vector, v_{2_i}, and any number of elements from the first vector, v_1. This skeleton can only be instantiated with the ARRAY_FUNC macro.

Listing 2.4 depicts a SkePU application that calculates the dot product of two vectors. The program utilizes a *MapReduce* skeleton instantiated with two macros, a multiplication macro and a sum macro, representing the *Map* and *Reduce* operations, respectively

(lines 1 to 4). The input data for the skeleton comprises two vectors of length *1000* each, whose elements are initialized with the value *2* (lines 5 and 6). The input data is then passed onto the skeleton and a scalar result is produced and printed (lines 7 and 8). The skeleton applies the multiplication operation to the vectors, element-wise, generating a vector with length *1000*, with the value *4* in its every position. The reduction the adds up every element in that vector, amounting to the value 4000 as the skeleton's final result.

Listing 2.4: SkePU map reduce example, taken from [EK10]

```
1  BINARY_FUNC( plus, double, a, b, return a+b;)
2  BINARY_FUNC( mult, double, a, b, return a*b;)
3  int main(){
4  skepu::MapReduce<mult, plus> dotProduct(new mult, new plus);
5  skepu::Vector<double> v0 (1000, 2);
6  skepu::Vector<double> v1 (1000, 2);
7  double r = dotProduct(v0, v1);
8  std::cout<<"Resutl:"<<r;
9  return 0;
10 }
11 // Output Result: 4000
```

We refrained from using the OpenCL example that has been used throughout this Chapter because of SkePU's limitations. Despite being possible to implement a matrix multiplication in SkePU, it is not accomplished in a simple way. This is mainly due to the fact that SkePU gives the programmer no control over how each thread accesses the input data, forcing each thread to work with a fixed position within the input data space. Although the ARRAY_FUNC macro lets every thread access all elements of a single vector, both arrays must be of the same type, and no additional input parameters are permitted. This does not help to simplify the process of matrix multiplication.

One possible solution is the following. Consider A, B, C as two input and one output square matrices, respectively, all of width $N > 0$. To produce a single row of values of C, Cr_i, one must combine row Ar_i of A with every column of B. This leads to N^2 threads, one for each element of C, so, thread T_{ij} would need to access row Ar_i and column Bc_j to produce value C_{ij}, where $0 < i, j \leq N$. This kind of mapping is hard to achieve in SkePU since each thread cannot access values of any position within a vector or matrix. As a result, in order to achieve the desired result, the input vector that maps A would be $A' = [Ar_{1_1}, \ldots, Ar_{1_N}, \ldots, Ar_{N_1}, \ldots, Ar_{N_N}]$, where Ar_{i_j} is j-th occurrence of row i. B' would be equivalent but, as the threads need columns instead of rows, B matrix would have to be transposed and then mapped as A', resulting in $B' = [Bc_{1_1}, \ldots, Bc_{N_1}, \ldots, Bc_{1_N}, \ldots, Bc_{N_N}]$. Logically, this approach would use a *Map* skeleton.

SkePU has multi-GPU support so as to increase the performance of its parallel applications. It utilizes the different GPUs by dividing the input vectors equally among them, and doing the calculations in parallel on them. By default it utilizes as many GPUs as it can find, although this can be parametrized.

SkelCL

SkelCL is a C++ ASkF for execution on OpenCL compatible platforms, proposed in 2011. It currently supports four basic skeleton: *map*, *reduce*, *zip*, and *scan*. It holds some similarities to SkePU like its skeleton memory management mechanisms, or its skeleton execution model.

SkelCL's execution model determines that OpenCL code is generated from the aggregation of user-defined functions and pre-implemented skeleton code, so as to be compiled by OpenCL API functions, and executed on OpenCL devices. Since OpenCL cannot pass function pointers to OpenCL devices, user-defined functions are passed as strings in SkelCL. And given that code generation is time-consuming, SkelCL only generates the OpenCL code for a specific kernel once. Subsequent kernel executions utilize pre converted code, stored in a local hard-drive.

The user-defined functions define the types of the input and output values for a kernel, as well as the operations to perform on those values. Much like SkePU, SkelCL gives the programmer very little control over what elements of the input data a thread can access, which in turn limits the capabilities of SkelCL's skeletons.

SkelCL's skeletons are parametrized with a specific data structure called Vector, internally comprised of pointers to corresponding areas of host and device memories. Vectors serve as input data for skeletons, and are generated by some skeletons as output data. A Vector is capable of storing data items of any primitive C/C++ data type (e.g., int, float), as well as user-defined data structures (structs). It has a specific memory management technique analogous to SkePU's vector, that is, a *lazy copying* behaviour.

Because SkelCL's skeletons receive and produce Vectors, skeleton nesting can be explicitly simulated through argument nesting, as shown in Listing 2.5. Although, this work-around does not scale well. SkelCL's skeleton have the following characteristics:

- **Map** – Receives a Vector as input data and applies the user-defined function to its every position in parallel. As a result, it generates an output Vector with the same size as the input one.

- **Reduce** – Uses a binary operator to combine all elements of the input Vector and returns a scalar result. SkelCL requires that the operator is an associative one (e.g., sum, subtraction), so that it can be applied to arbitrarily sized sub-ranges of the input Vector in parallel. The final result is obtained by recursively combining the intermediate results for the sub-ranges.

- **Zip** – Takes a customizing binary operator and applies it to corresponding elements of two equally sized input Vectors of size N, resulting in a Vector of the same size. *Zip* is parallelized in the same manner as *Map*.

- **Scan** – Applies a customizing binary operator $\oplus$ to the elements of an input Vector of size N, returning a Vector of the same size, and whose values are determined

in the following way: *scan* $\oplus$ v = $[id, x_0, x_0 \oplus x_1, \ldots, x_0 \oplus \ldots \oplus x_{N-2}]$, where $v = [x_0, \ldots, x_{N-1}]$ and id is the identity element of the operator $\oplus$.

In order to better understand the mechanics behind skeleton utilization in SkelCL an example is outlined below. For the same reasons as in the case of SkePU, we will refrain from using the OpenCL example that has been used throughout this Chapter. Instead, Listing 2.5 illustrates a SkelCL program that computes the dot produce of two Vectors.

Listing 2.5: SkelCL computation of the dot product of two vectors, taken from [SKG11]

```
1  int main (int argc, char const* argv[]) {
2    SkelCL::init(); /* initialize SkelCL */
3    /* create skeletons */
4    SkelCL::Reduce<float> sum (
5      "float sum (float x,float y){return x+y;}");
6    SkelCL::Zip<float> mult(
7      "float mult(float x,float y){return x*y;}");
8    /* allocate and initialize host arrays */
9    float *a_ptr = new float[ARRAY_SIZE];
10   float *b_ptr = new float[ARRAY_SIZE];
11   fillArray(a_ptr, ARRAY_SIZE);
12   fillArray(b_ptr, ARRAY_SIZE);
13   /* create input vectors */
14   SkelCL::Vector<float> A(a_ptr, ARRAY_SIZE);
15   SkelCL::Vector<float> B(b_ptr, ARRAY_SIZE);
16   /* execute skeletons */
17   SkelCL::Scalar<float> C = sum( mult( A, B ) );
18   /* fetch result */
19   float c = C.getValue();
20   /* clean up */
21   delete[] a_ptr;
22   delete[] b_ptr;
23 }
```

To produce the desired result two skeletons are needed: a *Reduce*, and a *Zip*. Upon their initialization, they receive a user-defined function as input (lines 4 to 7). The input elements are stored inside two arrays, used as inputs for two Vectors that, in turn, represent the skeleton's input data (lines 9 to 15). The next steps imitates skeleton nesting, by parametrizing the *Zip* skeleton with the Vectors, and the *Reduce* skeleton with the result of the *Zip* skeleton (line 17). Logically, the Vector generated by *Zip* is the input for *Reduce*. By combining these two skeletons, the desired behaviour is achieved and a scalar result is generated.

As SkePU, SkelCL supports multi-GPU executions. Both approaches are very similar.

Muesli

Muesli is a C++ template library, recently extended to support GPU executions, that offers data- and task-parallel skeletons. It provides support to many- and multi-core parallel

architectures by using MPI and OpenMP. In addition, CUDA is used to enable GPU executions of data-parallel skeletons, such as *Map*, *Zip*, *Fold*, and *Scan*. The latter are offered as member functions of distributed data structures, and used to manipulate elements stored by those structures. On the other hand, task-parallel skeletons, such as: *Farm*, *Pipe*, *DivideAndConquer*, and *BranchAndBound*, are used to construct process topologies.

Muesli's distributed data structures provide support for unidimensional arrays, as well as bi-dimensional matrices and sparse matrices. For the first two, the developer can decide whether the skeleton instances are executed on CPUs or GPUs. However, the sparse matrices are limited to the CPU. Parallelism is dived into inter-node and intra-node parallelization, via MPI and OpenMP/CUDA, respectively. For inter-node parallelization, each MPI created process only stores a part of the distributed data structure. For the intra-node parallelization, either the OpenMP or CUDA threading model is used to distribute the workload among all participant threads. Furthermore, distributed data structures have global and local perspectives, or views. The *local views* refer to the block-wise distribution of the structure, for instance of a matrix into sub-matrices. These are internal representations of the whole structure, and are mostly transparent to the developer, unless he or she selects a skeleton that uses local indices. In turn, the *global view* (the view by default) represents an integral structure. Every skeleton call affects the whole structure, and developers mainly deal with indices of global range.

Muesli exports the concept of higher-order function. The latter are functions that accept other functions as arguments, and/or return functions as results. Given that, in Muesli's programming model skeletons are functions that can take other functions as input, they are denominated as high-level functions. The latter can be overloaded and parametrized via the C++ template mechanism.

By using CUDA streams Muesli gives support to overlap between communication and computation, although it is not user-parametrizable. Muesli also gives support to multi-GPU executions.

Critical Analysis

The SkePU and SkelCL ASkFs are quite similar in respect to execution model, and functionalities. The major difference between them are the mechanisms by which the developers declare the parallel computations. Consequently, it is not surprising that they are afflicted by very similar issues. A direct comparison with Marrow shows that neither SkePU or SkelCL supports more than the *MapReduce* skeleton, and some variants, leaving out task-parallel ones. Also, nesting of skeletons is left out. Logically, these characteristics hamper flexibility, and modularity. Moreover, their skeletons do not introduce overlap between communication and computation, which is advantageous in many application scenarios. We also have considerations in terms of performance when matching these libraries against Marrow, although we will remit such a comparison to Chapter 5.

The two libraries also impose upon themselves limitations that arise from attempting

to abstract developers from the underlying OpenCL kernels. The mechanisms that they offer to the developer, so that he or she can adapt the parallel computations, are very limiting and unsophisticated. SkePU's macro constructs impose severe constrains to kernel implementation possibilities, considering that every kernel argument must have the same data-type (e.g., integer, float, double), and the actual number of arguments is limited to a pre-defined number. This number ranges from one to four, depending on the selected macro. In its turn, SkelCL declares and issues kernel computations as strings, compiled at runtime to OpenCL code. Naturally, it would be preferable if the parallel computations would be transformed and validated at compilation time. To make matter worse, both the macro language and kernel strings limit the scope of a single execution thread (work-item), in that it can only access the data that is directly associated to it, via its global identifier. This direct one-to-one mapping removes a significant amount of development possibilities, in particular to vicinity-like algorithms, and wastes some of the GPU's parallel potential.

Muesli is more complete than SkePU and SkelCL, since its kernels are more sophisticated and have a wider range of implementation possibilities. Be that as it may, it only offers GPU execution through data-parallel skeletons, similar to the ones of SkePU, and SkelCL. Ergo, leaving the task-parallel skeletons for the CPU execution domain. Skeleton nesting is functionality that is available in Muesli, but only in between task-parallel skeletons. Finally, by using CUDA streams, this library enables overlap between communication and computation. Yet, this functionality is not parametrizable.

2.5 Final Remarks

After studying and analysing the current state of the art associated to our proposal's research field, we consider that there is space for contributions in the design and implementation of tools, aimed at the development of intricate GPU applications. In the particular case of skeletons, we think that their use can go beyond the currently supported constructs, offering newer, richer, and more modular mechanisms to efficiently utilize the GPU's capabilities, for both beginners and experienced programmers alike.

3

The Marrow Skeleton Library

This chapter presents the fundamental concepts and functionalities provided by the proposed C++ ASkF, Marrow. Firstly, we expatiate Marrow's execution model and API (Section 3.1). Subsequently, Marrow's kernel context and utilization are detailed (Section 3.2), followed by the fundamentals behind overlap between communication and computation (Section 3.3). Afterwards, Marrow's nesting mechanism is described (Section 3.4), both logically and structurally. The skeletons that are supported by Marrow are then enumerated and defined (Section 3.5). Lastly, Marrow's programming model is described (Section 3.6), in addition to displaying programming examples for its major functionalities.

3.1 Execution Model and API

Our main design goal was to provide a dynamic, efficient, and flexible way of using the GPU for general purpose computing, for both beginners and experienced programmers alike. Marrow proposes to accomplish these objectives by focussing on the orchestration of the execution of OpenCL kernels, whilst additionally introducing optimizations to the overall execution through a technique known as overlap between communication and computation. The parallel computations (kernels) are left to the developer's consideration, being supplied to the skeletons at runtime. These kernels are used by the skeletons to achieve a desirable parallel application scheme.

Considering that Marrow's primary design purpose was to enable efficient GPU executions, we had to take into account the particularities of its architecture and optimize our library accordingly. One prominent particularity is the considerable execution overhead, when issuing GPU computations. While the latter computes, the main processing

unit may be left idle, thus wasting valuable resources. In turn, the decoupling of address spaces, between main and device memories, is also a detail that should be fully managed by the skeletons. Preferably, the applications should only be aware of the principal address space, and that a number of computations have been offloaded to external computational devices. These design goals and restrictions come together to define an execution model where:

- The orchestration of the execution is completely transparent to the developer, while the parallel computations are defined and submitted as regular OpenCL kernels.
- The skeletons seamlessly introduce performance optimizations, that may be tweaked by the developer.
- Skeleton computations are dissociated from application execution.
- The applications only manages the primary address space, that naturally includes skeleton input/output memory spaces.

Delegating the orchestration to the skeletons adds an abstraction layer between the developer and the underlying platform model (OpenCL), greatly reducing the host managing efforts. Even though the parallel computations are still represented as OpenCL kernels, the programmer does not have to manage the base, and low level, functionalities that arise from issuing executions with said kernels (e.g., error handling, memory management/transfers, synchronizing with the device). This abstraction lets the developer concentrate on the parallel computations, rather than spending effort implementing the respective orchestration. In its turn, the dissociation between application and device computations is advantageous, given that it makes sense to free up the application to perform additional computations (e.g., preparing the next input, evaluating the previous results) while the device carries out an execution.

Marrow's execution model is depicted in Figure 3.1. This model can be regarded as a master/slave scheme, where the application offloads asynchronous execution requests to the skeletons. An execution request comprises memory references from which the skeleton obtains the input, and to where it writes the computed results. These execution requests are always processed in a FIFO order, although not always sequentially (this idea is further elaborated in Section 3.4). Nevertheless, when the application thread submits an execution request to a skeleton (step 1 in the figure) the latter does not process it immediately. Instead, this request is queued, a future object is created (step 2) and a reference to it is returned to the application (step 3). The future allows the application to, not only, query the state of the execution, but also, wait (block) until the results are ready (step 4). When the skeleton becomes available to fulfil the request, the execution is submitted to the device (step 5). Once the results are ready, they are read to the targeted output memory (step 6), and the respective future is notified (7), an event that will wake up any application thread waiting for the results (step 8).

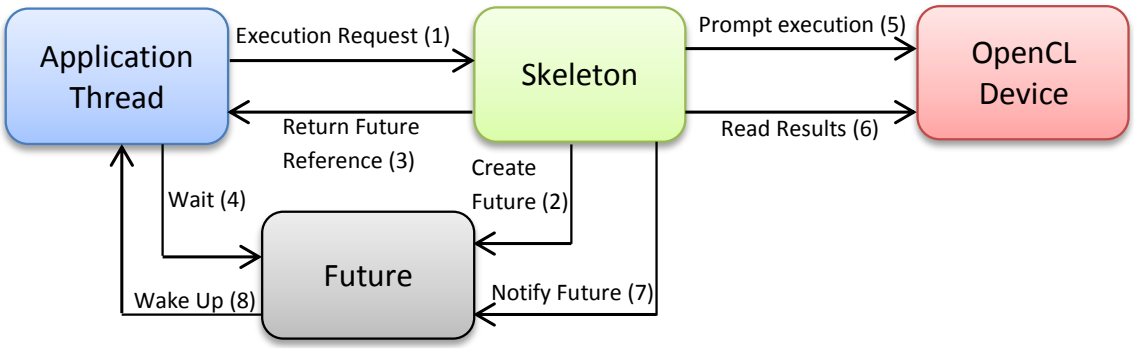


Figure 3.1: Marrow's execution model

This computational scheme motivates a rather simple API. Issuing a skeleton execution is accomplished by using an asynchronous write operation. The latter requests an OpenCL execution, and renders a future object. Additionally, this operation is parametrizable with C++ STL vectors for both input and output memory references. Naturally, the latter must be valid, and pre-initialized, input/output address spaces. Moreover, these memory references are accepted as C void* type. We selected untyped memory references over a generic typification mechanism (e.g., C++ templates) because it proved to be more suitable to a scenario where the kernel's arguments may vary in number and in type. On the other hand, the OpenCL language is very non-restrictive about the types of data loaded onto device memory, provided they are supported by the language, and each may be stored in a contiguous memory space. Therefore, Marrow supports any arbitrary combination of kernel arguments, asserting that their data-types (as enumerated in Section 3.2) are supported by the ASkF runtime.

Considering the current proliferation of parallel architectures, multi-threaded computing is becoming standard. In this context thread-safety is an important feature. On top of that, thread-safety allows for skeletons to be shared among multiple application-threads, reducing the memory footprint, equally on host and device sides. Consequently, Marrow's API, notably the write operation, is thread-safe. This removes the necessity of using synchronization mechanisms to perform access control, between application-threads that issue executions on the same skeleton.

3.2 Kernels

As previously stated, Marrow's programming model only abstracts the OpenCL kernel execution orchestration, rather than the OpenCL kernels. However the latter are usually associated to fundamental information about the computations (e.g., the execution resource requirements, kernel parametrization information). Some of this information is very closely bounded to the runtime parametrization (e.g., thread work-load, memory requirements), restraining any attempts at retrieving it solemnly with parsing techniques.

This active information is used by the skeletons to adapt their execution to a scheme that provides the correct computational behaviour.

Therefore, we deemed as beneficial to encapsulate (wrap) the kernel's logic and domain in a single executional object, allowing skeletons to easily access this information when orchestrating a kernel execution. The resulting object, named *KernelWrapper*, provides multi-level functionalities that classify it as an executional entity, instead of a simple information placeholder. These functionalities are enumerated as follows:

Compilation – This functionality ensures that any kernel supplied by the developer is valid according to the OpenCL specification. This assertion is checked via the OpenCL kernel runtime compilation, raising appropriate errors if necessary.

Information supplier – The *KernelWrapper* allows skeletons to query kernel execution related information through a standard interface. The obtainable information includes some fundamental aspects, such as: argument data-types and data-sizes, memory space requirements (global/local), NDRange index size (global/local).

Execution – The wrapper enables external computational entities, such as skeletons, to prompt kernel executions. Naturally, each *KernelWrapper* can only prompt executions within its own kernel domain, and when parametrized appropriately.

The instantiation of a *KernelWrapper* requires the definition of the kernel's argument information, separately for both input and output. This definition encompasses both structural and contextual information. The former designates the argument-layout, the required data-structures, and the data-types. The latter determines runtime specifications, such as required memory, work-load (work-items/groups), and kernel file path. In particular, the supported kernel argument data-types are:

- **Buffer** – Unidimensional arrays of elementary C++ and OpenCL types, and/or of C structs.
- **Singleton** – A single data element of a certain type, that may vary throughout multiple executions of a kernel. It is considerably faster to load onto device memory than a single element buffer.
- **Final** – This type of data is homologous to the singleton, but it is constant throughout multiple executions of a kernel.
- **Local** – OpenCL supports local memory as a way to increase the performance of kernel executions. It is useful to, for example, avoid unnecessary read operations generated when different work-items, within a work-group, access the same address of global memory.
- **Image2D** – This type of data is somewhat analogous to a buffer object. However it is naturally more appropriate when the application is working with images, since the OpenCL kernel API is rich in functions pointed out to image processing.

When instantiating a *KernelWrapper*, OpenCL's memory hierarchy dictates the utilization possibilities for the previous data-types, in that each one is only associated to a particular memory address space. This correlation somewhat constrains how the kernel's arguments may be organized, since the layout must be compatible with an OpenCL execution. Moreover, the OpenCL specification states that a kernel's output arguments must point to a region of global memory. Consequently, when using Marrow a programmer may only define a *Buffer*, or *Image2D*, as outputs for a kernel, regardless of their quantity. On the other hand, every data-type can be used as input, and combined arbitrarily. Table 3.1 illustrates the pairing between every supported kernel argument data-type, and the relevant memory address spaces of the OpenCL hierarchy. The *Other Memory* category relates to additional address spaces present in OpenCL's memory scheme (e.g., constant memory, compute unit registers).

	Global Memory	Local Memory	Other Memory
Buffers	X		
Singleton			X
Final			X
Local		X	
Image2D	X		

Table 3.1: Association between argument data-type and memory address space

The *KernelWrapper object* is a passive object. That is, it never actively provides information to another executional entity. A skeleton always acts as a questioner when requiring the knowledge of a kernel's executional information. For instance, consider a kernel k that uses two buffer objects of type T, each holding 500 elements. Upon querying the *KernelWrapper*, the skeleton learns that the execution requires two T buffer objects, each occupying $\mathsf{sizeof}(T) \times 500$ bytes of memory. This information allows the skeleton to manage its resources accordingly, and even to perform optimizations that yield performance gains.

3.3 Overlap Between Communication and Computation

Modern GPUs are capable of performing simultaneous bi-directional data transfers between memories (host and device), while executing computations related to one or more kernels. This ability allows developers to introduce another level of concurrency at the computational level. Not only can the computations be executed in parallel, in accordance the SIMD model, but the decoupling between memory spaces, when combined with the GPU's previously stated capabilities, induces developers to submit the operations in a such a fashion that reduces the GPU's idle time. This technique is refereed to as overlap between communication and computation. However, such an execution scheme adds a serious amount of complexity to the host orchestration. For this reason, hiding

this complexity inside a skeleton proves to be ideal.

Our approach identifies three classes of operations that may take advantage of this overlap: transfers to device memory (writes), kernel executions, and transfers from device memory (reads). Usually these classes of operations are associated to a particular kernel execution, meaning that operations associated to a specific data-set are independent from the remainder. Therefore, our skeletons reach overlap by optimizing the interweaving of operations associated to distinct data-sets in a manner that enables their parallel execution, hence seamlessly increasing overall performance. Figure 3.2 depicts an operation execution order where the operations, relative to three different data-sets, are computed in an order that minimizes the device's idle time. In this exemplification, after write W_1 is completed the device proceeds to carry out W_2, doing so in parallel with the kernel execution K_1. Subsequently, after K_1 is completed, the device starts read R_1, while executing K_2. This scheme continues for the successive operations.

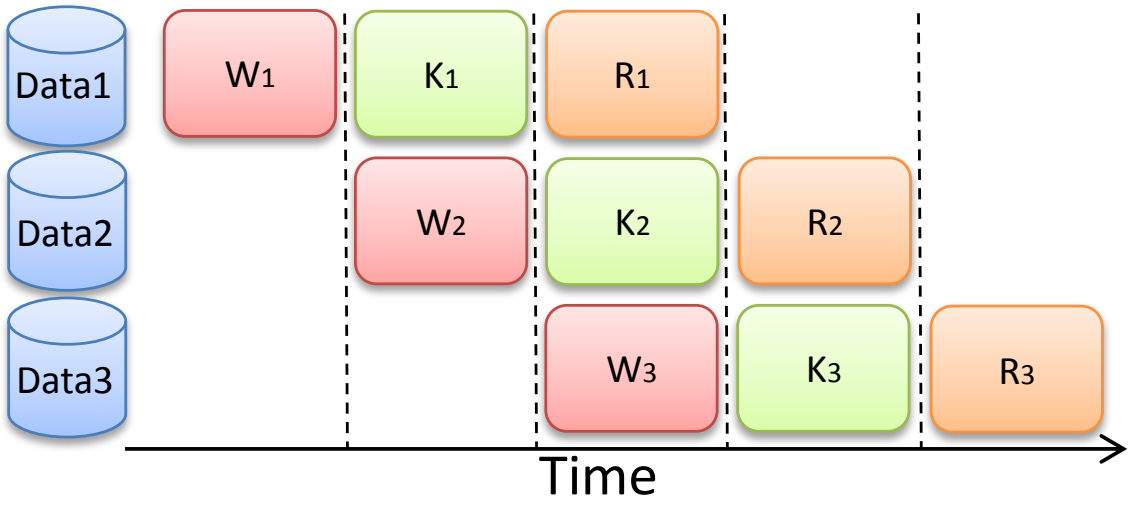


Figure 3.2: An overlapped execution order

Be that as it may, not all applications benefit from the overlap. For instance, memory transfer bound applications usually tend to not benefit as much as computation heavy applications. The perfect scenario, i.e. that yields the highest performance increase, would present itself when the three classes of operations, of a particular data-set execution, take an approximate amount of time to complete, just as in the illustrative example. Furthermore, the performance increase provided by the overlap is accumulative, consequently it is more significant when the skeleton is used to process a high number of data-sets before being deallocated. These preliminary assessments are further elaborated, and supported by experimental results, in Chapter 5.

The overlap mechanism is closely connected to the nesting mechanism, such that the former uses the latter to introduce its functionality to every nested skeleton, as expatiated in the subsequent section. This compatibility removes the requirement for all skeletons to provide overlap as a base functionality, which simplifies the skeleton design process. Therefore, only a very particular set of skeletons provides overlap, being its functionality

widespread to the remainder via nesting. Overlap enabling skeletons are highlighted in Section 3.5.

3.4 Nesting

Skeleton nesting is a design functionality for the development of intricate, often multi-kernel, parallel applications. It is characterized by the ability to couple different skeletons into a single multi-level construct, that encloses a diverse set of behaviours. Accordingly, from the developer's perspective, employing nesting to build a more sophisticated application is analogous to developing a complex system by combining distinct software modules. This added flexibility is also beneficial to the skeleton design process, in that each skeleton is simpler, and only offers a few specific functionalities, not present in other skeletons.

A nested skeleton application can be regarded as a composed acyclic graph (composition tree), on which every node shares a general computational domain. The tree's nodes can be categorized into three classes, based on the interactions with their ancestor/children, on the resources they manage, and on the types of operations they issue to the device. These three categories are: root skeleton, inner skeleton, and leaf node. At minimum, a composition tree has two levels, being comprised of one root with one or more leafs. More complex trees have inner skeletons, which naturally increases the height of the tree.

The root node is the primary element of the composition tree. It is responsibly for processing the application's execution requests, which implies submitting one or more OpenCL executions. Therefore, the root must manage most of the resources necessary to accomplish such execution, as well as performing data transfers between host and device memory. Additionally, it prompts executions on its children, parametrizing them with a specific set of context resources. For instance, the OpenCL structures that they must use to issue executions, or memory objects that hold the input data and are targets for output. The root must also ready the resulting data for the application, by setting it on the desired memory cells and notifying the respective future object. Lastly, the effectiveness of the overlap between communication and computation is directly proportional to the position, on the composition tree, of the node that applies it. The higher it is, the more sub-trees it affects. Hence, in order to maximize performance, overlap between communication and computation is always applied by the root node.

The inner nodes are skeletons whose role is to introduce a specific execution pattern/behaviour to their sub-tree. These nodes might not need to allocate resources, since they are encased in a computational context created by the root. This computational context is also used by the inner node when issuing executions on its children.

Leaf nodes should not be referred to as skeletons because they do not introduce a specific execution behaviour, instead they export an executional object (kernel). As a result, leaf nodes are represented by *KernelWrapper* objects, which, in turn, are used to finalize

the construction of the composition tree. The *KernelWrapper* abstracts other executional entities from the notion of kernel. By doing so, it helps to standardize the concept of a Marrow executional entity, i.e., a composition tree node.

A skeleton whose executional pattern requires the manipulation of input/output data on host memory is incompatible with the nesting mechanism, and thus can only be used as a root node. In turn, to be used as an inner node (to support nesting), a skeleton must be able to perform its execution on pre-initialized device memory, issued by its ancestor. Furthermore, it should be able to share an execution environment with other nodes, even if it adds state (e.g., memory objects, executional resources) to that environment. In any case, a skeleton that supports nesting is also eligible to become the root of a composition tree.

3.5 Skeletons

Marrow currently supports the following set of task- and data-parallel skeletons: *Stream*, *MapReduce*, *Pipeline*, *Loop*, and *For*.

3.5.1 Stream

The *Stream* skeleton, depicted in Figure 3.3, defines a computational structure that confers the impression of persistence to an OpenCL execution. This notion is achieved by introducing parallelism between device executions associated to distinct data-sets, particularly, between the three classes of operations previously identified in Section 3.3. The *Stream* is able to apply this parallelism even if the ensemble of input data is supplied to the skeleton over time, in a discrete manner. As a result, the *Stream* introduces overlap between communication and computation in the skeleton execution. To simplify the overall framework design, only the *Stream* natively provides this functionality. If this behaviour is desirable elsewhere, it is obtainable via nesting on a *Stream*, or its direct usage. Even if other skeletons provide overlap as a stand-alone it is done by, in some way, using a *Stream*.

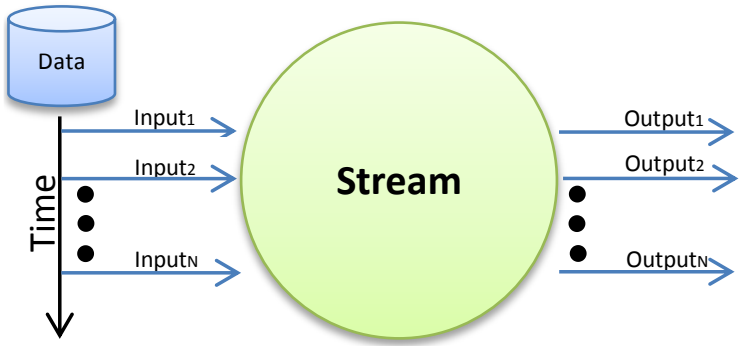


Figure 3.3: The *Stream* skeleton

As expected, this execution behaviour requires direct control over the input data. As

a consequence, the *Stream* is only qualified as a root node, and thus, cannot be nested. However, this is not a limitation, since it is desirable to provide overlap to as many nodes as possible. Ergo, the *Stream* is most useful when it is the root of the composition tree. By definition, this skeleton applies the same computation to every data-set, regardless if the former is a kernel or a sub-skeleton. Consequently, the *Stream* is classifiable as a data-parallel skeleton.

3.5.2 MapReduce

The proposed *MapReduce* skeleton, illustrated in Figure 3.4, applies the same computation to independent partitions of a given data-set, and, as such, is classifiable as a data-parallel skeleton. Its execution pattern is separated into three stages:

1. **split** – This stage divides the input data into a user-defined number of partitions. It is computed on the host side.
2. **execute** – This stage issues executions on each of the data partitions. Each of these executions maps the kernel to a set of elements from the input data-set. It is executed by the OpenCL device.
3. **merge** – This stage aggregates (reduces) the results of the executions. It is carried out on the host side.

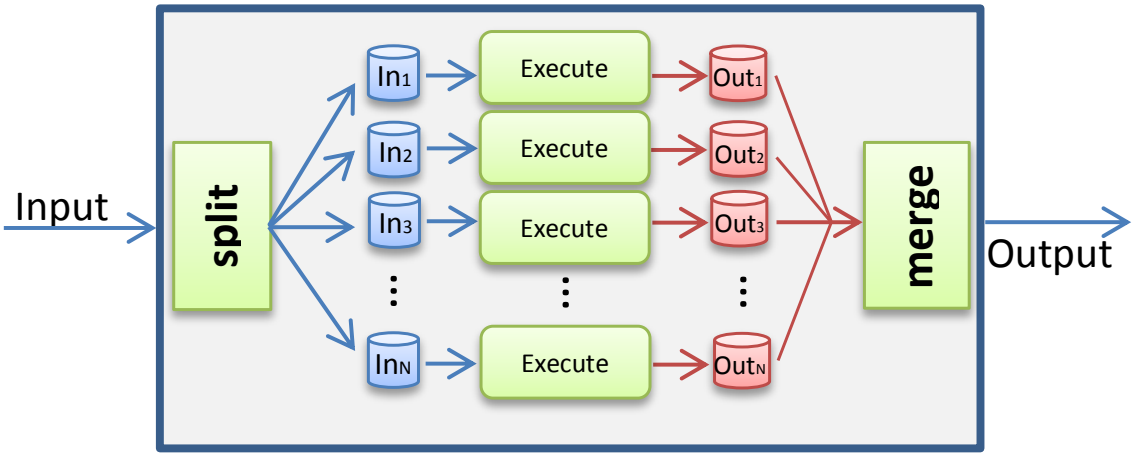


Figure 3.4: The *MapReduce* skeleton

Considering that this construct is designed for GPU computing, its behaviour differs from the general definition of a *MapReduce* skeleton, such as Skandium's Map. In this context, the overheads that originate from a GPU execution suggest that the latter should only be used to process a large amount of data-elements. Ergo, an input data-set is split into partitions, instead of singular data-elements, and an OpenCL execution is issued on each one. On the other hand, GPUs are not particularly efficient when aggregating, or reducing, all the elements of a data-set into a single scalar value. Instead, it is preferable

to reduce part of the data on the GPU and return N number of elements, where N is a power of two larger than a certain threshold, to be finally reduced on the CPU. This threshold varies across GPUs, and its selection should result from an analysis identical to the one presented in Chapter 2, in the end of Section 2.1. Nonetheless, this strategy is compatible with the utilization of a GPU reduction. To this end, the developer should implement a reduction kernel and used it when instantiating a *MapReduce*.

To adjust the behaviour of the *MapReduce* skeleton the developer must implement two functions: split, and merge. The first receives the input data and divides it into partitions, whilst the second receives the output of each execution and combines them into a single value. Both functions work with memory references and do not require in-function data transfers, which in turn increases their overall efficiency.

Splitting the input data into several partitions offers a great opportunity to use overlap between communications and computations, particularly between executions associated to distinct partitions. As such, the *MapReduce* skeleton overlaps the operations associated to each partition, increasing application performance. Should be noted that this skeleton introduces overlap as a standalone, i.e. does not need to be nested with any other skeleton. The reasons behind this will become clearer in the next chapter.

Concerning the nesting mechanism, the *MapReduce* is not nestable, since it directly manipulates both the input and output data. This decision was made considering that in our execution model it makes more sense, performance-wise, to divide the input in the host-side and then to apply overlap, than to write everything to device memory and subsequently sub-divide the resulting memory objects.

3.5.3 Pipeline

The proposed *Pipeline* skeleton, illustrated in Figure 3.5, is analogous to the one presented in Section 2.4, of Chapter 2. It allows the developer to efficiently combine a series of data-dependant serializable tasks, that can exploit the overlap between communications and computations. Bearing into consideration that memory transfers between main and GPU memories introduces a significant overhead, this behavioural scheme is ideal for GPU execution since the intermediate data does not need to be transferred back to main memory for it to become available to next stage. In turn, the application of distinct computations to different data-sets classifies the *Pipeline* as task-parallel skeleton.

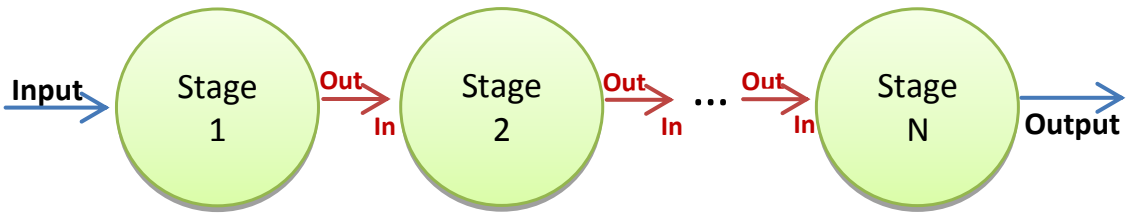


Figure 3.5: The *Pipeline* skeleton

This execution pattern is suitable for an execution that starts with pre-initialized device memory objects, and is fully compatible with a shared execution environment. Accordingly, this skeleton supports nesting. For the sake of simplicity, a single *Pipeline* features only two stages. Similarly to Skandium, the construction of pipelines with arbitrary number of stages must resort to nesting.

Overlap is very advantageous to a pipelined execution, since it enables overlap between memory transfers and kernel executions, as well as between stages, assuming that the computing device supports multi-kernel executions. This characteristic boosts the *Pipeline*'s parallelism factor, and naturally the application's performance. However, the *Pipeline* by itself does not provide overlap. Such behaviour can only be attained by nesting a *Pipeline* in an overlap skeleton.

3.5.4 Loop and For

The *Loop* skeleton, depicted in Figure 3.6, presents a construct that applies an iterative computation to a given data-set, while a certain condition is met. The *Loop* supports two distinct computational strategies: one where its condition is affected by changes that occur externally to the execution domain (like in a *for* loop), and another where the condition is affected by partial results, computed by every iteration (like in a *while* loop).

When the *Loop* is parametrized with input data, for instance after the application invokes write, it starts by checking the validity of the condition. If the latter is valid, the computations (e.g., kernel, sub-skeleton) are applied to the input data. After the execution is completed the *Loop* progresses its state, using the appropriate computational strategy. At this point, if the partial results affect the *Loop*'s state they are read to host memory, allowing them to be processed. Therefore, the *Loop*'s condition is re-evaluated, inducing the usual semantics of a *while* loop. If the condition is still valid, after the last step, the process repeats itself. Otherwise, the *Loop*'s state is reset and the last partial results are presented as the final results.

Therefore, in this scenario, the overlap between communications and computations is very beneficial since the device may conciliate several *Loop* executions, each performed on a independent input data-set. For example, the device might execute one kernel instance while performing partial reads for another. Be that as it may, to apply overlap to its computations the *Loop* must be nested in a skeleton that provides it, like a *Stream*.

To support overlap between communications and computations the *Loop* skeleton must be able to provide an isolated loop state environment to each execution, letting the latter evolve independently from each other. A loop state wraps the necessary information about the manner in which the *Loop* executes: how to evaluate its condition, and how to progress and reset its state. As a result, the *Loop* manages multiple state environments, allowing the operations issued within an environment to be carried out independently, and concurrently, with others.

In order to adapt the *Loop*'s computational behaviour to a specific application, the

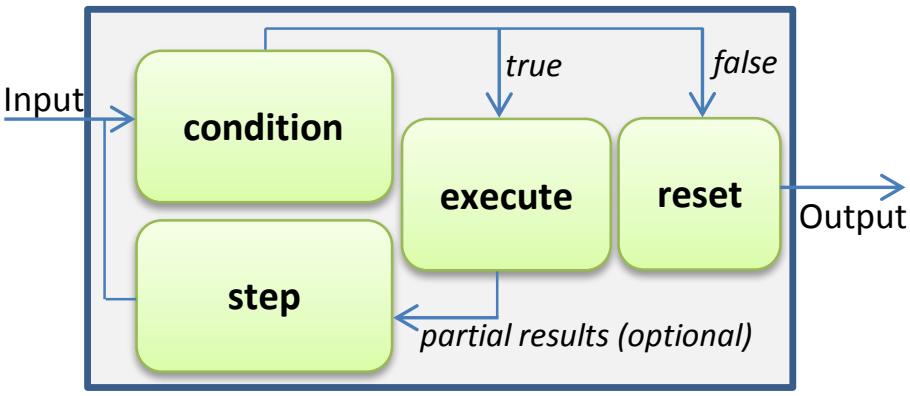


Figure 3.6: The *Loop* skeleton

developed must provide concrete implementations of the following functions, associated to a loop state environment:

condition – Evaluates the condition that determines if the *Loop* should continue or terminate. It returns true or false according to the need to reapply the computations to the data.

step – Function executed after each iteration. It allows the advancement of the respective state, using for that matter, or not, the partial results computed by the last iteration. For instance, in case of a *for* loop this function simply decrements the number of remaining iterations. In turn, in case of a *while* loop this function would change the variables evaluated in condition in accordance with the partial results.

reset – Resets the state to its original values, allowing it to be re-utilized with other data-sets.

The *Loop* must be able to create environments on demand, for it to enable overlapped and independent loop executions. Therefore the programmer must implement a factory-like functions called: createState. This function returns a reference to a pre-initialized state environment. Given that the latter are resettable, the *Loop* needs only to create as many as deemed necessary by parametrization values, issued by the application.

An iterative executional pattern, that always applies the same computations to every data-set, is able to execute upon pre-initialized device memory objects, and to share an execution environment with other nodes. Hence, the *Loop* supports the nesting mechanism, and it is classifiable as a data-parallel skeleton.

In turn, the *For* skeleton is a specialization of the *Loop* skeleton that does not require implementing any functions, simplifying the developer's job. Logically, a *For* skeleton applies the same computations a user-defined number of times to the input data.

3.6 Programming Model

3.6.1 Programming Model Structure

Marrow's programming model comprises three major stages: skeleton initialization, prompting of skeleton executions, and skeleton deallocation. This model is familiar to developers that have used an object-oriented programming language, as the skeletons are utilized as regular objects that provide a particular functionality. Moreover, these skeleton objects contain an implicit parallel computation scheme, that provides its own synchronization mechanism. The latter is represented by the future object, yielded after each successful skeleton execution prompt.

Skeleton initialization

The skeleton initialization stage defines the structures and behaviours that are successively used to issue OpenCL executions. This stage is the most complex and verbose of all three, as it encompasses the skeleton instantiations, and the *KernelWrapper* declarations. Instantiating a skeleton involves parametrizing it with other execution entities (e.g., skeletons, *KernelWrappers*) via its constructor, which is the basis for Marrow's nesting mechanism. Nesting is a simple task of instantiating a skeleton with the appropriate executional entities, in order to build the desirable composition tree. Moreover, a skeleton that provides overlap can be parametrized with values that affect how many memory-object sets are allocated.

In its turn, the *KernelWrapper*, used as a leaf in the composition tree, requires additional parameters when being instantiated. Firstly, the developer must define the OpenCL work-load used by the kernel. A usual OpenCL work-load has up to three dimensions of work-items, further divided into work-groups. The next step is to define the input and output argument information. This information is specified by parametrizing the wrapper with a set of data-type objects exported by Marrow, that extend a IWorkData super type. These objects include every data-type listed in Section 3.2, and contain relevant information such as: number of data-elements, basic type, and access permission. This data is used to correctly allocate the memory objects on which the kernel instances compute. The developer must guarantee that the order and type of the data-type objects (both input and output), used to parametrize the *KernelWrapper*, match the syntactic representation of the kernel arguments, in the kernel source. Lastly, the *KernelWrapper* requires a kernel source file path, and a kernel function name, both passed as strings.

Marrow takes advantages of some functionalities available in C++11. For example, the use of smart pointers when nesting skeletons. This mechanism ensures a safe utilization of the inner nodes, and safeguards that the same instance is not mistakenly used in multiple composition trees. Furthermore, skeleton constructors are only parametrized with smart pointers, that reference other executional entities. As such, only the root node can be allocated on the stack. Considering that prompting a skeleton execution is done

through the root node this mechanism is acceptable, and facilitates the root's internal resource management routines.

Prompting of skeleton executions

This state specifies how the application issues execution requests. By calling the root's write function the application issues a single skeleton execution, that is parametrized with input and output untyped memory references. This scheme is obviously affected by Marrow's asynchronous execution model. After issuing an execution request, the application is free to perform any desirable computations, which are performed in parallel with the skeleton execution. This flexibility allows an application thread to submit, in succession, multiple execution requests without blocking between them. This scheme can become conceptually more complex by using a multi-threaded application, on which various threads concurrently issue execution requests. In any case, synchronizing with the skeleton is always done by using a future, returned from soliciting an execution request.

Using untyped memory references as parameters has a singular advantage: it enables the logical division of the input/output data by means of pointer arithmetic. Obviously this is more efficient than memory copies, albeit somewhat more complex for beginners. Given that Marrow does not perform memory copies other than to/from device memory, the developer may introduce overlap to a singular data-set without in memory divisions. This functionality is exemplified in Subsection 3.6.3. However, not performing host-side memory copies means that Marrow does not store the input data prior to its transfer to the device. Accordingly, it is up to the developer to ensure that the input is always available throughout the hole respective skeleton execution, up to its completion.

Skeleton deallocation

The final stage is trivial. Since the root node manages every skeleton resources used to build the composition tree, removing it implicitly deallocates every branch. This includes inner skeletons, as well as *KernelWrappers*. Note that if the root skeleton object is allocated on the stack instead of the heap, this step is unnecessary.

3.6.2 Comparison between Marrow's and OpenCL's models

Table 3.2 presents a comparison between the programming model of both, OpenCL and Marrow. It is observable that the latter has a simpler, and of higher-level, model than the former. As a result Marrow's programming model is friendlier towards inexperienced programmers since: there are few to none low-level programming concerns, it is less error prone, and the amount of code generated is smaller. Additionally, the asynchronous API improves the flexibility of the resulting programs, without hampering the simplicity of our execution model. Nonetheless, our model is still useful for advanced programmers that know how to use the OpenCL API to develop complex and efficient parallel applications.

OpenCL	Marrow
1. Context plus command-queue creation. 2. Compiling the kernel code and creating the respective kernel. 3. Memory object allocation, for both input and output data. 4. Input memory objects initialization, usually through write operations. 5. Prompting a kernel execution with specific kernel arguments. 6. Waiting for the results, and subsequently reading them back to host memory. 7. Deallocation of resources.	1. Skeleton creation. This creation covers skeleton nesting and kernel argument definition (stage 1). 2. Issuing execution requests. This step requires that the input and output memory addresses are provided to the skeleton (stage 2). **(Optional)** – At this point other application computations not related to the skeleton execution. 3. Waiting on the future for the results to become available (stage 2). 4. Skeleton deallocation (stage 3).

Table 3.2: Execution pattern of OpenCL and the proposed skeletons

The previous simple OpenCL application pattern does not introduce overlap between communication and computation. If it were to be introduced, the OpenCL application would become considerably more complex, and delve into particular aspects associated to C/C++ concurrency mechanisms. Yet, Marrow transparently supports this functionality. In Chapter 5 we present a more in-depth comparison between these two models, namely in regards to productivity (number of lines of code).

3.6.3 Programming examples

Following we present some programming examples that illustrate how to use Marrow's core functionalities. The presentation follows the model's stage sequence.

Skeleton/*KernelWrapper* Initialization

Listing 3.1 depicts a simplified code snippet that shows how to instantiate, and use, a *KernelWrapper*. This example applies a Gaussian Noise image filter to a bi-dimensional image, of width imgWidth and height imgHeight (line 2). The work-load is specified via two vectors (lines 9 to 16), each of size two (two dimensions). For this kernel, the NDRange index and the image should have equal proportions. In fact, the local work-size parameter (which is optional) is only included to provide an utilization example. Regardless, the global and local work-sizes must have the same number of dimensions.

Listing 3.1: Initialization of a basic composition tree

```
1  unsigned int numberPixels; // Number of image pixels
2  unsinged int imgWidth, imgHeight; // Image width and height
3  std::string gaussNoiseKernelFile; // Gaussian Noise OpenCL kernel source file
4  unsigned int numMOS; // Number of used memory object sets for overlap
5  unsigned int blockSize; // Number of work-items per group, in each dimension

7  /****** STAGE 1: Skeleton Initialization ******/
8  // Number of work-items in each dimension
9  std::vector<unsigned int> globalWorkSize(2);
10 globalWorkSize[0] = imgWidth; // x-dimension
11 globalWorkSize[1] = imgHeight; // y-dimension

13 // Number of work-items in each dimension. This parameter is optional
14 std::vector<unsigned int> localWorkSize(2);
15 localWorkSize[0] = blockSize; // x-dimension
16 localWorkSize[1] = blockSize; // y-dimension

18 // Instantiate kernel input argument info
19 std::shared_ptr<IWorkData> bufferInfo(new BufferData<cl_uchar4>(numberPixels));
20 std::vector<std::shared_ptr<IWorkData>> gaussInputData(2);
21 gaussInputData[0] = bufferInfo;
22 gaussInputData[1] = std::shared_ptr<IWorkData> (new SingletonData<int>());

24 // Instantiate kernel output argument info
25 std::vector<std::shared_ptr<IWorkData>> gaussOutputData(1);
26 gaussOutputData[0] = bufferInfo;

28 // Instantiate KernelWrapper
29 std::unique_ptr<IExecutable> kernel (new KernelWrapper(gaussNoiseKernelFile, "
       gaussian_transform", gaussInputData, gaussOutputData, globalWorkSize,
       localWorkSize));

31 // Instantiate root skeleton with numMOS memory object-sets
32 Stream *s = new Stream(kernel, numMOS);

34 /****** STAGE 2: Prompting of Skeleton Executions ******/
35 // ... Shown in another listing

37 /****** STAGE 3: Skeleton Deallocation ******/
38 delete s;
```

The following step is to define the kernel argument information, for both input and output parameters (lines 18 to 26). This Gaussian Noise kernel is parametrized with an input image, and outputs another. Consequently, it uses two cl_uchar4 buffers, both of size imgWidth $\times$ imgHeight (numberPixels). In this example, the images used as input have a color depth of 255, in a RGBA color space. Hence, a single, four byte, cl_uchar4 can store an individual pixel. In addition, the kernel uses a user-defined scalar value in its computations, represented as the second input argument (line 22).

Finally, the *KernelWrapper* is ready to be instantiated, and parametrized with the appropriate values (line 29). Creating a skeleton, for instance a *Stream*, is a simple process of using the appropriate executional entities as constructor arguments, in this case a *KernelWrapper* (line 32). To adjust the overlap provided by the *Stream* a parameter numMOS is used. This tells the skeleton that at any given point it should be able to concurrently dispatch a maximum of numMOS execution requests.

The second stage of Marrow's programming model is exemplified later on, in this Subsection. In turn, the third stage is fully represented in this example at line 38.

Loop Instantiation

Some skeletons are provided as abstract C++ objects, naturally requiring the implementation of their pure virtual members. An example of such an abstract skeleton is the *Loop*. As expatiated in Section 3.5, a *Loop* skeleton must be able to create loop state objects on demand. The latter adapt the skeleton's logic to a particular execution scheme, in which each state object is, implicitly at any given moment, at most connected to a single ongoing execution request. Listings 3.2 and 3.3 show the code for an instantiation example. The implemented *Loop* is intended to keep issuing OpenCL executions until the standard deviation of the resulting set of elements is lower than a given threshold. The used kernel is not relevant, as the reader may consider that it consumes an input buffer and outputs an equivalent one. The kernel operation, applied to the input data-elements, is elementary and of little interest to this example.

Listing 3.2: Declaring a *Loop* state class

```
1  /** Mean loop state class **/
2  class DeviationLoopState: public ILoopState {
3  public:
4    DeviationLoopState(unsigned int numElements, float threshold):
5    numElements(numElements),
6    threshold(threshold),
7    cond(true)
8    {}
9
10   ~DeviationLoopState(){}
11
12   // @Abstract
13   bool condition(){
14     return cond;
15   }
16
17   // @Abstract - Calculate standard deviation
18   void step(std::vector<void*> &previousOut){
19     float deviation = 0;
20     float *values = (float*) previousOut[0];
21     // Calculates standard deviation on values array
22     deviation = calculateStdDeviation(values);
```

```
23      // Affect loop continuity
24      if(deviation < threshold){
25        cond = false;
26      }
27    }
28    // @Abstract - Reseting the loop state for consequent skeleton execution
29    void reset(){
30      cond = true;
31    }
32  private:
33    unsigned int numElements; // Number of data elements in calculation
34    float threshold; // Standard deviation threshold
35    bool cond; // Continuity condition
36    // Sets cond as false if standard deviation is lesser than threshold
37    float calculateStdDeviation(float* values){/** ... **/}
38  };
```

To implement a loop state object, Listing 3.2, the developer must declare a C++ class that extends the ILoopState interface (lines 1 to 38). The condition and reset functions are trivial, and self-explanatory (respectively lines 13 to 15, and 29 to 31). On the other hand, the step function calculates the standard deviation from the output buffer, taken from the previousOut vector (line 20). It then decides whether it should continue, or stop, the respective execution (lines 24 to 26).

Listing 3.3: Initialization of a *Loop*

```
1   float threshold; // Standard deviation threshold
2   unsigned int numElems; // Number of elements on the arrays
3
4   /** Mean Loop class **/
5   class DeviationLoop: public Loop {
6   public:
7     DeviationLoop(std::unique_ptr<IExecutable> &exec, unsigned int numElems,
        float threshold):
8     Loop(exec, true),
9     numElems(numElems),
10    threshold(threshold)
11    {}
12  private:
13    unsigned int numElems;
14    float threshold;
15
16    // @Abstract - Create Loop States on demand
17    DeviationLoopState* createState(){
18      return new DeviationLoopState(numElems, threshold);
19    }
20  };
21
22  // KernelWrapper object instantiation
23  // ... parameter definition
```

```
24 std::unique_ptr<IExecutable> kernel (/* ... Parametrized as previously */);
25 // Instantiate Loop object
26 DeviationLoop *ml = new DeviationLoop(kernel, numElems, threshold);
27 // Given that it is a nestable skeleton is must be explicitly initialized
28 ml->start();
```

To utilize a *Loop*, Listing 3.3, the programmer must implement a class that extends the abstract Loop class (lines 4 to 20). The first aspect to consider is whether the partial results are necessary to assert the execution's continuity. If so, the *Loop*'s constructor should be called with a true boolean value in its second argument (line 8). The simplest *Loop* constructor considers the partial results necessary, by default. As expected, the standard deviation loop naturally needs to process the partial results. The second aspect is the implementation of the createState function. The example function simply allocates and returns references to the user-defined DeviationLoopState objects (17 to 19). All that is left to do is instantiating a DeviationLoop (line 26), using the appropriate *KernelWrapper*. Should be noted that since this skeleton is nestable and is being used as a root, it has to be initialized explicitly (line 28).

MapReduce Instantiation

Similarly to the *Loop*, using the *MapReduce* skeleton involves declaring a base derivative class that extends the MapReduce abstract class. Listing 3.4 illustrates such an utilization example. In it, the skeleton is used to apply a trivial arithmetic operation to an input array, and to subsequently reduce the results into a single scalar value. Again, the kernel is not, in itself, relevant to the example. Notwithstanding, there is no device reduction being applied, only a host side one. The implementation process begins by the definition of a class that extends MapReduce, namely MyMapReduce class (lines 6 to 48). It expounds the two main functionalities: splitting the input, and merging the results.

Listing 3.4: Initialization of a *MapReduce*

```
1  unsigned int numberElems; // Total number of elements
2  unsigned int numMOS; // Number of buffer sets used for overlap
3  unsigned int numDivisions; // Number of partitions generated

5  /** MapReduce class **/
6  class MyMapReduce: public MapReduce {
7  public:
8    MyMapReduce(std::unique_ptr<IExecutable> &map, unsigned int dataSize,
         unsigned int numDivisions,unsigned int numMOS):
9    MapReduce(map, numDivisions, numMOS),
10   dataSize(dataSize)
11   {}

13   MyMapReduce(std::unique_ptr<IExecutable> &map, unsigned int dataSize,
         unsigned int numDivisions):
14   MapReduce(map, numDivisions),
```

```
15    dataSize(dataSize)
16    {}

18    // @Abstract - Split the input data into partitions
19    void split(const std::vector<void*> &input, std::vector<std::vector<void*>> &
          splited){
20      unsigned int numDiv = getNumDivisions(); // Number of divisions
21      unsigned int offset = dataSize / numDiv; // Number of elements per
            partition
22      // Defining where the partitions start on the input array, for each
            division i
23      for(unsigned int i = 0; i < numDiv; i++){
24        splited[i][0] = input[0] + (i * offset * sizeof(float));
25      }
26    }

28    // @Abstract - Reduce the output results, for each partition execution
29    void merge(const std::vector<std::vector<void*>> &results, std::vector<void*>
           &output){
30      unsigned int numDiv = getNumDivisions(); // Number of divisions
31      unsigned int workSize = dataSize/numDiv; // Number of elements per
            partition
32      float *outValue = (float*) output[0]; // Result placeholder
33      float aux = 0; // Auxiliar counter
34      for(unsigned int i = 0; i < numDiv; i++){
35        // Get partial result (the output array)
36        float* result = (float*) (results[i][0]);
37        // Reduce all values in the partial result
38        for(unsigned w = 0; w < workSize; w++){
39          aux += result[w];
40        }
41      }
42      // Save to output
43      *outValue = aux;
44    }

46  protected:
47    unsigned int dataSize; // Total number of elements
48  };

50  // KernelWrapper object instantiation
51  // Number of elements per partition
52  unsigned int workSize = numberElems/numDivisions;
53  // Input/output argument info
54  std::vector<std::shared_ptr<IWorkData>> inDataInfo(3);
55  inDataInfo[0] = std::shared_ptr<IWorkData> (new BufferData<float>(workSize));
56  vector<std::shared_ptr<IWorkData>> outDataInfo(1);
57  outDataInfo[0] = std::shared_ptr<IWorkData> (new BufferData<float>(workSize));
58  std::unique_ptr<IExecutable> kernel (new KernelWrapper(KERNELFILE,
        KERNELFUNCNAME, inDataInfo, outDataInfo, workSize));
```

```
59 // Instantiate a MyMapReduce base skeleton, with optional numMOS parameter
60 MapReduce *mr = new MyMapReduce(kernel, numberElems, numDivisions, numMOS);
```

Before implementing the split function, the developer must take into consideration that the executions are applied to partitions of the original input data-set. Therefore, the *KernelWrapper*'s data-type arguments should be parametrized in a manner that defines a sub-workspace. That is, the work-load and memory spaces used in the computations should define an execution that processes a data-partition, instead of a full data-set. For instance, the parametrization done in the example sets the size of the argument buffer as worksize $= \frac{\text{numberElems}}{\text{numDivisions}}$ (line 52), that is to say, the size of a partition (lines 54 to 57). This value also defines the unidimensional work-load. Note that, since the work-load is unidimensional, it can be passed as an integer to the *KernelWrapper* (line 58). The split function works well with pointers and does not require memory transfers. This function (lines 19 to 26) uses an input vector that contains the application-defined input data (input), plus the partitions matrix as a place holder for the partitions. The latter are structured as follows: the first dimension represents each partition, while the second represents the input arguments for each partition execution. In the example, the MyMapReduce divides the input by setting the appropriate memory offset in the second dimension of partitions for each partition, via pointer arithmetic (lines 23 to 25).

In turn, the merge function (lines 29 to 48) is inversely parametrized. It reduces the data taken from the results matrix (lines 34 to 41) into one or more values. The latter are consequently saved to memory, pointed out by references taken from the output vector (lines 32 and 43). The memory references stored in output are provided by the application when prompting skeleton executions. Again, this function works with pointer arithmetic in order to avoid heavy memory copies.

Since the *MapReduce* skeleton provides overlap it may be parametrized alike a *Stream*, in Listing 3.1. In contrast, the numDivisions parameter is mandatory (line 9), as it represents the number of partitions that the developer intends to employ.

Nesting

Listing 3.5 gives an example of Marrow's skeleton nesting. The code snippet shows a three-staged image filter pipeline construction, that introduces overlap to its computations. In detail, after the *KernelWrappers* are appropriately initialized (alike in Listing 3.1), the *Pipeline* $p1$ is instantiated with the first two kernels - representing the first two stages. Then, *Pipeline* $p2$ is created and parametrized with $p1$ along with the last *KernelWrapper*. Ultimately, the *Stream* s is instantiated with $p2$. This scheme creates a composition three represented by $s1(p2(p1))$, in which the kernels associated with the innermost skeleton are computed first. As shown, Marrow's skeletons do not distinguish *KernelWrappers* from nestable skeletons, thus standardizing the nesting mechanism.

Listing 3.5: Nesting exemplification

```
1 // ... instantiate KernelWrappers
```

```
2  // instantiate inner skeletons
3  unique_ptr<IExecutable> p1 (new Pipeline(gaussKernel, solariseKernel));
4  unique_ptr<IExecutable> p2 (new Pipeline(p1, mirrorKernel));
5  // instantiate root skeleton
6  Stream *s = new Stream(p2);
```

Prompting Skeleton Executions

After the composition tree is constructed the application may requisite skeleton executions on its root node. To demonstrate this functionality (stage 2) we point to Listing 3.6. It displays a code snippet that builds upon the example presented in 3.1, using that particular composition tree, along with the same kernel. It goes without saying that the kernel data-type arguments defined at initialization time are intimately correlated with the consequent parametrization of an execution request. Not only do the arguments must match, in type and size, but also they should be set by the same order as the kernel arguments. In particular, the memory references assigned to the skeleton execution are sent in two void* vectors (lines 6 to 13). These define the memory positions from which, and to where, the skeletons perform data-transfers. Logically these should point to the correct input/output data (lines 2 and 3).

Subsequently the application requests a skeleton execution (line 15), and receives the corresponding future. From this point onward it can perform additional computations (line 17), or wait for the execution's completion (line 20). Note that the application is responsible for managing the returned future objects, namely their deallocation (line 21).

Listing 3.6: Execution example

```
1  // ... Previous variables
2  unsigned int *inputImg, *outputImg // Image input and output buffers
3  unsigned int factor; // Internal kernel value
4  IFuture* future; // Future placeholder

6  /***** STAGE 2: Prompting of Skeleton Executions *****/
7  std::vector<void*> inputValues(2); // Input memory references placeholder
8  std::vector<void*> outputValues(1); // Output memory references placeholder

10 // Defining the input/output parameters
11 inputValues[0] = inputImg;
12 inputValues[1] = &factor;
13 outputValues[0] = outputImg;
14 // Prompting a execution request
15 future = s->write(inputValues, outputValues);

17 // ... Other application computations here

19 // Waiting for the results
20 future->wait();
21 delete future;
```

Overlap to a Single Input Data-set

In the previous example the *Stream* skeleton is able to apply overlap between concurrent filter applications. That is, it may concurrently orchestrate numMOS instances of the Gaussian Noise kernel, allowing the concurrent processing of numMOS input images. Yet, if the kernel is applicable to a sub-set of the input data, for instance a slice of the image, the application may apply a finer grained overlap (on a single data-set). This is demonstrated in Listing 3.7, an example based on Listing 3.6.

Supporting such an execution scheme implies modifying the *KernelWrapper* instantiation to match a sub data-set execution. This requires redefining the kernel work-load, in addition to changing the sizes of the kernel argument data-types. To quantify these assertions consider a 1024×1024 input image that is to be divided into four segments by its height. Thus, rather than a imgWidth $\times$ imgHeight NDRange index space, the latter should be delimited by imgWidth $\times$ deltaY (lines 6 to 10), where imgWidth $=$ imgHeight $= 1024$ and deltaY $= \frac{\text{imgHeight}}{4}$. In turn, the buffer memory objects must be re-dimensioned to imgWidth $\times$ deltaY (line 15). The skeleton definition remains unaltered, as the nesting mechanism is not affected.

When preparing the data-sets for skeleton processing, dividing the input data can be accomplished by simply performing the appropriate pointer arithmetic calculations. In the example, an offset is calculated prior to each segment execution request, and used to determine the beginning of the segment on both input/output application buffers (lines 31 to 35). Logically, every execution request returns a future object. The next step depends on the application's internal logic. The application might carry out additional computations, or wait for the completion of the skeleton executions (line 44). Remember that the application must guarantee that the input/output memory references are valid while the application request, to which they are associated, is not completed.

Listing 3.7: Execution example

```
1  // ... previous variables
2  unsigned int deltaY; // height of each segment
3  unsigned int numSegments; // Number of data segments
4  unsigned int factors[numSegments]; // internal kernel values

6  /****** STAGE 1: Skeleton Initialization (modified) ******/
7  // Number of work-items in each dimension
8  std::vector<unsigned int> globalWorkSize(2);
9  globalWorkSize[0] = imgWidth; // x-dimension
10 globalWorkSize[1] = deltaY; // y-dimension
11 // ... Local work size definition

13 // Instantiate kernel input argument info
14 std::shared_ptr<IWorkData> bufferInfo(new BufferData<cl_uchar4>(imgWidth*deltaY
       ));
15 std::vector<shared_ptr<IWorkData>> gaussInputData(2);
16 gaussInputData[0] = bufferInfo;
```

```
17 gaussInputData[1] = std::shared_ptr<IWorkData> (new SingletonData<int>());

19 // Instantiate kernel output argument info
20 std::vector<std::shared_ptr<IWorkData>> gaussOutputData(1);
21 gaussOutputData[0] = bufferInfo;
22 // ... Instantiate KernelWrapper

24 /***** STAGE 2: Prompting of Skeleton Executions *****/
25 IFuture* futures[numSegments]; // Future placeholder
26 std::vector<void*> inputValues(2); // Input memory references placeholder
27 std::vector<void*> outputValues(1); // Output memory references placeholder

29 // Aplication of overlap to a single data-set
30 for(int y = 0, i = 0; i < numSegments; i++, y += deltaY){
31   int offset = sizeof(cl_uchar4)*imgWidth*y; // Start of each segment, per
         iteration
32   // Setting input/output memory references
33   inputValues[0] = inputImg + offset;
34   inputValues[1] = &(factors[i]);
35   outputValues[0] = outputImg + offset;
36   // Prompting a execution request
37   futures[i] = s->write(inputValues, outputValues);
38 }

40 //.. Other application computations here

42 for(int i = 0; i < numSegments; i++){
43   // Waiting for the results
44   futures[i]->wait();
45   delete futures[i];
46 }
```

4

Architecture and Implementation

In this chapter we present a detailed view of our proposal's architecture, in addition to explaining the implementation behind Marrow and its skeletons. We start by introducing Marrow's software stack (Section 4.1). Afterwards, we elaborate on the implementation of Marrow's functionalities (Section 4.2). Lastly, we explore another implementation strategy that was attempted, but did not meet the expectations (Section4.3).

4.1 Architecture

Our library's software stack, illustrated in Figure 4.1, is divided into four layers: *User Applications*, *Skeleton Library*, *Runtime*, and *OpenCL Enabled Device*. The communication between layers is always processed downwards, towards the computational device (fundamentally GPUs), and is achieved via well defined APIs. The ASkF was built on top of the OpenCL language, used in both *Skeleton Library* and *Runtime* layers, yet at different levels. That is, OpenCL functionalities used in the *Runtime* layer are not used in the *Skeleton Library*, and vice versa.

Applications

This layer represents the C++ applications that use our skeleton constructs. These applications are completely oblivious to the underlying OpenCL runtime management, and issue OpenCL computations through one or more skeletons, from the skeleton library layer.

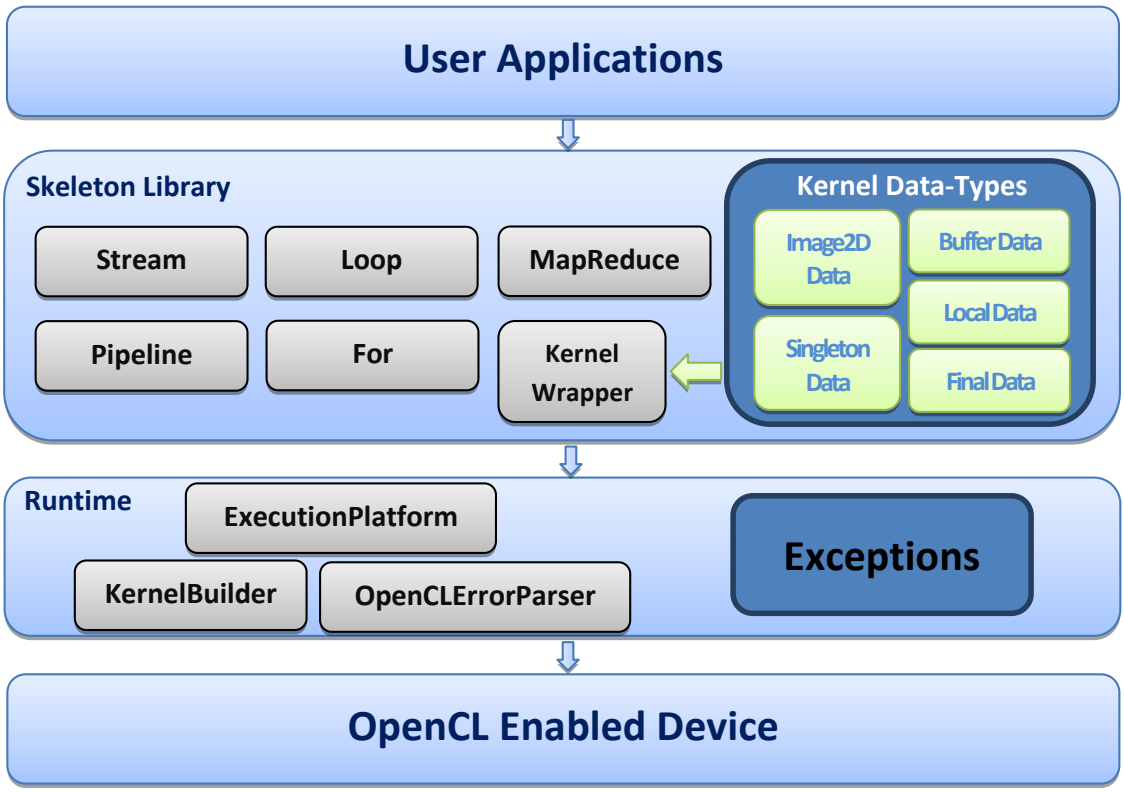


Figure 4.1: The Marrow software stack

Skeleton Library

The skeleton library is the largest layer of Marrow's software stack. It not only holds the provided skeletons, but also, the objects that define and parametrize an executable object (e.g., *KernelWrappers*, data-types). Taking as an example the Gaussian Noise kernel of Listing 3.1, the *KernelWrapper* must inform the *Stream* that the kernel's arguments are two buffers of uchar4 elements (input/output), both of size sizeof(uchar4) $\times$ uiImageWidth $\times$ deltaY. This information is contained in *IWorkData* objects utilized to instantiate the *KernelWrapper*. Thus, the *IWorkdata* objects are shared among the nodes of the composition tree, all the way up to the root. These kernel data-type objects are defined as follows:

- **BufferData** – Defines an argument as being a buffer memory object of type T and size N, where $N > 0$. The programmer defines T using the C++ template mechanism, and parametrizes the object with an integer that reflects the number of values (N) of type T that the buffer may hold.

- **SingletonData** – Defines an argument of type T as a single value that may vary between executions, where T is defined via the C++ template mechanism. Upon execution request, a memory reference to this value is passed to the root node, along with the other input/output references.

- **FinalData** – Equivalent to *SingletonData*. However, it is constant over every execution and, as such, the corresponding data value is defined when the object is instantiated.

- **LocalData** – This data-type behaves similarly to *BufferData*. However, instead of defining a buffer memory object, this data-type prompts the *KernelWrapper* to pre-allocate device local memory, for subsequent kernel executions.

- **Image2DData** – Defines a 2D image memory object of width W and height H, with $W, H > 0$. The instantiation process requires the definition of both W and H, along with the image channel data type and image channel order. At the time of writing, the supported channel data type and order were, respectively, UNORM_INT8, and RGBA. The latter are exported by *IWorkData*.

Runtime

The runtime layer is composed of behavioural modules that aggregate and export base OpenCL functionalities, used recurrently in the upper layer. These base functionalities can be grouped into three categories: resource allocation/management, kernel creation, and error parsing/exceptions. The modules that offer these functionalities are consequently expatiated.

The nesting mechanism requests that every node, of the composition tree, shares a common computational environment. In turn, an OpenCL execution requires the definition of a context that holds most of the information associated to a kernel execution. Consequently, every node must be able to access the same OpenCL context, and to add information to its state. To address this issue, the runtime layer offers an object that represents an OpenCL context, named *ExecutionPlatform*. The latter is shared between every node, and can be used to allocate OpenCL resources on demand (e.g., command-queues, memory objects). Furthermore, it is able to query information about the device associated to the context, useful to ascertain support to a particular functionality (e.g., device global/local memory sizes, max number of kernel dimensions, max number of work-item per dimension). When the *ExecutionPlatform* is released, it frees all the OpenCL resources that it had previously allocated upon request, thus acting as a resource manager. In this way the node's memory management is clearly simplified, at least regarding OpenCL resources. Note that, there is only one *ExecutionPlatform* per composition tree, whose allocation is responsibility of the root node.

Creating an OpenCL kernel requires performing a series of well defined constant steps, that can be combined into one base functionality. Accordingly, the runtime layer provides a *KernelBuilder* object that manages all the stages related to the kernel creation process. The *KernelBuilder* is able to generate a kernel object from a specific set of runtime values. This set includes: a kernel file path, a kernel function name, a context, and an OpenCL device (associated to the context). The kernels are either built after compiling

the source code, or built from pre-compiled binaries. At runtime, the binaries are created after the kernel is compiled for the first time, or updated after the kernel is modified. Moreover, they are named after the kernel file plus an added *.bin* extension, and stored next to the original source. For example, kernel *k.cl* would render a binary file named *k.cl.bin*. It comes as no surprise that building a kernel from binaries, instead of from source, is much quicker, and as a result preferable when possible.

Error parsing in OpenCL is a tedious and verbose process, since the errors are returned as integers that have to be compared with error values provided the OpenCL API (e.g., CL_INVALID_COMMAND_QUEUE, CL_INVALID_CONTEXT). In an effort to normalise error handling we have taken two measures. Firstly, we have identified fatal errors that can be triggered by the programmer, and have associated them to an exception. For instance, if the given kernel file does not exist, or the device does not have enough memory to accommodate the execution an exception is thrown. Secondly, we have implemented a class called *OpenCLErrorParser* that is able to parse integer errors to representative strings, facilitating error identification. Together, these tools make for a strong error handling system, not only internally but also for the developer.

OpenCL Enabled Device:

The lowest layer is naturally the execution platform that runs the parallel computations. Although Marrow is compatible with any OpenCL enabled device, its rationale is centred on the GPU architecture. However, since we use OpenCL the code produced by our ASkF is portable to other OpenCL compatible parallel architectures, like multi-core CPUs. Note that overlap between communication and computation is mostly advantageous to a GPU, and may not be of great use in other parallel architectures.

4.2 Implementation

This section presents the implementation of the major functionalities provided by Marrow, in particular: the execution model, the nesting mechanism, and the skeletons.

4.2.1 Execution Model Specificities

The separation between application and skeleton computations is the most distinguishing property of the Marrows's execution model. Our approach to achieve an asynchronous computational model was to use a *master-worker* scheme, where the application is the *master* and each skeleton has at least one *worker*. Naturally, this model requires the use of concurrent programming constructs (e.g., threads, mutexs, conditions). Our initial approach was to use the C++11 threading facilities. However, when the implementation process began not all C++ compilers where fully compatible with C++11. As such, we opted to use a popular non-standard C++ multi-platform library that offers its own

threading facilities, named Boost C++ Libraries[1]. As a result, Marrow's source code, plus its resulting applications, are portable and platform-independent, requiring only that the underlying platform supports both OpenCL and Boost C++. On the other hand, if need be Boost's threading facilities are completely interchangeable with, and equivalent to, C++11's, even sharing the same names between homologous constructs.

Our execution model is, logically, common to all the provided skeletons. Ergo, all skeletons have to transparently provide an equivalent *master-worker* computational scheme. To accomplish this requisite all skeletons extend an abstract class named ISkeleton. This class offers functionalities that are common to every skeleton, yet leaves some core behaviours to be implemented by each of these. The class' definition is illustrated in Listing 4.1.

Listing 4.1: ISkeleton class member function definition

```
1  class ISkeleton {
2  public:
3    /**
4     * Issues a skeleton execution that uses as input the references from
          inputData, and writes the results to the references at outputData.
5     * This method is asynchronous.
6     * @param inputData - Vector containing the input data memory references.
7     * @param outputData - Vector containing the target memory for the results.
8     * @return  A Future object associated to the execution. When the latter is
          completed the Future is notified.
9     */
10   virtual IFuture* write(const std::vector<void*> &inputData, const std::vector
         <void*> &outputData);
11 protected:
12   /**
13    * Used to adapt the execution behaviour to each skeleton. It is the main
          function of the execution process since it prompts the hole skeleton
          execution.
14    * It is where the writes/reads to/from device memory are performed, and
          where sub-skeleton execution is requested.
15    * @param inputData - The execution request, that contains input/output
          memory references, plus the Future object associated to the execution.
16    */
17   virtual void executeSkel(requestData &inputData) = 0;
18   /**
19    * Allows each skeleton to have its own OpenCL initialization process, to
          allocate the necessary OpenCL resources.
20    */
21   virtual void initOpenCL() = 0;
22   /** OTHER PROTECTED FUNCTIONS **/
23   /**           ...             **/
24 };
```

[1] http://www.boost.org/

To issue a skeleton execution, the *master* calls the write function with the appropriate parametrization. This function places a skeleton execution request in a FIFO request-queue (a C++ STL list). An execution request comprises memory references provided by the *master*, along with a reference to a future (not to be confused with C++11 future). Storing an execution request triggers pending workers, prompting one of them to process the request. Before finishing, the write function returns a reference to the future object.

When a *worker* takes a request from the queue, it passes it to the executeSkel pure virtual function. The latter is responsible for performing the computations according to the skeletons involved in the execution, which may involve issuing write/read operations, and also prompting sub-skeleton executions. Subsequently, when execution is completed the *worker* tries to fetch another request. This cycle continues until the *master* destroys the root node. Since all nodes of the composition three share the same computational environment, it is logical to consider that only one of them needs to manage the requests. Naturally, such a responsibility befalls upon the root node.

In turn, the initOpenCL pure virtual function is used to create the computational environment used throughout the tree, as well as other OpenCL resources specific to the root's requirements. It is where, for instance, the *ExecutionPlatform* object and/or the required memory objects are allocated. Given that non-root nodes do not create their own computational environment, this function is only called by the root node.

4.2.2 Nesting

Supporting a nesting mechanism for OpenCL based skeletons poses several challenges. First of all, we had to define a method that would enable a skeleton to share its computational environment with others. Moreover, we had to determine what information about the execution pattern of a nested skeleton is useful to the root node. Lastly, the solution had to permit a nested skeleton to export its execution, such that it may be prompted by its ancestor.

Our approach was to define a nesting interface class called IExecutable, depicted in Listing 4.2. The class offers informative functions, used by other skeletons to query executional requirements, in addition to two executional functions, namely: initExecutable - to initialize the node with the shared computational environment, and execute - exports the node's computations in a well defined manner.

Listing 4.2: IExecutable class member function definition

```
1 class IExecutable {
2 public:
3   /*###################### Executional Functions ############################*/
4   /**
5    * Initializes this executable according to its ancestor's computational
         environment.
6    * After this step the executable is fully functional.
7    * @param executionContext - Shared computational environment.
```

```
8       * @param memCount - Number of memory object sets to be allocated.
9       */
10      virtual void initExecutable(std::shared_ptr<IPlatformContext>
            executionContext, unsigned int memCount) = 0;
11      /**
12       * Method that exports the execution behaviour offered by the executable.
13       * It issues its execution to the received command-queue, using the pre-
             inicialized input/output memory objects.
14       * It synchronizes  its computations, with the received waitEvent, only
             executing after the latter is deemed as completed.
15       * Lastly, if the executable performs read operations, they should be
             targeted at the values stored in resultMem.
16       * This may remove unnecessary read operations, performed by the root.
17       * @param executionQueue - The used command-queue.
18       * @param inputData - The memory objects that hold the input data.
19       * @param singletonInputValues - Singleton values to be passed onto the
             kernels.
20       * @param outputData - The memory objects where the results should be stored.
21       * @param waitEvent - Event used to synchronize the execution. The latter may
              only start after the first is completed.
22       * @param resultMem - If the executable performs read operations, they should
              be performed to these memory references.
23       * @return  An event associated to this object's execution.
24       */
25      virtual cl_event execute(cl_command_queue executionQueue, std::vector<cl_mem>
             &inputData, std::vector<void*> &singletonInputValues, std::vector<cl_mem
            > &outputData, cl_event waitEvent, std::vector<void*> &resultMem) = 0;

27      /*###################### Informative Functions ############################*/
28      /**
29       * @return True if the objects has been initialized, or false otherwise.
30       */
31      virtual bool isInitialized() = 0;
32      /**
33       * @return A vector of IWorkData objects associated to the input arguments.
34       */
35      virtual const std::vector<std::shared_ptr<IWorkData>> getInputDataInfo() = 0;
36      /**
37       * @return The number of input kernel arguments.
38       */
39      virtual unsigned int getNumInputEntries() = 0;
40      /**
41       * @return A vector of IWorkData objects associated to the output arguments.
42       */
43      virtual const std::vector<std::shared_ptr<IWorkData>> getOutputDataInfo() =
            0;
44      /**
45       * @return The number of output kernel arguments.
46       */
47      virtual unsigned int getNumOutputEntries() = 0;
```

```
48    /**
49     * @return The amount of global memory required to carry out the execution.
50     */
51    virtual unsigned long requiredGlobalMem() = 0;
52    /**
53     * @return The amount of local memory required to carry out the execution.
54     */
55    virtual unsigned long requiredLocalMem() = 0;
56    /**
57     * @return True if OpenCL read operations are performed during execution, or
          false otherwise.
58     */
59    virtual bool readsData() = 0;
60  };
```

Since most executional requirements originate from the kernels, the *KernelWrapper* objects are the nodes that offer most information. In particular, the information about the kernel's arguments, and the amount of device memory needed. By contrast, the inner-skeletons simply relay the queries to their children, descending the tree until the leafs are reached and the information is obtained. Naturally, the inner-skeletons effectively cache this information for future use. The information taken from the leafs is used by the other nodes, specially the root, to appropriately allocate the necessary resources.

Generally, the nested nodes are initialized by their ancestors, through the initExecutable function, parametrized with the shared computational environment. After initialized, a node can be prompted by the ancestor to carry out its execution, via the execute function. Logically, the children require environmental parameters to perform said execution, namely: a command-queue, pre-initialized device memory objects, and a synchronization event value. These enable sub-nodes to carry out their computations in a synchronized manner, whilst enclosed in a shared computational environment. In the end, execute returns an execution-bound synchronization event.

4.2.3 Skeleton Implementation

Up to this point we have discussed the base skeleton implementation (ISkeleton class), in addition to the fundamentals behind nestable skeletons (IExecutable class). To conclude this section we will present particular implementation details, relative to each proposed skeleton. It will help clarify how each skeleton achieves its executional pattern.

Stream: The *Stream*'s most important functionality is the ability to apply overlap between communications and computations amidst operations associated to distinct datasets. Nevertheless, supporting this functionality is far from trivial, requiring a heavy host-side management of both input/output data and operation submission/synchronization.

First of all, each device execution works on a data-set stored as device memory objects, for input and/or output. Therefore, supporting multiple concurrent device executions implies the simultaneous coexistence of multiple data-sets in device memory. Hence, a skeleton must allocate a number of memory objects that enables it to issue operations associated to distinct data-sets, in an concurrent and independent manner. This strategy is designated as multiple buffering, or N-Buffering. For instance, lets consider a *Stream* s, and kernel k that requires two buffers for its execution (one input and one output). The configuration of s that uses k to concurrently process two data-sets at any given moment, d_1 and d_2, requires the allocation of two sets of memory objects, b_1, b_2. Totalling four memory objects, one input and one output per set. The skeleton starts by loading d_1 into one of the memory object-sets, say b_1, and subsequently prompts kernel execution k_1. As soon as k_1 begins, the *Stream* is able to start loading d_2 into the remaining memory object-set (b_2). This behaviour is applicable not only between write operations and kernel executions, but also with read operations, as previously depicted in Figure 3.2. The actual number of memory object-sets that are pre-allocated is parametrized by the developer when instantiating the *Stream*, as stated in Section 3.6.

Nevertheless, attaining overlap requires more than using a N-Buffering technique. The operations have to be issued, via OpenCL command-queues, in a fashion that allows the device to execute them in parallel. These command-queues offer two execution modes: in order, and out of order. Ideally, out of order execution would be preferable since it would allow a command-queue to schedule the operations according to the device's availability, as opposed to the order in which the operations where issued. However, we have ascertained that not every OpenCL implementation supports out of order command-queues. Consequently, our solution was to use as many in order command-queues as memory objects-sets, associating each one to a single set. Thus, to achieve overlap between independent device executions, a *Stream* uses a single set of memory objects per execution while issuing operations to its associated command-queue. This scheme can be scaled out as many times as needed, provided that the platform can supply the resources.

Albeit the previous orchestration is feasible in a single threaded execution, is it not very flexible and complicates the implementation and design of every skeleton, notably in regards to their synchronization. As such, we decided to use more than one *worker* per each *Stream*, specifically as many *workers* as pre-allocated memory object-sets. In other words we use a pool of *workers*, where each is associated to a memory object-set and its particular command-queue. This not only isolates distinct device executions, but also simplifies the skeleton design process, since a *worker* may perform blocking operations without impacting other executions.

MapReduce: A standard *MapReduce* pattern applies a function f to every element of a given data-set. This results in an output where each element is a function of its respective input counterpart. Logically, this scheme has an one to one relation between the input

and the output elements. Consequently, the sizes of the input data-sets must be equal between themselves, and also to the output ones. Considering, as an example, a kernel k that accepts two arrays (a_1, a_2) and returns another (a_3). If the sizes of the arrays are respectively N, M, P, then, so that k is a valid *MapReduce* kernel, $N = M = P$. This pre-requisite is checked upon initialization by the *MapReduce* implementation, raising an exception if it is not fulfilled.

The fundamentals behind our *MapReduce* implementation are somewhat simple. The first step resorts to the split function to divide the input data into partitions, generating N smaller data-sets, with $N > 0$. The skeleton then considers each partition as smaller data-set and issues N device executions. After the latter have been completed, the outputs are transferred to the host, in order to be combined into a single result using the merge function. The biggest particularity of our approach is that *MapReduce* issues device executions using internal skeletons. This means that *MapReduce*'s write function in turn triggers N writes of a given skeleton instance.

To offer overlap between communication and computation the *MapReduce* uses an appropriate internal skeleton, in this case a *Stream*. The latter is used as a stand-alone, or with a *Pipeline* if the a reduction executional entity is supplied. This strategy greatly simplifies the *MapReduce*'s implementation, as well as introducing overlap between device executions associated to independent data-set partitions.

Pipeline: A pipelined execution has one major pre-requisite: the output of a non-final stage must be compatible to the input of the next one. In turn, applying this notion to OpenCL implies that the output memory objects from a kernel must be successfully accepted as input to next one, and that these are enough to guarantee an execution without external feeding of data. This pre-requisite is checked at runtime by the *Pipeline* implementation, by comparing the *IWorkdata* output objects of the first stage with the input ones of the second stage. If the stages are not data-compatible, an exception is thrown.

Assuming a couple of data-compatible stages, the *Pipeline* must manage the intermediate data, that is, the data resulting of stage one that is used as input to stage two. Given that this data naturally needs to be stored in device memory, the *Pipeline* adds the appropriate memory objects to the computational environment. Accordingly, the first stage processes the input data, and stores its result in the intermediate memory. Which is then consumed by the second stage in order to yield the final result. This behaviour is irrespective of the number of stages that compose the execution due to the nesting mechanism. Since each instance of *Pipeline* only manages two stages, regardless if the latter are kernels or sub-skeletons, combining multiple *Pipelines* is simply scaling-out the problem, and does not involve a very particular implementation.

To support overlap between communication and computation the *Pipeline* must allocate the appropriate number of intermediate memory object-sets. Otherwise, if it were to allocate just enough memory to support a single execution at any given moment, it would cause a bottleneck in its section of the composition tree. The actual number of

memory object-sets is defined by the its ancestor node, assuming that the *Pipeline* is not a root node. For example, considering a composition tree with a *Stream* and a *Pipeline*, if the programmer parametrizes the root to allocate two memory object-sets, the *Pipeline* also receives this information and allocates two sets of memory objects for its intermediate data.

Loop: The *Loop*'s implementation is based on the premiss that its output is equivalent to its input (e.g., same data types, same number of elements). This is a valid assertion given that the *Loop* is used to apply an iterative computation to a data-set, while a condition is met. This implies that the output data is elementally equivalent to its input, and the reapplication of the computation is viable.

Efficiently implementing this behaviour suggest utilizing the output of a particular iteration as input to the next without performing transfers within device memory, or even to/from host memory. Consequently, we decided to make the *Loop* re-utilize the memory objects used to store the output, as input to the following iteration. This intercalation is performed before re-applying the computation, if necessary. Yet, this intercalation may end up positioning the final results in the input memory objects. When this happens, the *Loop* takes advantage of the fact that the memory objects are equivalent between input/output, to simply swaps the two C++ STL vector placeholders, an operation of constant complexity. Other tree nodes are oblivious to this change, and are not affected by it. Remember, however, one of the condition evaluation strategies causes the *Loop* to perform read operations at the end of each iteration.

The loop state environments are represented by an abstract class named ILoopState. This base class lets the developer create its own state objects, used to define the *Loop*'s isolated computational behaviour. Necessarily, the *Loop* uses a factory-like method, also implemented by the programmer, to create the derived loop state objects. The latter are allocated only once, when the *Loop* is initialized, and are reused multiple times by distinct executions. Introducing overlap to this equation entails pre-allocating more than one loop state per *Loop*, since this allows multiple concurrent and independent device executions. Otherwise, if a single loop state was used, the *Loop* might cause a bottleneck in its section of the composition three. The number of necessary loop states is determined by the developer when he or she instantiates the root node, assuming the root supports overlap.

For: This skeleton is a simple specialization of the *Loop* skeleton. We have implemented a *ForState*, whose internal logic represents a regular for cycle. That is, it has a counter (i), and a threshold (N), both initialized by the developer. Its condition is simply:

$$return\ true \text{ if } (i < N), \text{ otherwise } return\ false$$

Advancing the state is done by incrementing i, while resetting it is affecting i to its original value. As one would expect, here the createState *Loop* function allocates and returns *ForState* objects.

4.3 Additional Approaches

At the beginning of the implementation process we considered a different strategy for supporting an asynchronous execution model. This strategy included a functionality introduced in OpenCL version 1.1, the *callback* mechanism. Callbacks were intended to be use as an alternative to a multi-worker environment, removing the need for performing blocking OpenCL synchronization functions (e.g., clFinish(...), clWaitForEvents(...)). Even though, conceptually, using callbacks somewhat complicated the implementation, their use would reduce the amount of resources necessary when orchestrating multiple kernel executions. Moreover, using a base OpenCL functionality instead of using a third party library (e.g., Boost C++) to achieve the same results, improves Marrow's portability.

However, as it turned out, after introducing callbacks in the implementation of some skeletons, we have found that the callback mechanism had a fundamental flaw – performance. Following the profiling of applications on top of distinct OpenCL implementation we reached divergent results, that influenced our implementation approach. As was experimentally confirmed, on AMD's and Intel's OpenCL implementation the callback mechanism introduced negligible overheads. On the other hand, on our testing platforms – two distinct NVIDIA GPUs – the OpenCL implementation had a serious performance drawback. Regardless of the application being profiled, the time space between the point when the operation associated to the callback concluded and the actual start of the callback operation, was an average constant of 18 milliseconds. Obviously this largely hampers performance, particularity when processing smaller grain data-sets. Figure 4.2 depicts the execution times, in milliseconds, of a set of test case-studies. The latter, having an OpenCL and a Marrow callback version, were executed on three input data sets of increasing computational grain. Furthermore, the Marrow executions use a varying number of memory object-sets to apply overlap between communication and computation. We refer the description of the actual case-studies, as well as the testing methodology, to the next chapter.

The results show a very clear pattern, that is constant throughout the case-study set. To simplify the explanation consider that the first case-study processes five input data-sets before terminating, and the remaining process four. This slightly diverges from the actual execution process, but it reduces the problem to the essentials, in order to better understand what is happening. Also, regard that every OpenCL version execution takes less then 100 milliseconds to process its input data-sets, independently of grain size. The last factor is that every single skeleton execution (that processes a single data-set) issues a callback, and therefore takes on average at least 18 milliseconds to deliver the results to the application.

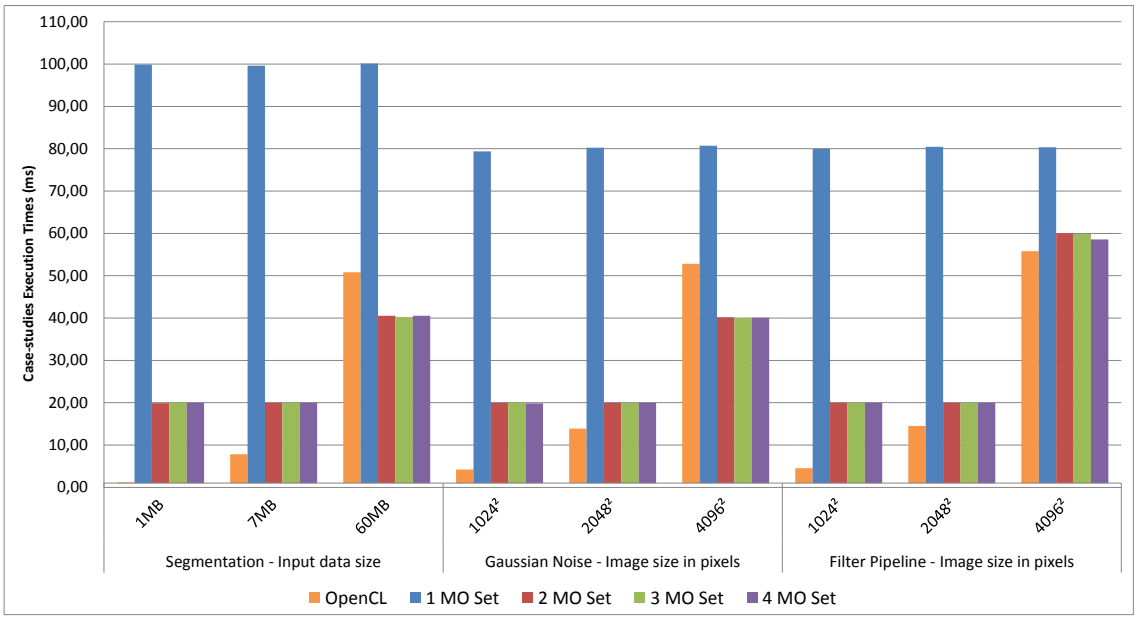


Figure 4.2: Execution times for a set of applications

Undeniably, when the Marrow versions use a single memory object-set the execution times scale dramatically. This happens because the skeletons can only process the input data-sets in a sequential order. This order of execution does not mitigate in any way the overheads associated to callbacks, resulting in an execution that takes about $18 \times (numberOfInputDataSets)$ milliseconds to complete. For instance, the last two case-studies, that execute on four input data-sets, take about 80 (or 18×4) milliseconds to complete. Using more than one memory object-sets enables the concurrent execution of distinct data-sets, i.e. enables overlap. The latter lessens the overheads that result from callbacks, though only when processing the larger data-sets do the skeletons outperform the OpenCL versions. Interestingly, the overlapped executions always take a multiple of 20 milliseconds to complete the computations. This evidences that the callback overhead is never fully mitigated, and the last callback always introduces performance penalizations.

We were not able to uncover the reasons behind the callback's misbehaviour, that clearly is OpenCL implementation-dependant. Hence we opted for a solution that proves efficient irrespective of the underlying OpenCL implementation. This solution has already been expatiated in this chapter.

5

Evaluation

This chapter presents the evaluation that was conducted to validate Marrow's design purposes. In particular, this validation aimed to assess Marrow's behaviour when compared against other technologies, of both lower and higher level. Towards this evaluation we implemented various multi-version case-studies. Each one has a minimum of two versions – one on top of Marrow and, at least, one more on top of a different library. Furthermore, this evaluation was done according to two perspectives: performance, and programming model. The performance evaluation compares the time intervals that different versions of the same case-study take to complete its execution. It was mainly focussed on quantifying the performance gains associated to overlap, relative to executions that do not apply it. In turn, the programming model evaluation aims to determine the productivity (lines of code) and complexity associated to Marrow's programming model, against other technologies.

Considering that the performance benefits attributed to the use of overlap may vary depending on the number of memory object-sets used, the Marrow versions were executed multiple times, on a varying number of sets per run. This variation took place while every other computing OpenCL parameter (e.g., NDRange index, work-group size) remained constant through every Marrow version run. On top of this, each case-study, regardless of version, was computed with three different inputs, of increasing computational grain. All the runs were performed on two systems with the following specifications:

System One (S_1):

- **CPU** – Quad-core Intel Xeon E5506 @2.13 GHz

- **Motherboard** – ASUS P6T7 WS Super Computer LGA 1366 DDRIII 7PCIE16X
- **Main Memory** – 12 GB RAM DDR-3
- **GPU** – NVIDIA Tesla C2050
- **GPU Driver Version** – 295.41
- **Operating System** – Linux Ubuntu 10.04.4 LTS (kernel 2.6.32-41)

System Two (S_2):

- **CPU** – Quad-core Intel Xeon E5506 @2.13 GHz
- **Motherboard** – ASUS P6T7 WS Super Computer LGA 1366 DDRIII 7PCIE16X
- **Main Memory** – 12 GB RAM DDR-3
- **GPU** – NVIDIA GeForce GTX 680
- **GPU Driver Version** – 295.41
- **Operating System** – Linux Ubuntu 10.04.4 LTS (kernel 2.6.32-41)

The evaluation is divided into two sections: against OpenCL (Section 5.1), and against SkePU/SkelCL (Section 5.2).

5.1 Comparison with OpenCL

This section of the evaluation compares the OpenCL language against Marrow's skeleton constructs. Since our comparison is mainly interested in evaluating the performance gains obtained from overlap, the OpenCL versions of the case-studies do not introduce it. In addition, this approach enables us to confirm if Marrow's programming model is advantageous, by comparing Marrow versions to OpenCL versions of lower complexity, i.e. that do not introduce overlap between communication and computation. We totalled five case-studies for this comparison, namely: Gauss Noise filter, Image Filter Pipeline, Segmentation, Hysteresis, and N-Body.

Case-Study 1 – Gaussian Noise Image Filter: The first case-study is the concretion of the Gaussian Noise image filter as previously presented in Section 3.6, particularly in Listings 3.1 and 3.6. The filter is applied to the images in accordance with the execution pattern depicted in Listing 3.6, that is, by dividing the input image in slices and subsequently applying the filter to each slice. Yet, in the OpenCL version the slices are processed sequentially, in contrast with the Marrow version of the case-study. As expected, to apply overlap to this scheme a *Stream* skeleton is used in the Marrow version. The kernel used in this case-study was taken, and adapted, from AMD's OpenCL samples[1].

[1] http://developer.amd.com/sdks/AMDAPPSDK/samples/Pages/default.aspx

Case-Study 2 – Image Filter Pipeline: The second case-study is a pipelined application of image filters, namely Gaussian Noise, Solarise, and Mirror. These filters were selected due to fact that they produce the same results regardless of being applied to a hole input image, or segments of an image divided by its height. Alike in the previous case-study, this runtime strategy allows for an overlapped execution of a single data-set. Consequently, both versions of this case-study perform equivalently to their respective counterparts in the previous case-study, differing only by applying multiple filters in succession to each slice. Naturally, the Marrow application uses *Pipelines* nested in a *Stream* (identically to Listing 3.5).

Case-Study 3 – Segmentation: The third case-study is a tomographic image enhancing technique, named Segmentation. A tomographic image is represented by a three-dimensional space, where each element, a volumetric pixel (voxel), shows the amount of radiation absorbed in its specific sample zone. The segmentation technique helps to clearly define the frontiers, between the volumetric components. This is done by affecting the color of each voxel from its original value, to either black, white, or gray, depending on its intensity. However, the image is not processed as a whole. Instead, the computations are independently applied to non-superimposing segments of the input image, specifically to segments of size equal to one-fifth of the input image. The OpenCL version processes the segments sequentially. On the other hand, the Marrow version concurrently processes the image segments. To introduce overlap between segment executions a *Stream* is used.

Case-Study 4 – Hysteresis: The third case-study applies a tomographic image processing method, denominated as Hysteresis [CFMQRVV10]. The latter allows an iterative elimination of gray voxel sets from a tomographic image, by determining if they should be altered into white or black ones. This algorithm comprises three stages, where each builds upon the results of its predecessor to refine the grey voxel elimination. Alike in the previous case-study, the image is divided in segments, prior to applying the computations. Additionally, these segments are stage related, and therefore may not match between stages (e.g., start/end of segment, number of segments). In any case, at every given stage the computations are iteratively applied to a single segment until the results stabilize. Also, the application only advances to the next stage after the previous one has completely processed the input image. The OpenCL version processes the segments sequentially, as well as the stages. On the other hand, the Marrow version concurrently processes the segments of a single stage. The skeletons used in this case-study are *Loops* nested into *Streams*, one combination per stage. Note that, the mismatch between corresponding stage related segments prevented us from using a pipelined execution in between stages. The hysteresis technique is usually applied to tomographic images that have been previously processed with a segmentation.

Case-study	Input Size	S_1 – Exec Time (ms)	S_2 – Exec Time (ms)
Gaussian Noise (pixels)	1024^2	4.23	4.17
	2048^2	13.86	13.26
	4096^2	52.79	49.96
Filter Pipeline (pixels)	1024^2	4.52	4.43
	2048^2	14.48	13.26
	4096^2	55.82	49.96
Hysteresis (MB)	1	402.98	399.50
	8	2952.98	2899.34
	60	19742.80	19550.60
Segmentation (MB)	1	1.17	1.63
	8	7.83	7.37
	60	50.82	47.29
N-Body (particles)	1024	39.98	14.98
	2048	79.25	28.77
	4096	207.08	63.92

Table 5.1: OpenCL versions execution times in milliseconds

Case-Study 5 – N-Body: The fifth, and final, OpenCL case-study is an N-Body simulation. The latter computes a fixed number of one hundred iterations. Contrary to the previous case-studies, a N-Body input data-set can not be computed in segments due to the nature of the algorithm. Dividing the input data would affect the calculation of every particle in every iteration, ultimately leading to an erroneous result. As such, the OpenCL and Marrow versions are equivalent, in the sense that they do not introduce overlap to the computations. Consequently, only a *For* skeleton is used. This case-study may be used solemnly to verify if the Marrow version introduces noticeable performance overheads, as well as to validate the *For* skeleton. Should be noted that the N-Body kernel was also taken, and adapted, from AMD's OpenCL samples.

5.1.1 Performance Evaluation

The execution times of the OpenCL versions of the case-studies, on the targeted systems, are presented in Table 5.1. These measurements, presented in milliseconds, are relative to the time intervals required to accomplished the desired OpenCL execution, yet excluding the initialization and deallocation processes. All things considered, the latter are only done once per application run. Therefore, these are not relevant for our performance evaluation. In addition, excluding these processes enables us to perform a finer grained measurement of the performance benefits that arise from an overlapped execution. Subsequently, the respective speedup values obtained from the Marrow versions of the case-studies are presented in Figures 5.1, 5.2, 5.4, 5.3, and 5.5.

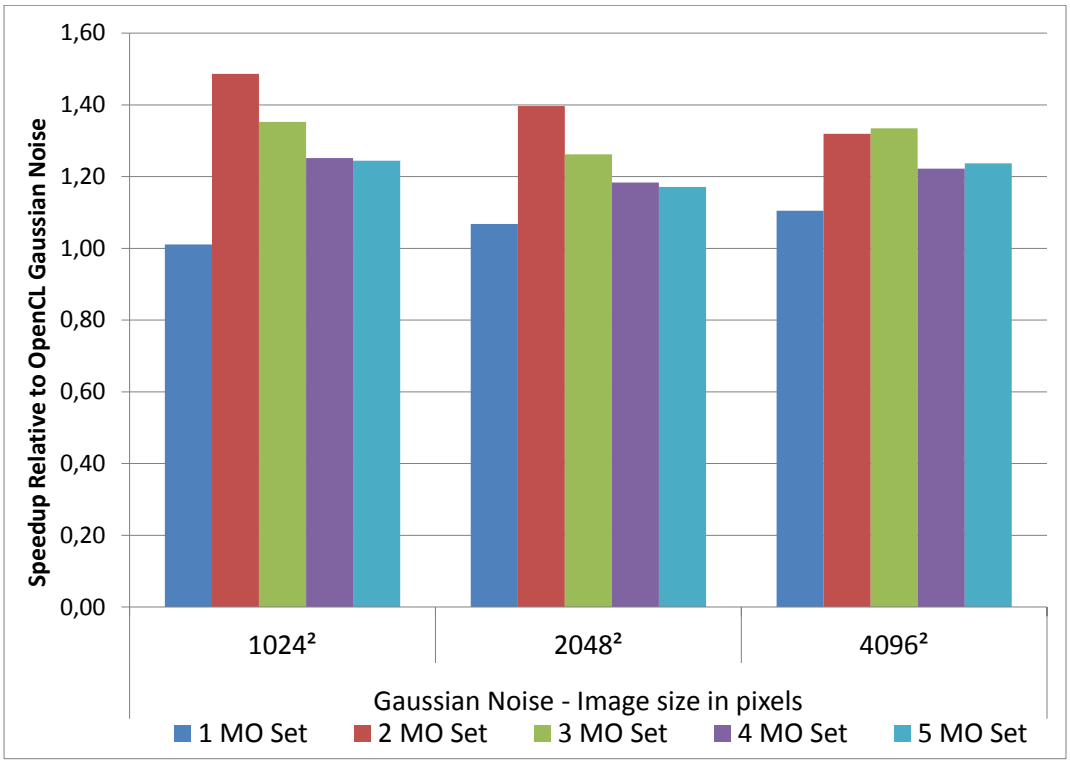
In general, every case-study benefited in terms of performance from overlap, albeit

some more significantly than others. Moreover, the resulting speedups were greater in the system S_1, although the applications were parametrized exactly in the same way on both systems. These results show that N-Buffering, used in the implementation of overlap between communication and computation, is a very parametrization and hardware sensitive technique. For instance, in the Gaussian Noise and Filter Pipeline case-studies the input images are always divided into four segments regardless of input size or system. Thus, a four-byte per pixel input image generates memory segments (chunks) of size: 1MB, 4MB, and 16MB, respectively. Clearly this is not the ideal parametrization to optimize overlap between communication and computation in these case-studies on system S_2, given the specifications of that GPU. Also, S_1's GPU is a scientific processing unit, while S_2's is aimed at video game graphics processing. Other studies on the subject of N-Buffering and GPUs [SMV10] consolidate our findings, suggesting that this behaviour is to be expected especially if the parametrization was not fine-tuned to a particular application, on a particular hardware.

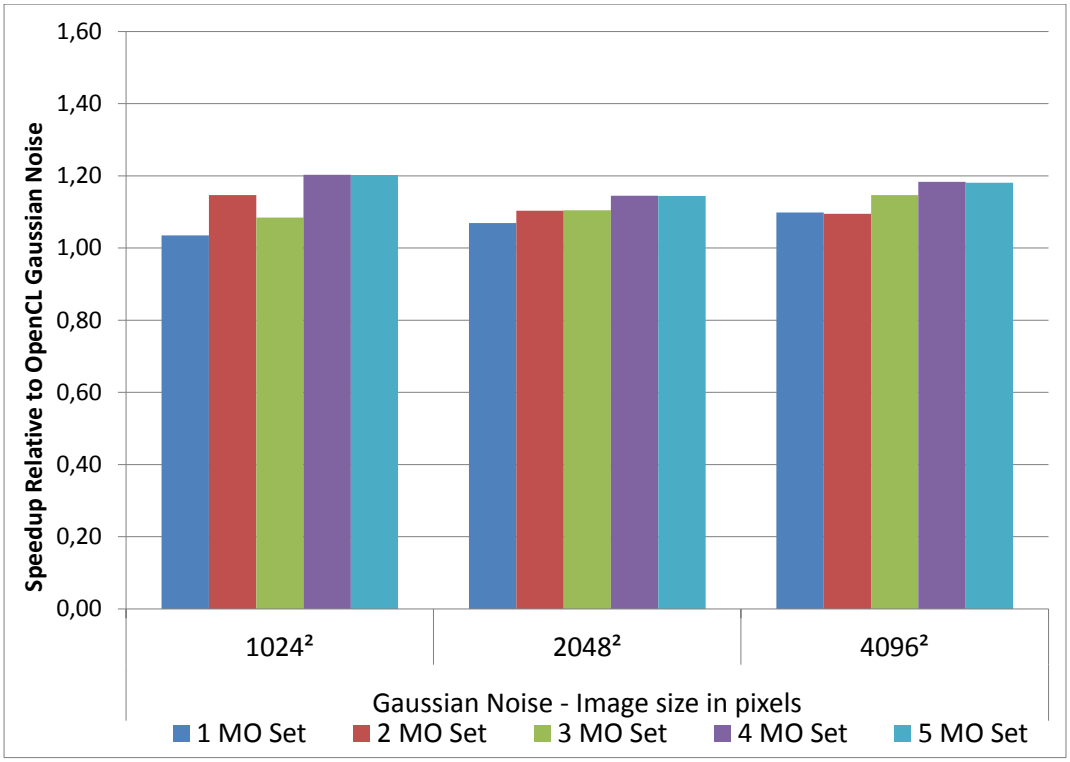
The first two case-studies (Figures 5.1 and 5.2) show a very similar behaviour. In system S_1 (Figures 5.1a and 5.2a) overlap introduces the highest speedup when using two or three memory object-sets, depending on the input data grain, and starts to decrease as more sets are used. This behaviour is expected considering that, firstly, the operation scheduling overhead increases with the number of command-queues used. Moreover, independently of the number of used memory object-sets, executing the same operations (e.g., writes, reads, kernel executions) on equivalent data-sets takes on average a constant amount of time to complete. Therefore, having a maximum speedup with two/three memory object-sets simply implies that the underlying GPU is only capable of executing in parallel, at any given moment, operations associated to those many data-sets, given their execution times and memory chunk sizes. In turn, in system S_2 (Figures 5.1b and 5.2b) the highest speedup is obtained when using around four memory object-sets. Evidently, in this parametrization scenario, S_2's GPU is better able to introduce parallelism between distinct executions than S_1's, even if this added concurrency is not directly translated into performance.

The speedup values obtained from the Marrow version of the Segmentation case-study (Figure 5.3) practically fall in line with the speedups of the first two case-studies. Once more, on system S_1 (Figure 5.3a) the best results appear when using two or three memory object-sets. Whereas, on system S_2 (Figure 5.3b) the highest speedups present themselves, in general, when using about four memory object-sets. The one megabyte execution of Figure 5.3b is the only exception, by behaving better with just two memory object-sets – another example of the unpredictability of overlap performance gains.

The Hysteresis' case-study (Figure 5.4) execution pattern differs from that of previous ones. Its computation flow dictates that after each loop iteration, which processes a single segment, the *Loop* reads the results to host memory, and subsequently evaluates them to access its continuity, a process of complexity $O(N)$, where N is the size of a segment. These two actions are computationally heavy and leave the GPU available to execute

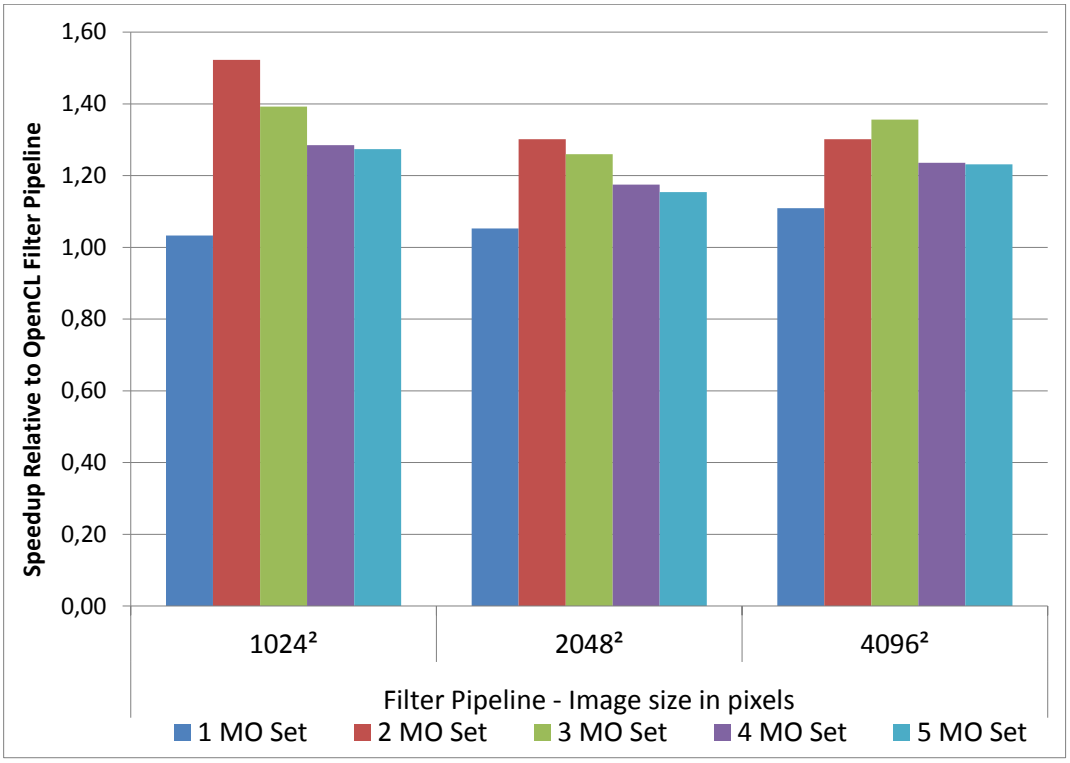


(a) Gaussian Noise speedup values, system S_1

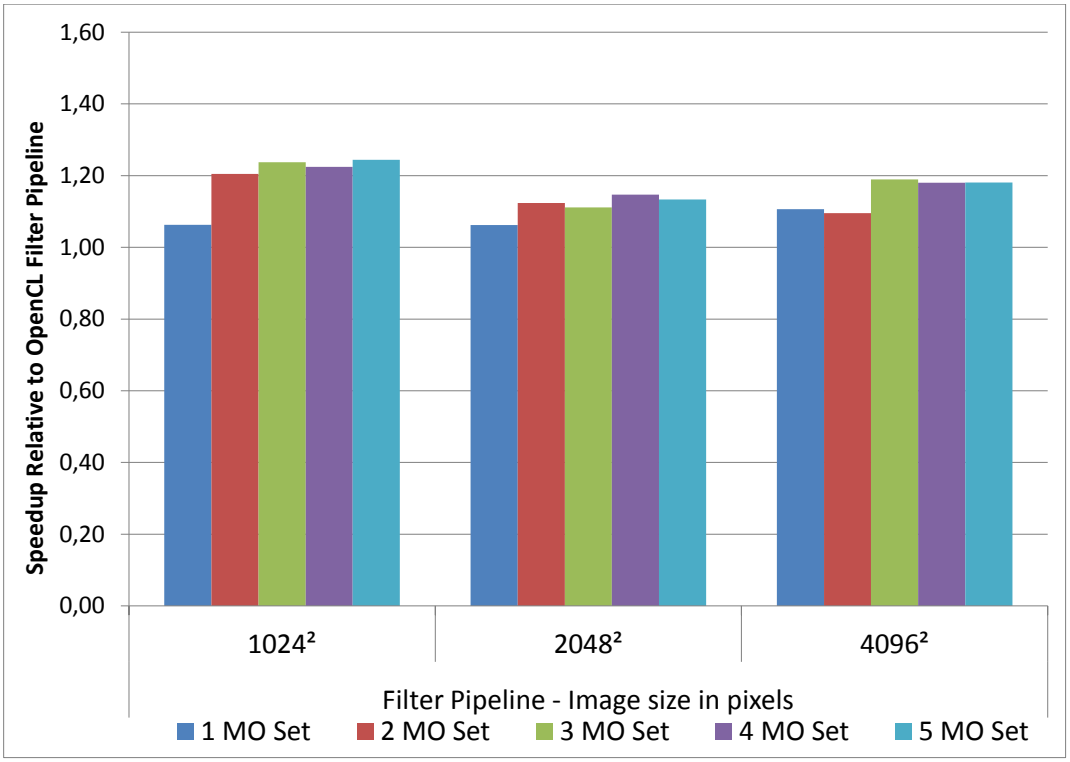


(b) Gaussian Noise speedup values, system S_2

Figure 5.1: Gaussian Noise Marrow Speedup Values

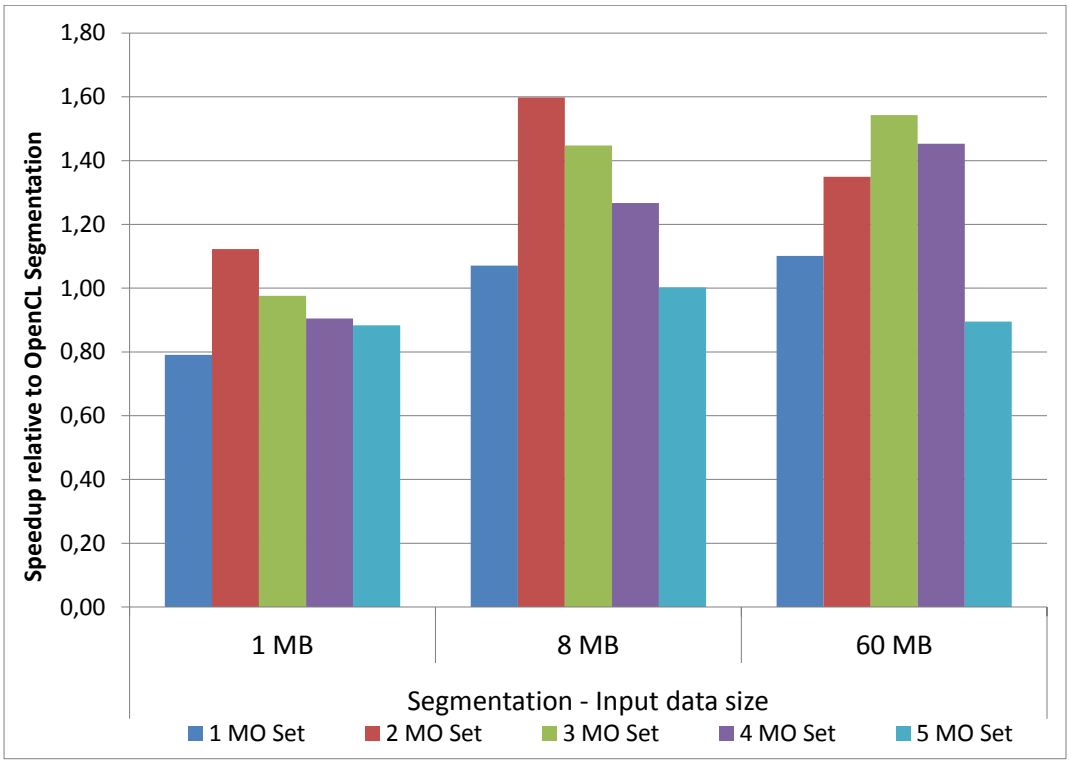


(a) Filter Pipeline speedup values, system S_1

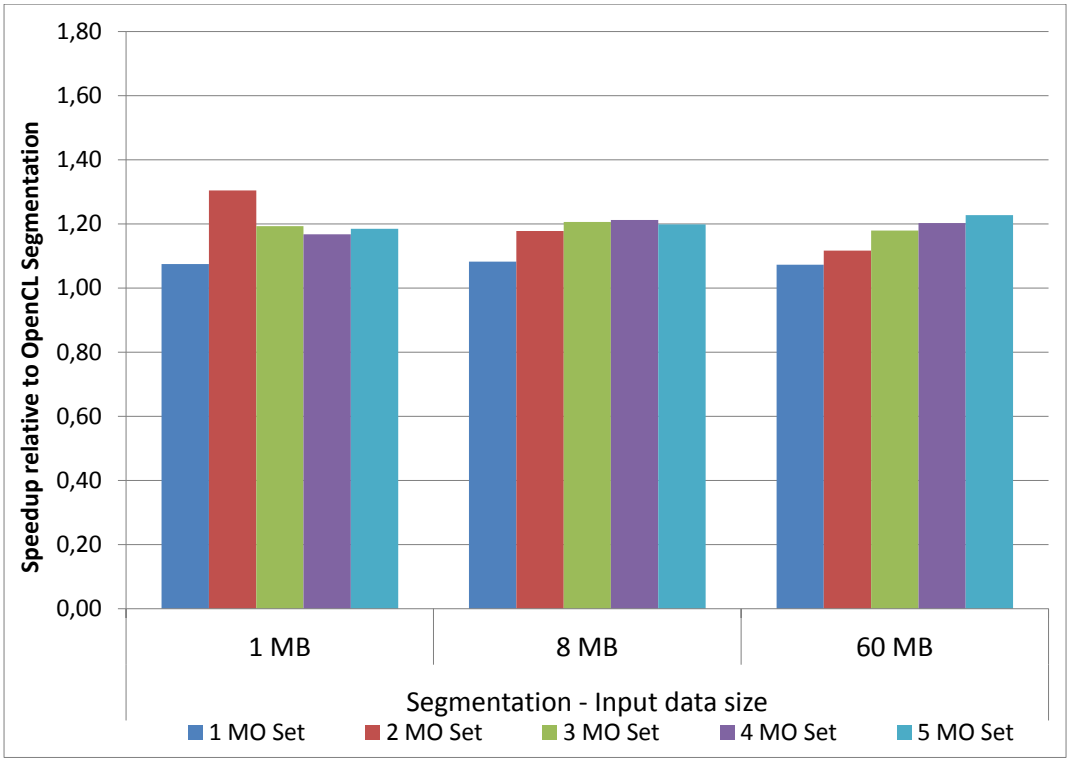


(b) Filter Pipeline speedup values, system S_2

Figure 5.2: Pipeline Marrow Speedup Values

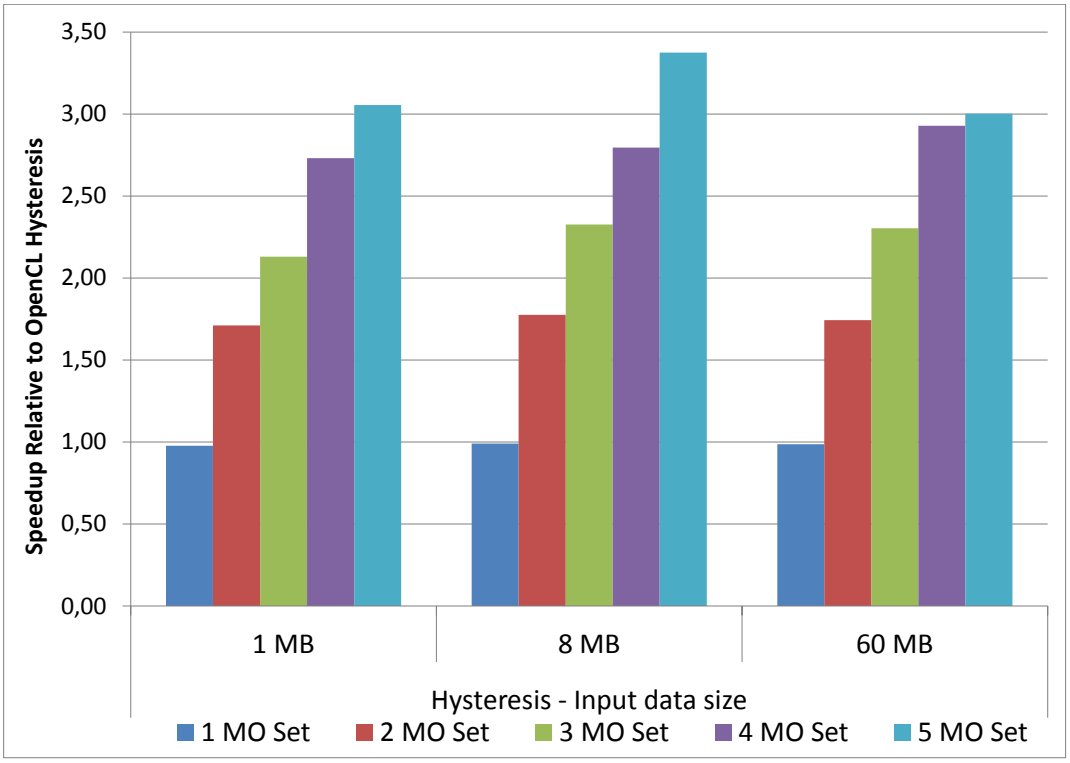


(a) Segmentation speedup values, system S_1

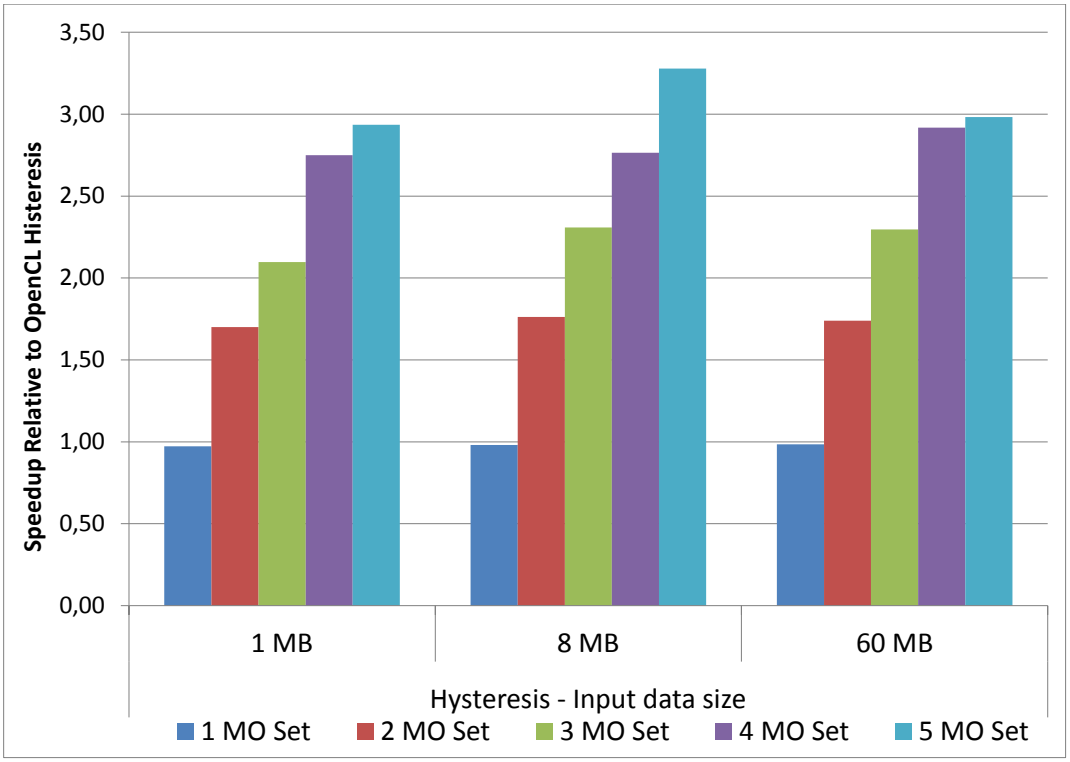


(b) Segmentation speedup values, system S_2

Figure 5.3: Segmentation Marrow Speedup values



(a) Hysteresis speedup values, system S_1



(b) Hysteresis speedup values, system S_2

Figure 5.4: Hysteresis Marrow Speedup Values

upon other data-sets. As previously stated, the application issues a maximum of five skeleton executions (one per segment) for each of the three stages. Consequently, we assert the existence of a considerable amount of applicable parallelism between segment executions. On top of that, the speedup is incremental given that each of the three stages introduces it. Ergo, the speedup systematically increases as more memory object-sets are used, maxing at the same number as of maximum segments per stage. This behaviour is consistent on both systems (Figures 5.4a and 5.4b).

The N-Body case-study (Figure 5.5) was executed with only a single set of memory objects, since it does not support an overlapped computation of a single input data-set. Still, this application allows us to observe the overhead introduced by the skeleton constructs, against a hard-coded OpenCL execution. Both systems perform equivalently, resulting in a 1.75% maximum overhead (system S_2). This value is negligible and even gets reduced as the input grain increases.

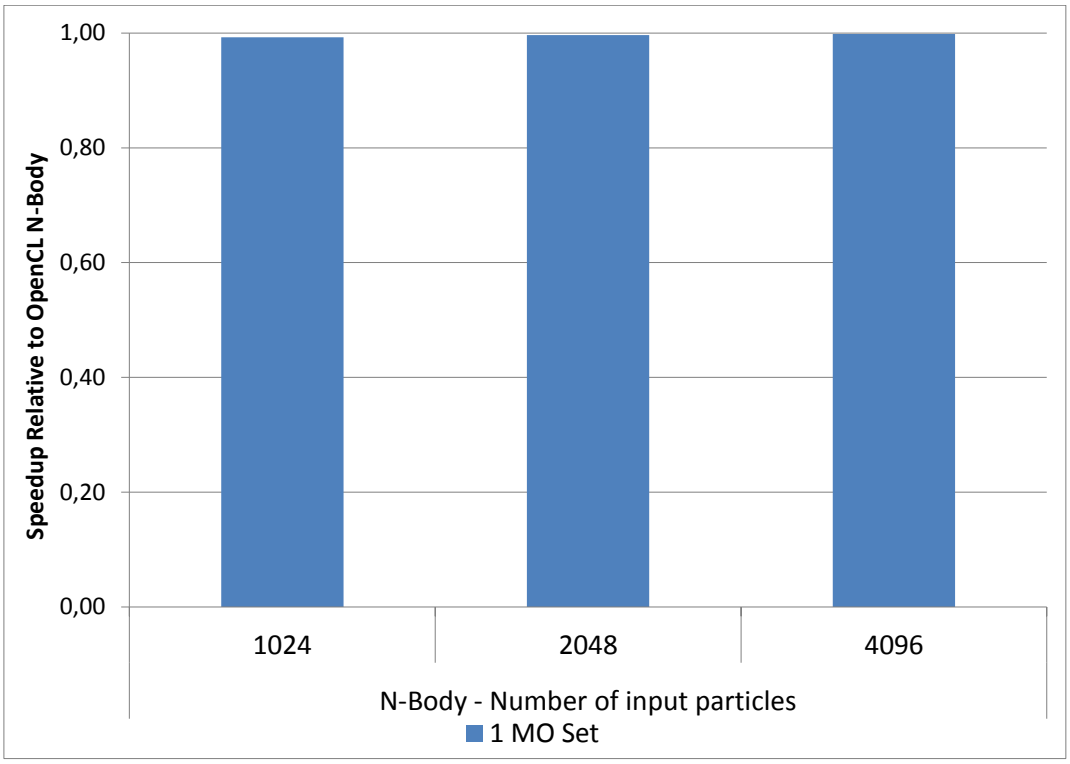
5.1.2 Programming Model Evaluation

It comes as no surprise that our programming model is simpler, and of higher-level than OpenCL's. Not only do the skeletons orchestrate the whole execution, but also introduce transparent performance increments. To quantify these judgements, Figure 5.6 presents the sizes (in number of lines of code) of three application versions: OpenCL, OpenCL with overlap, and Marrow. These values do not include: blanc lines, main function, headers, and error handling. To introduce overlap in an OpenCL application we estimated a minimum increase in seventy lines of code, adding to the design complexity which would surely grow substantially.

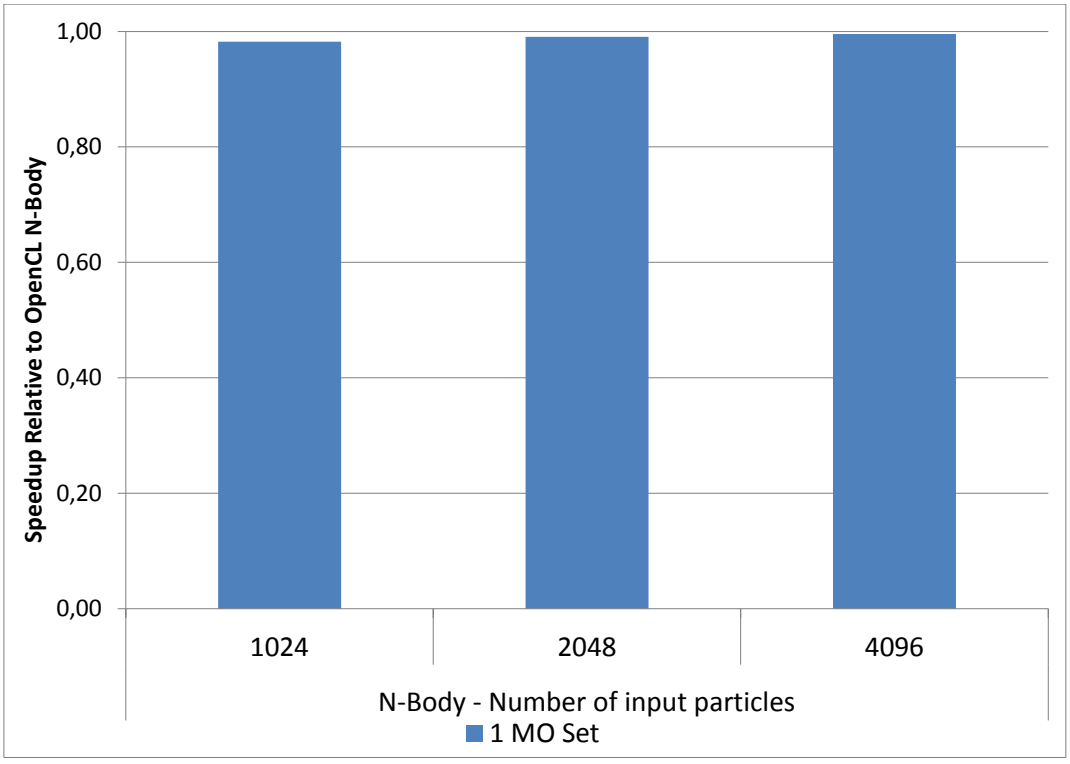
To be fair, Marrow's programming model productivity should only be expected to trump the overlapped OpenCL applications. Nevertheless, Marrow's programming model simplicity results in less code per application than even the basic OpenCL versions. The only exception seems to be the Hysteresis case-study, requiring roughly more 40% of code than the OpenCL version. This increase in program size comes as a result of two factors: skeleton initialization, and *Loop* instantiation. First of all, initializing three *Loops* nested into three *Streams* is somewhat verbose. Secondly, to use a *Loop* skeleton the developer must declare a class that derives it, and implement its inherited abstract functions (as exemplified in Listing 3.3). Joining these two factors adds a considerable amount of lines of code to the application, justifying the discrepancy between OpenCL and Marrow Hysteresis versions.

5.2 Comparison with SkePU and SkelCL

To compare Marrow to other skeleton libraries we used SkePU (version 0.6) and SkelCL as target libraries. When selecting case-studies for this comparison we sought to find applications that would, at the same time, benefit from overlap and be compatible with the



(a) N-Body speedup values, system S_1



(b) N-Body speedup values, system S_2

Figure 5.5: N-Body Marrow Speedup values

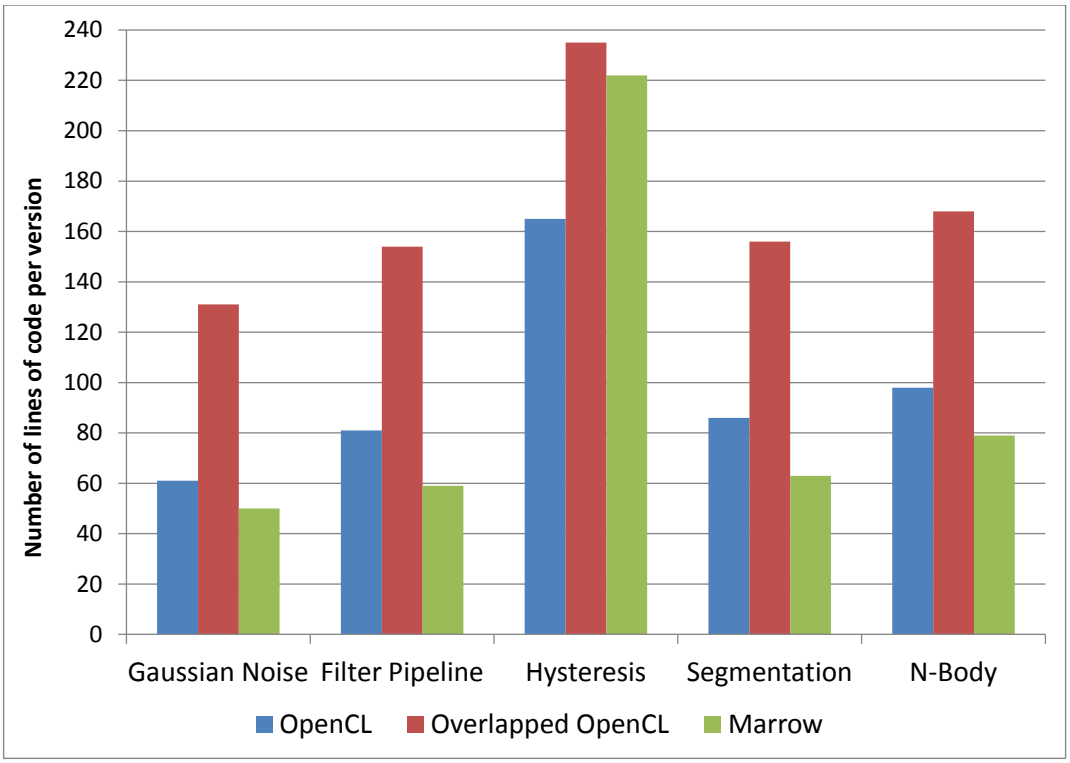


Figure 5.6: Productivity comparison between distinct application versions

target libraries. However, we were constrained by the limitations of the chosen skeleton libraries, significantly reducing our possible collection of case-studies. The resulting set consists of four applications, namely: Saxpy, Array Multiplication, Gaussian Noise Filter (only for SkelCL), and Solarise Filter.

Case-Study 1 – Saxpy: Saxpy is a common operation in computations with vector processors. It is a combination of scalar multiplication and vector addition. The application applies the Saxpy technique to two equally sized input arrays (a and b), plus a constant scalar value (α): $c[i] = \alpha a[i] + b[i]$. Subsequently, a host-side reduction is performed on the output results. The SkePU and SkelCL versions apply the computations to the entire input data via a *Map* skeleton, before reducing the results. On the contrary, the Marrow version divides the input data into M segments and uses N memory object-sets for overlap, where M, N vary between one and four. This strategy allows us to investigate two aspects: how do different block sizes (chunks) affect performance, and what impact does overlap have on this application. Since our *MapReduce* skeleton provides a split and merge functionality it is used to compute Saxpy, in the Marrow version.

Case-Study 2 – Array Multiplication: The second case-study is very similar to the first, except it performs a per element multiplication of two vectors with the same size: $c[i] = a[i] \times b[i]$, and does not reduce the results. All developed versions compute alike the respective versions in the previous case-study, with the exception of Marrow's skeleton

selection. As it stands, this case-study uses a *Stream* skeleton.

Case-Study 3 – Gaussian Noise Image Filter: The third case-study is close to the Gassian Noise filter example of Subsection 5.1. The major difference being that in the Marrow version the number of divisions of the input image varies from one to four. In its turn the SkelCL version processes the whole image at once, using for this purpose a *Map* skeleton. We were unable to implement a SkePU version given that the latter does not support trigonometric operations in its user-defined functions.

Case-Study 4 – Solarise Image Filter: The last case-study is a reimplementation of the preceding one, using a Solarise filter (used in the Image Filter Pipeline case-study). Every remaining application detail does not vary from the Gaussian Noise case-study.

5.2.1 Performance Evaluation

Table 5.2 shows the execution times, in milliseconds, for the target library versions of the case-studies, on both systems. It is not discernible at which point in the execution do the target libraries create, and initialize, the required memory objects. Consequently, we decided to include the library initialization process in the measurements, in order to make these more fair. Comparing the results of both libraries, overall execution times favour SkePU, apart from the Saxpy case-study. We concluded that this discrepancy is due to the host reduction. If the latter is removed from the measurements the two versions of Saxpy perform similarly. Lastly, the measurements for the Marrow versions were obtained from executions on a varying number of input data segments, plus memory object-sets. This approach was intended to further disclose the effect of N-Buffering on different chunks sizes, when computed by distinct GPUs.

From a general standpoint, the results indicate that overlap does not prove to be beneficial to all case-studies. This lack of all-encompassing performance gains can be attributed to three main factors:

1. **Initialization Overheads** – Logically these dilute any obtainable speedup, specially when executing upon smaller grained data-sets.

2. **Case-study Nature** – Not every application may benefit from overlap. For instance, memory-transfer bound applications tend to benefit less than computational heavy ones.

3. **Memory Block (chunk) Size** – Each GPU has its own ideal block size for a given application. These chunks are closely associated to the GPU's memory bandwidth and throughput.

It is observable that the results vary somewhat significantly from system to system, and between distinct parametrizations (e.g., chunk size, memory object-sets used). Therefore we cannot assert the existence of an outright solution for every possible execution

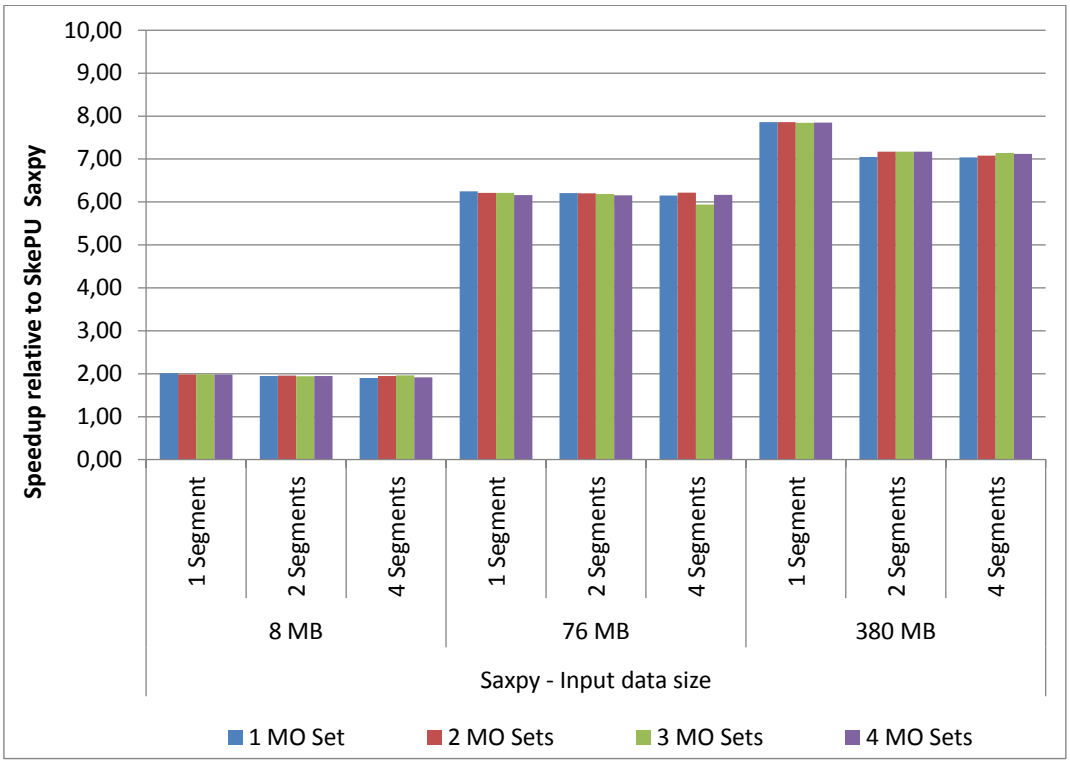
Case-study	Input Size	SkePU – Exec Time (ms)		SkelCL – Exec Time (ms)	
		S_1	S_2	S_1	S_2
Saxpy (MB)	8	145.15	160.06	135.08	122.74
	76	1455.31	1656.09	458.67	446.02
	380	7227.38	8129.86	1877.52	1862.26
Array Multiplication (MB)	8	17.87	18.87	79.34	66.82
	76	183.79	183.20	276.07	264.733
	380	868.90	873.68	1155.64	1146.13
Gaussian Noise (pixels)	1024^2	N/A	N/A	87.39	75.88
	2048^2	N/A	N/A	172.73	158.67
	4096^2	N/A	N/A	588.39	571.89
Solarise (pixels)	1024^2	9.52	10.74	68.50	57.36
	2048^2	32.05	32.35	99.79	86.29
	4096^2	159.14	160.04	289.26	274.64

Table 5.2: SkePU/SkelCL versions execution times in milliseconds

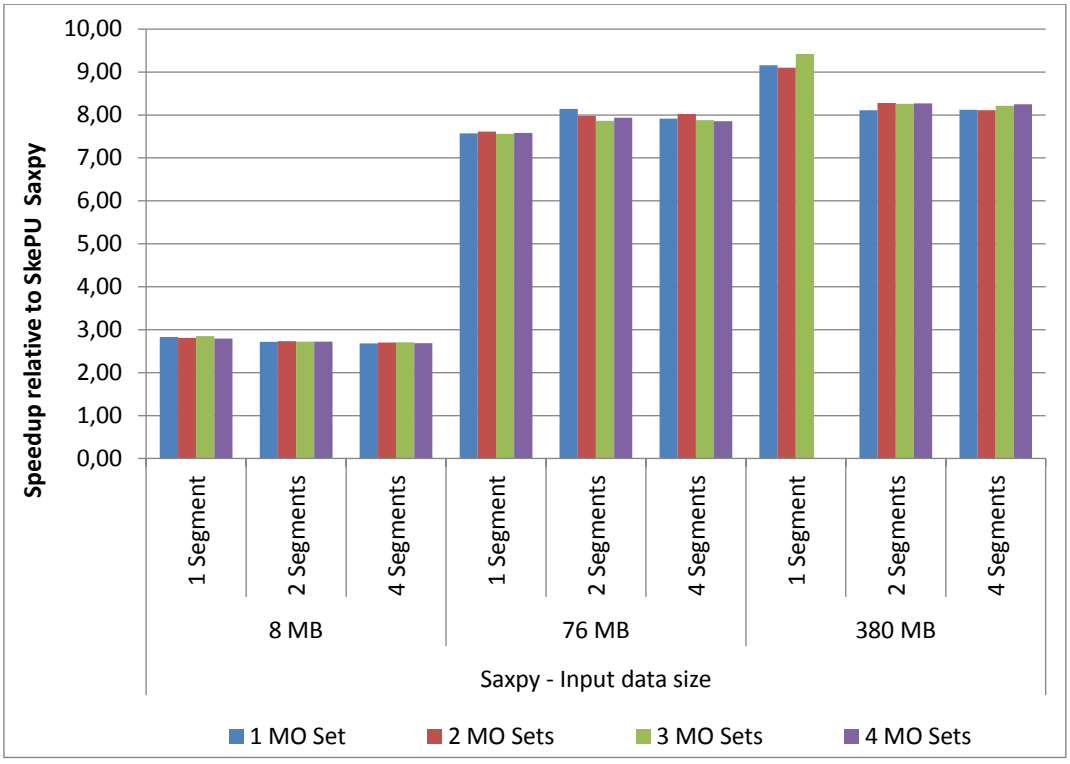
scenario. Quite the contrary, there is virtually limitless set of possible parametrization schemes, considering the number GPUs that are compatible with OpenCL (and implicitly with SkePU, SkelCL, and Marrow). This idea completes the conclusions drawn from the previous evaluation. In any event, we will subsequently expatiate the results obtained from each of the case-studies, that were selected for this comparison. The speedup values are presented in Figures 5.7 to 5.13

The Saxpy case-study (Figures 5.7 and 5.8) can be classified as a memory transfer-bound application, since it uses two input buffers and computes a trivial algebra operation. Accordingly, it is not a good candidate to benefit from overlap, as apparently dividing the input data and using multiple memory object-sets does not lead to significant, if any, performance gains. Moreover, the best performance is almost always obtained from a single segment execution. The special case is the execution on 76MB of input data on system S_2, that shows better results with two segments (33MB each), visible in Figures 5.7b and 5.8b. Nevertheless, even without benefiting from overlap, the Marrow version is the most efficient one. System S_2 (Figures 5.7b and 5.8b) yields the best speedup when compared to both target skeleton libraries.

Given the similarities between the second (Figures 5.9 and 5.10) and the first case-study, it comes as no surprise that its results are somewhat equivalent. Yet, the speedups are noticeably affected by the number of memory object-sets used, specially in system S_1 (Figures 5.9a and 5.9a). This, however, does not always imply the best performance. For example, on system S_1 (Figures 5.9a and 5.10a) the highest speedup is obtained with the single segment executions. Additionally, Marrow performs better than SkelCL but only bests SkePU when the input data grain increases. Furthermore, system S_2 (Figures 5.9b and 5.10b) surpassed S_1 in speedup.

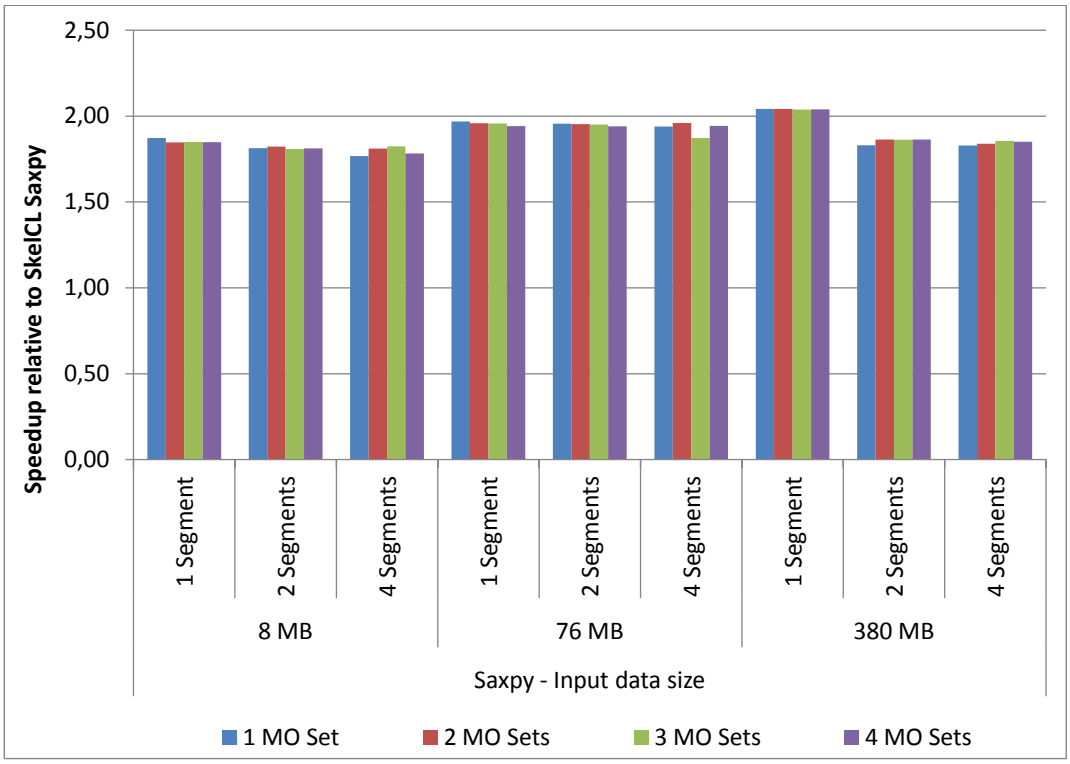


(a) Saxpy SkePU speedup values, system S_1

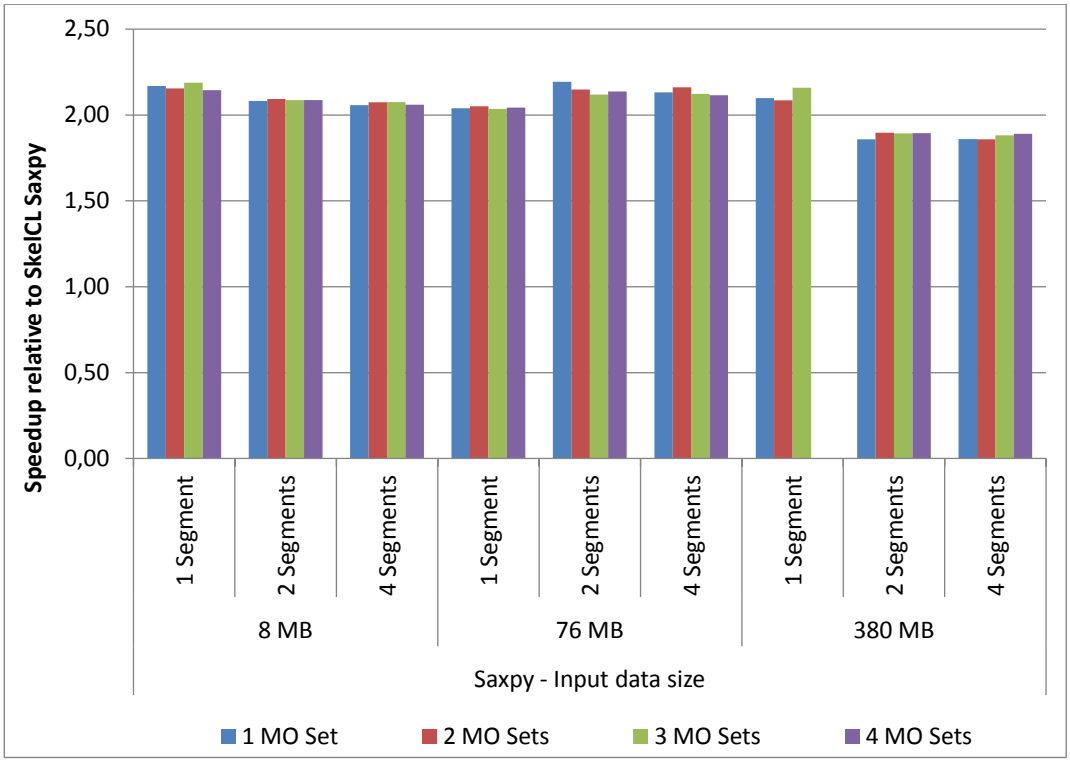


(b) Saxpy SkePU speedup values, system S_2

Figure 5.7: Saxpy SkePU Speedup Values

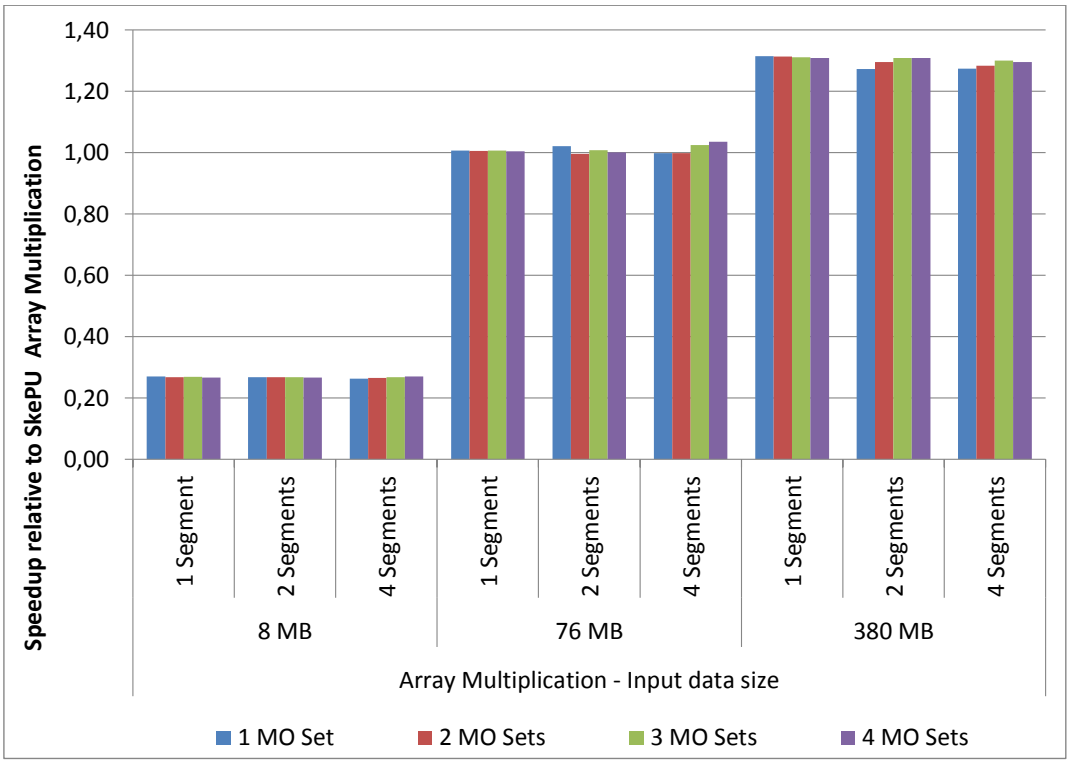


(a) Saxpy SkelCL speedup values, system S_1

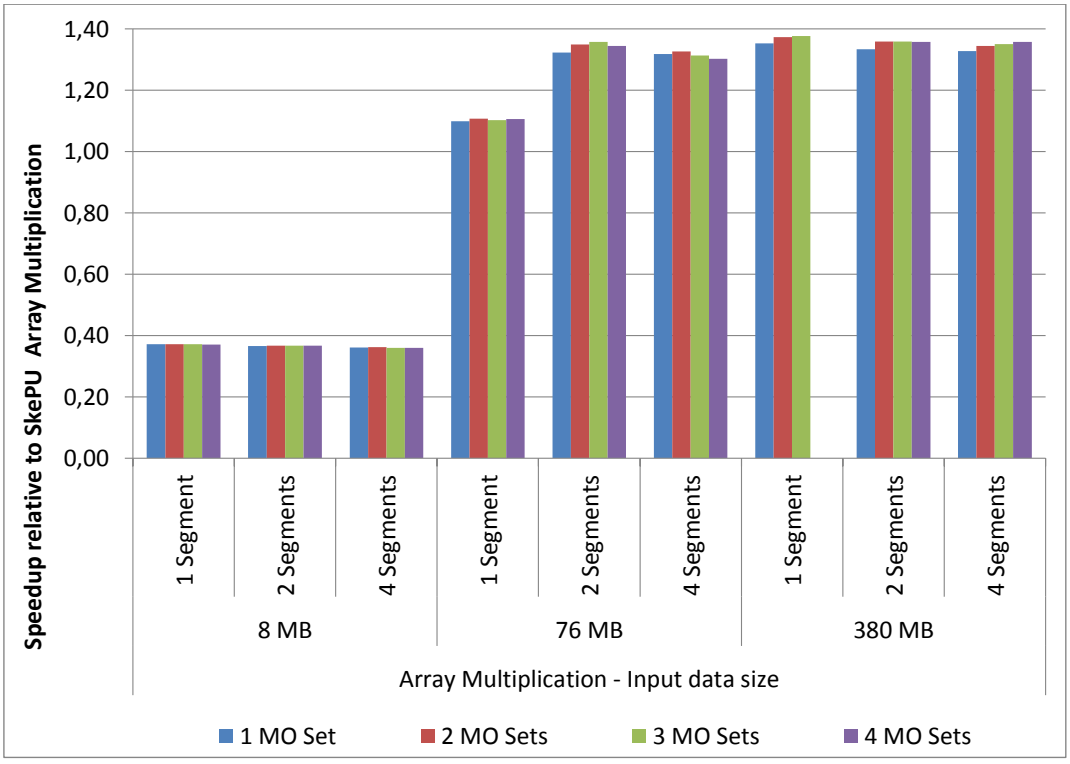


(b) Saxpy SkelCL speedup values, system S_2

Figure 5.8: Saxpy SkelCL Speedup Values

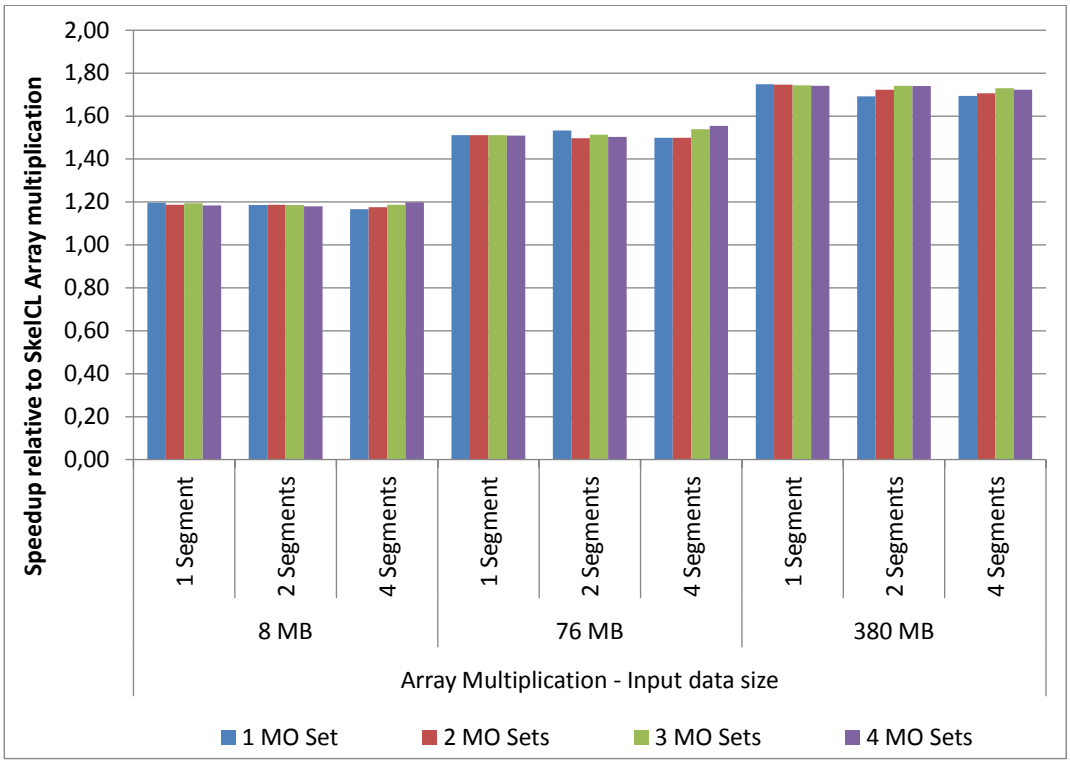


(a) Array Multiplication SkePU speedup values, system S_1

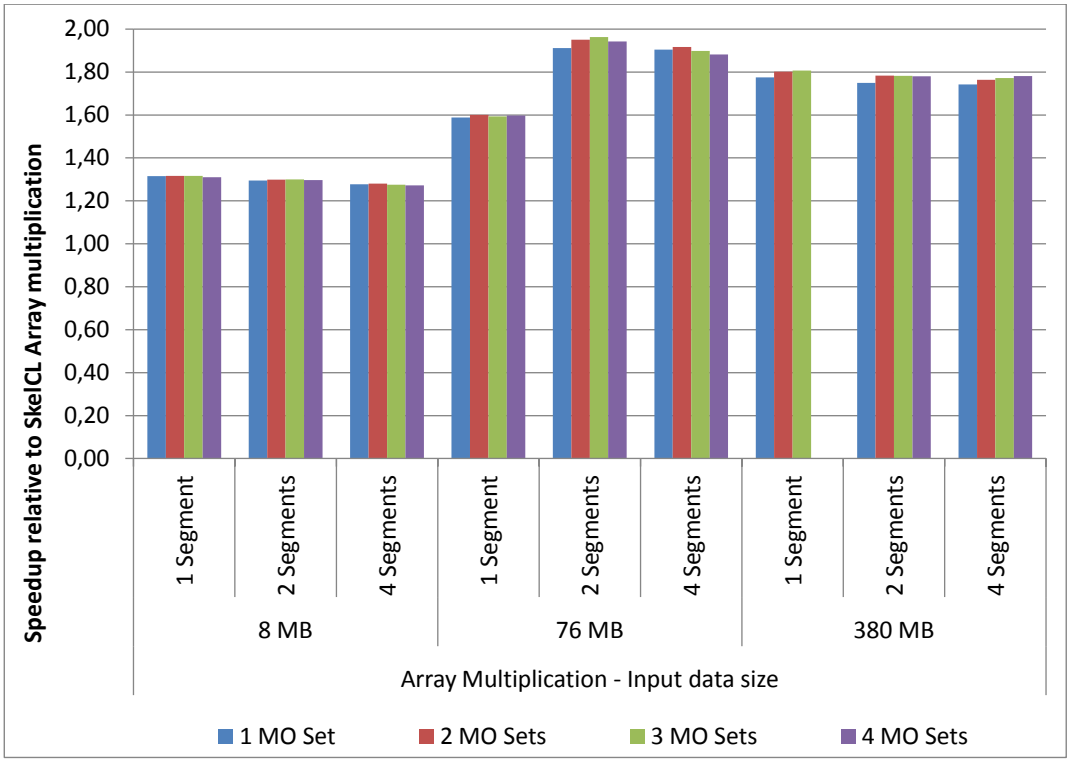


(b) Array Multiplication SkePU speedup values, system S_2

Figure 5.9: Array Multiplication SkePU Speedup Values

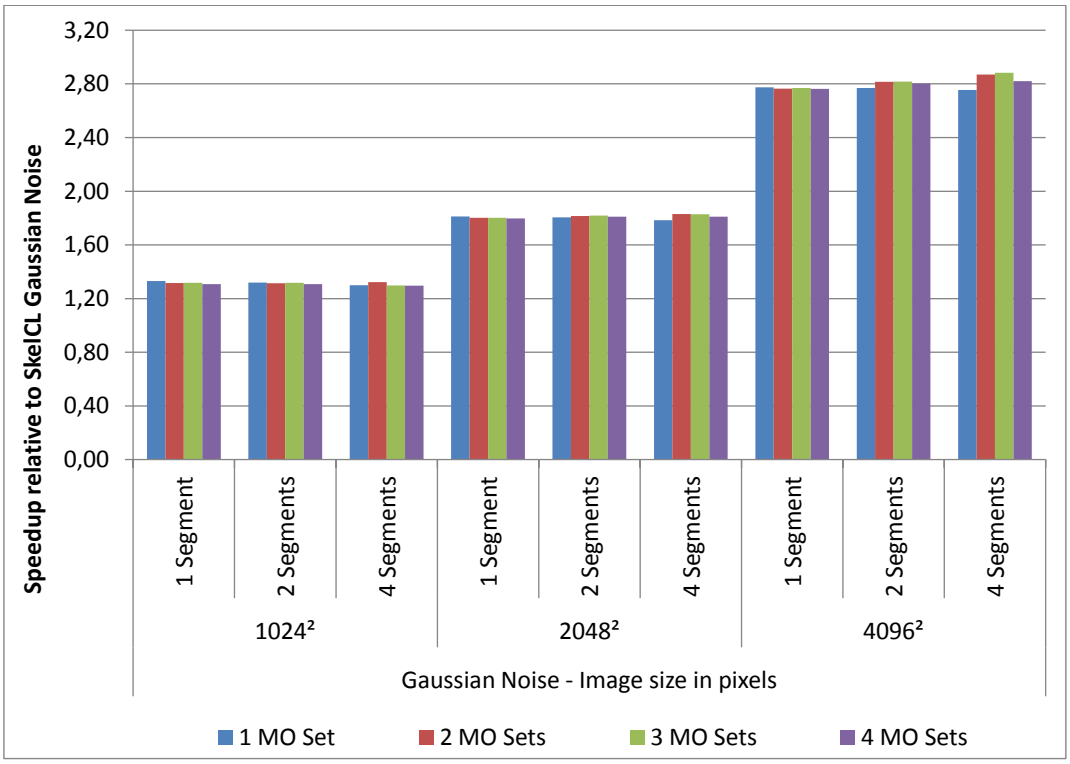


(a) Array Multiplication SkelCL speedup values, system S_1

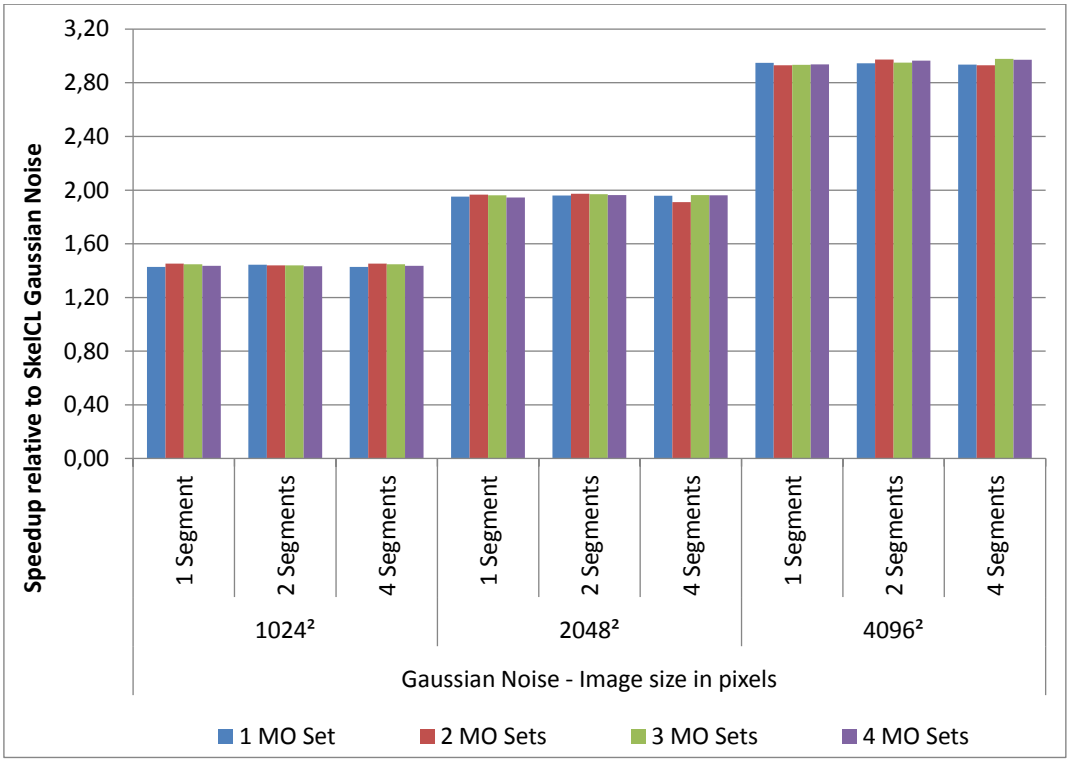


(b) Array Multiplication SkelCL speedup values, system S_2

Figure 5.10: Array Multiplication SkelCL Speedup Values



(a) Gaussian Noise SkelCL speedup values, system S_1



(b) Gaussian SkelCL speedup values, system S_2

Figure 5.11: Gaussian Noise SkelCL Speedup Values

The Gaussian Noise case-study (Figure 5.11) presents results similar to the ones obtained against its OpenCL version, in Section 5.1. In system S_1 (Figure 5.11a), when the input images are larger then 1024^2, computing four segments using two or three memory object-sets is best, performance wise. In contrast, in system S_2 the overlap performance gains are less noticeable, arguing in favour of computing the input image as a whole, and using just one memory object-set. Nonetheless, with or without overlap, the Marrow version is more efficient than the SkelCL one.

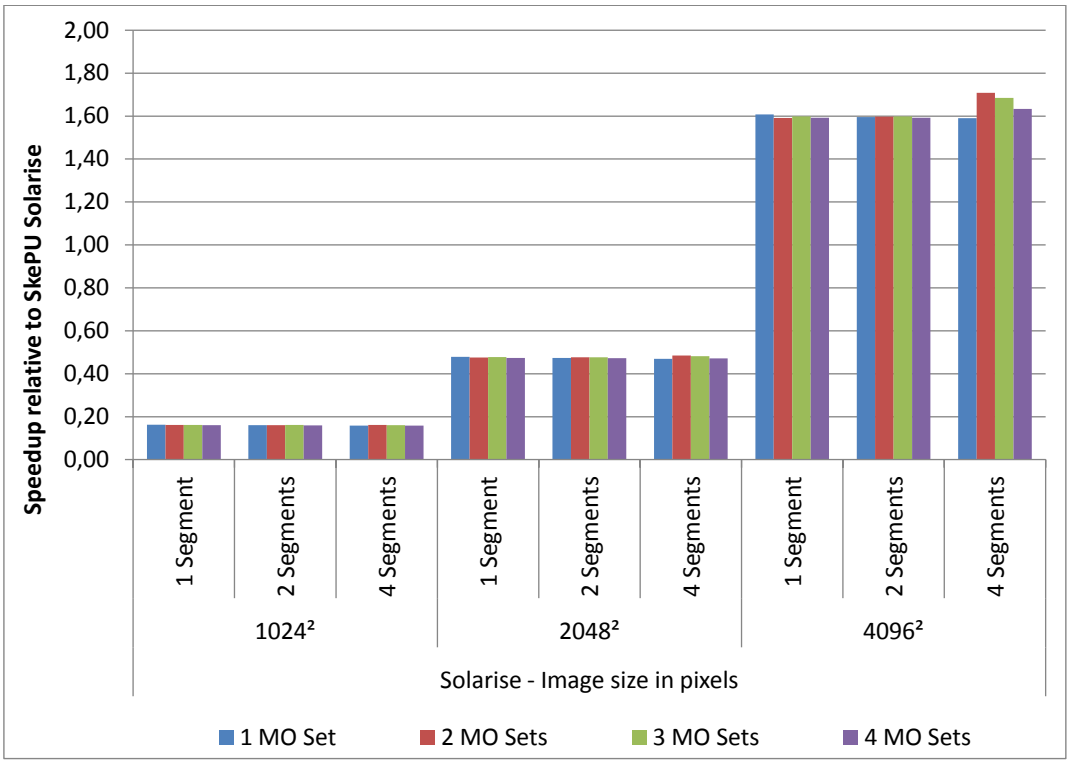
The final case-study (Figures 5.12 and 5.13) behaves similarly to the previous one, considering that it is a reimplementation with a different kernel. However, since applying the Solarise filter is not as computationally heavy as applying the Gaussian Noise filter, overlap between communication and computation should not lead to significant speedup increases. This premiss holds true. Overlap is only clearly beneficial on images of larger size, namely 4098^2. In terms of overall results, the Marrow version is more efficient than SkelCL's (Figures 5.13a and 5.13b), but only presents better results than SkePU's with the largest input (Figures 5.12a and 5.12b). As a side note, S_2's speedups are more significant than S_1's.

Regarding the better results obtained when running the SkePU versions of the case-studies, namely with smaller input grains. We were not able to disclose the reasons behind this unusually good performance. However, we suspect the latter may come as a result of our testing methodology. Every application run iteratively processes the same input $100 < N < 0$ times, so as to calculate an average of the time required to compute one input. We suspect that this strategy positively affects SkePU, because the slowest SkePU iteration of the average usually falls in line with the average iteration of both Marrow, and SkelCL. Consequently SkePU's performance behaviour may result from a caching mechanism, that caches the results of the first iteration. This would also explain why processing considerably larger input grains has proportionately worse performance, since the size of the cache may not have been sufficient to store the results.

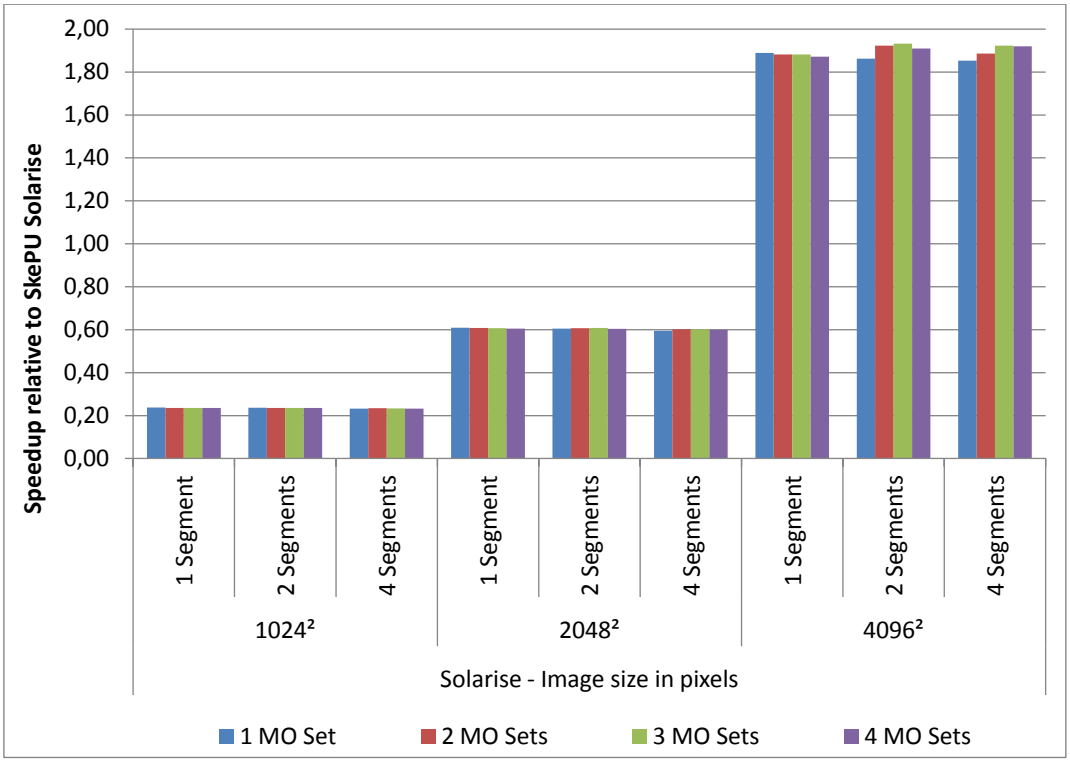
5.2.2 Programming Model Evaluation

Both SkePU's and SkelCL's programming models offer a more complete abstraction from the underlying computing model than Marrow's, also abstracting the programmer from the kernel domain. However, this comes at the expense of flexibility and implementation possibilities. These limitations, previously discussed in Chapter 2, effectively limited the comparison evaluation, given the small amount of distinct behaviours that both libraries support. In summary, Marrow supports a wider range of application uses than SkePU and/or SkelCL, giving the programmer more control over how the actual executions are performed.

In terms of productivity, Marrow is almost always surpassed by SkePU and SkelCL because of its verbose initialization stage, that may require defining C++ classes to use

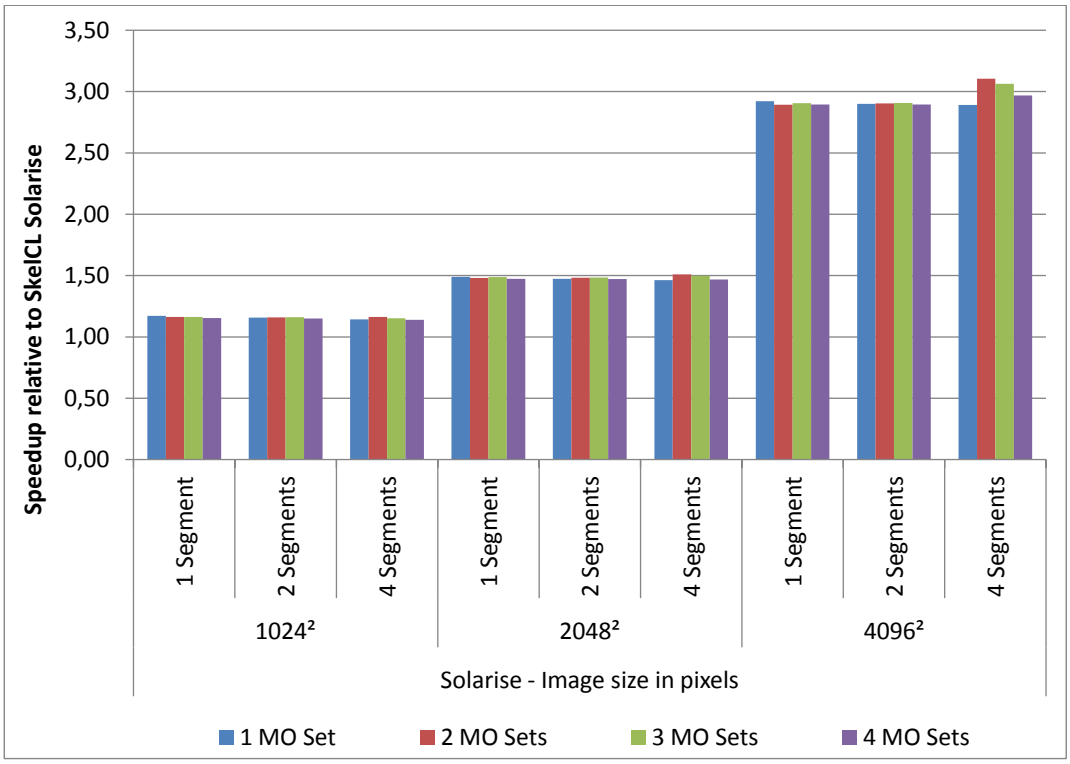


(a) Solarise SkePU speedup values, system S_1

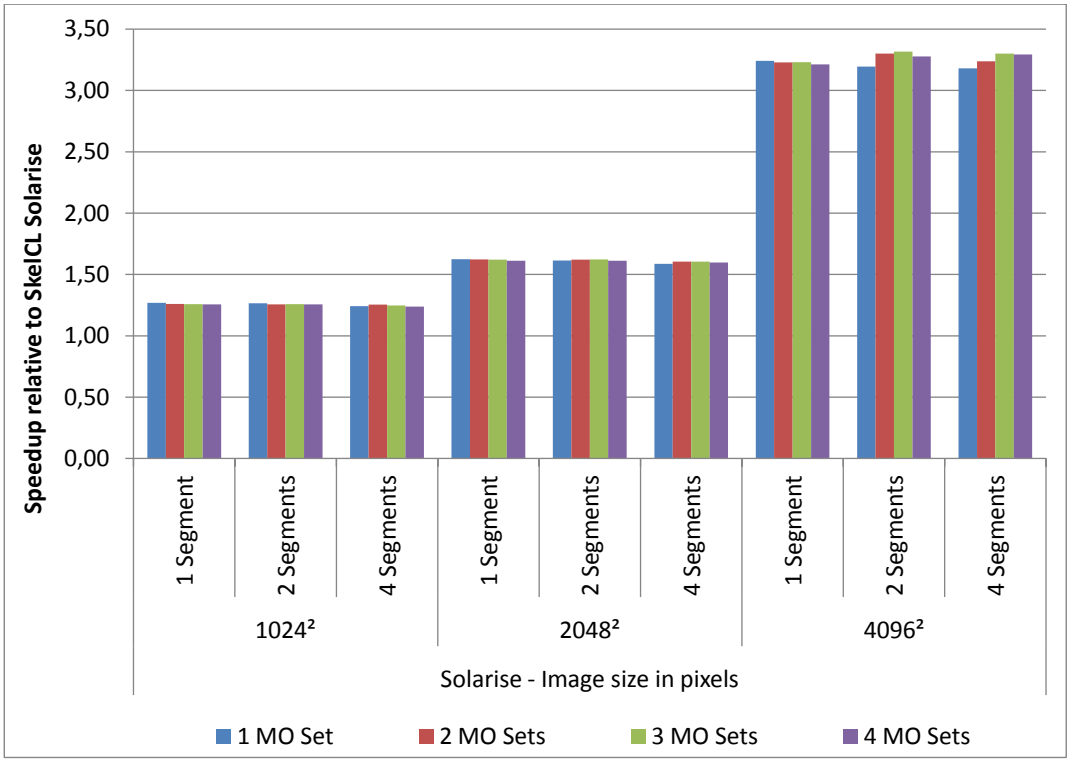


(b) Solarise SkePU speedup values, system S_2

Figure 5.12: Solarise SkePU Speedup Values



(a) Solarise SkelCL speedup values, system S_1



(b) Solarise SkelCL speedup values, system S_2

Figure 5.13: Solarise SkelCL Speedup Values

certain skeletons (e.g., *Loop*, *MapReduce*). Also, the process of prompting a skeleton execution on Marrow is more complex than in other libraries, given that Marrow's execution model is asynchronous. Be that as it may, Marrow's programming model is more flexible than others, on the account of the ability to nest skeletons, and the asynchronism of its main API function (write). Ultimatly, Marrow has the most verbose, and flexible programming model of the three libraries.

6

Conclusion

This final chapter presents a summarized view of the developed work. Firstly, it starts by highlighting some core aspects about our work (Section 6.1), such as: what is our thesis (and its objectives), what came out of our work, how did we evaluate that work, an what were the evaluation results. Lastly, it presents future research topics, to be included in further development of this project (Section 6.2).

6.1 Objectives and Results

Our thesis identified some shortcomings in the constructors offered for the development of complex GPU-based applications. The existing tools essentially focus on offloading isolated computations (kernels), which require a complex orchestration when the application scope goes beyond these limitations. To this extent, this dissertation addressed the design and implementation of a C++ algorithmic skeleton framework, called Marrow, whose main goal is to permit the high-level orchestration of OpenCL kernels. For this purpose, we wanted to broaden the set of skeletons available in the field, beyond the usual *MapReduce* and variants, by introducing task-parallel ones, like the *Pipeline*. Our skeletons offer a high-level programming model without compromising performance. In fact, they introduce transparent performance gains by overlapping communication with computation. This technique, implemented with a N-Buffering scheme, is not common in GPGPU ASkFs. In fact, those that offer it do not provide mechanisms for parametrizing it. Finally, Marrow addresses the lack of flexibility of other GPGPU ASkFs, by enabling skeleton nesting. This feature, as far as we know, is not yet available in the GPU context.

Our work resulted in a functional prototype, a C++ ASkF that incorporates all of the

previous objectives, namely: introducing new skeletons to the context of GPGPU, providing transparent performance gains via overlap, and enabling skeleton nesting. Consequently, we were able to achieve all of the proposed design goals. Additionally, our work resulted in an accepted paper [MPM12].

We divided the proposal's evaluation into two sub-domains: performance and productivity. The first was mainly interested in quantifying the performance benefits of an overlapped execution, while the last compares our library's programming model to other GPGPU technologies. We developed an extensive collection of multi-version case-studies for validating each and every proposed skeleton. Moreover, the case-studies served to confirm the performance advantages of both nesting and overlap between communication and computation. The resulting applications were executed on two systems with distinct GPUs. Lastly, we compared Marrow against technologies of lower and higher abstraction level, particularly OpenCL and SkePU/SkelCL, respectively. This allows us to better understand how Marrow fairs against its most direct competitors.

The accomplished evaluation attested the effectiveness of our proposals. Compared to hand-tuned OpenCL applications that do not introduce overlap, the *Stream* skeleton consistently boosted performance without compromising the simplicity of the Marrow programming model. In addition, the remaining skeletons supply a set of high-level constructs to develop complex OpenCL based applications with negligible performance penalties. Compared to other ASkFs for GPU computing, Marrow takes a different approach. It focuses on the orchestration of OpenCL kernels rather than worrying about both the orchestration and kernel programming. In this way, Marrow allows for a more flexible and powerful framework, whose kernels are bound only by OpenCL's restrictions, and whose skeleton set is richer and more modular (the nesting support is an example of such modularity). Still, Marrow surpasses SkelCL, in terms of performance, and scales better than SkePU.

A more thorough analysis of the executional behaviour of overlap between communication and computation indicates that its performance gains are not linearly equivalent in every system. To be fully beneficial, overlap has to be correctly parametrized according to a particular system and application. These results are interesting and open up new future research topics, within this context. In turn, the complexity of Marrow's programming model, in particular when applying overlap to a single data-set, is an issue. It can, however, be mitigated by auto-tuning techniques.

6.2 Future Work

Future work will focus on three major aspects: adding constructs that provide new behaviours, increasing Marrow's performance, and simplifying the programming model. There are other aspects that can be incorporated in Marrow, such as also providing an abstraction to the parallel computations. Be that as it may, we consider the three previous aspects as the most relevant research topics, in a short to medium term.

We plan on adding constructs that map recurring algorithmic behaviours in the context of GPU computing. For instance, a common execution model is to have a computing element (thread) apply a computation that not only uses its own data-elements, as input, but also those of its neighbourhood. This vicinity-based execution is present in many types of GPGPU-version algorithms, particularly in many image filter techniques (e.g., Gaussian Blur, Sobel Filter).

To increase performance, in a direct manner, we intend to provide multi-GPU support. Naturally, is not yet decided how we will accomplish such a functionally. We may distribute the execution requests evenly by a group of GPUs. Or even, divide the composition three into sections (or branches), and spread the resulting sub-trees by the available GPU accelerators.

The peculiarities, in regards to performance, of the N-Buffering strategy, made us realize that it is useful to have the library automatically detect the best execution parametrization for the specific application-system combination. This technique is commonly known as auto-tuning. The latter, can infer on the best values for a set of runtime variables. For example, it can detect the recommended memory chunk sizes, the ideal number of memory object-sets (for overlap), or even if a multi-GPU scenario provides better performance than a single-GPU one. Moreover, we can use auto-tuning to simplify our programming model, specially its first and second stage. It can hide the complexity of dividing the input data set into segments, and determine the best segment size.

Bibliography

[ADT03] M. Aldinucci, M. Danelutto, and P. Teti. "An advanced environment supporting structured parallel programming in Java". In: *Future Generation Comp. Syst.* 19.5 (2003), pp. 611–626.

[AMD11] AMD Corporation. *Aparapi API for data parallel Java*. `http://developer.amd.com/zones/java/aparapi/Pages/default.aspx`. 2011.

[BS10] P. Bakkum and K. Skadron. "Accelerating SQL database operations on a GPU with CUDA". In: *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010, Pittsburgh, Pennsylvania, USA, March 14, 2010*. Ed. by D. R. Kaeli and M. Leeser. Vol. 425. ACM International Conference Proceeding Series. ACM, 2010, pp. 94–103. ISBN: 978-1-60558-935-0.

[BD98] B. Bargen and P. Donnelly. *Inside DirectX*. First. Microsoft Press, 1998. ISBN: 1572316969.

[Ber] Berkeley, University of California. *The Parallel Computing Laboratory*. `http://parlab.eecs.berkeley.edu/`.

[Bly06] D. Blythe. "The Direct3D 10 system". In: *ACM Trans. Graph.* 25.3 (2006), pp. 724–734.

[Bro10] N. Brookwood. "AMD Fusion Family of APUs: Enabling a Superior Immersive PC Experience". In: *Insight* 64.1 (2010), pp. 1–8.

[BFHSFHH04] I. Buck, T. Foley, D. R. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. "Brook for GPUs: stream computing on graphics hardware". In: *ACM Trans. Graph.* 23.3 (2004), pp. 777–786.

[BMRSS96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., 1996. ISBN: 0471958697.

[CFMQRVV10] 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.

[CL07] D. Caromel and M. Leyton. "Fine Tuning Algorithmic Skeletons". In: *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Proceedings*. Ed. by A.-M. Kermarrec, L. Bougé, and T. Priol. Vol. 4641. Lecture Notes in Computer Science. Springer, 2007, pp. 72–81.

[CGSDKEPSS05] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, and V. Sarkar. "X10: an object-oriented approach to non-uniform cluster computing." In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*. ACM, 2005, pp. 519–538.

[Col91] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Cambridge, MA, USA: MIT Press, 1991. ISBN: 0-262-53086-4.

[Cor09] N. Corporation. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. `http://www.nvidia.co.uk/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`. 2009.

[CBS11] D. Cunningham, R. Bordawekar, and V. Saraswat. "GPU programming in a high level language: compiling X10 to CUDA". In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. X10 '11. ACM, 2011, 8:1–8:10.

[DM98] L. Dagum and R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science Engineering, IEEE* 5.1 (1998), pp. 46–55. ISSN: 1070-9924. DOI: `10.1109/99.660313`.

[DD06] M. Danelutto and P. Dazzi. "Joint Structured/Unstructured Parallelism Exploitation in Muskel". In: *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part II*. Ed. by V. N. Alexandrov, G. D. van Albada,

P. M. A. Sloot, and J. Dongarra. Vol. 3992. Lecture Notes in Computer Science. Springer, 2006, pp. 937–944.

[EPL94] T. Ellis, I. Philips, and T. Lahey. *Fortran 90 programming*. Addison-Wesley Reading, MA, 1994.

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

[EK12] S. Ernsting and H. Kuchen. "Algorithmic skeletons for multi-core, multi-GPU systems and clusters." In: *International Journal of High Performance Computing and Networking (IJHPCN)* 7.2 (2012), pp. 129–138.

[Fly72] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *Computers, IEEE Transactions on* C-21.9 (1972), pp. 948 –960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[Gam95] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GVL10] H. González-Vélez and M. Leyton. "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers". In: *Software: Practice and Experience* 40.12 (2010), pp. 1135–1160.

[Har05] M. Harris. "Mapping computational concepts to GPUs". In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. ACM, 2005.

[Kas09] Kaspersky Lab. *Kaspersky Lab utilizes NVIDIA technologies to enhance protection*. http://kaspersky.com/about/news/business/2009/Kaspersky_Lab_utilizes_NVIDIA_technologies_to_enhance_protection. 2009.

[KMMS10] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders. "A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects". In: *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ParaPLoP '10. Carefree, Arizona: ACM, 2010, 9:1–9:8. ISBN: 978-1-4503-0127-5. DOI: http://doi.acm.org/10.1145/1953611.1953620.

[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 2010*. IEEE Computer Society, 2010, pp. 289–296.

[MPM12] R. Marques, H. Paulino, and P. D. Medeiros. "Desenho e Implementação de uma Biblioteca de Padrões Algorítmicos para GPGPU". In: *INForum 2012 - Atas do 4º Simpósio de Informática*. Ed. by A. Lopes and J. O. Pereira. Universidade Nova de Lisboa, 2012, pp. 298–301.

[McC06] M. D. McCool. "Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform". In: *GSPx Multicore Applications Conference*. Vol. 9. 2006.

[MQP02] M. D. McCool, Z. Qin, and T. S. Popa. "Shader Metaprogramming". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Eurographics Association. 2002, pp. 57–68.

[Moo65] G. E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (1965), pp. 114–117. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[Mun+09] A. Munshi et al. *The OpenCL Specification*. Khronos OpenCL Working Group. 2009.

[NDW97] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, 1997.

[NSLMGTWDCW+11] C. J. Newburn, B. So, Z. Liu., M. McCool., A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, et al. "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language". In: *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE. 2011, pp. 224–235.

[NVI] NVIDIA Corporation. *NVIDIA CUDA*. http://www.nvidia.com/object/cuda_home_new.html.

[NVI09] NVIDIA Corporation. *NVIDIA OpenCL Best Practices Guide*. First. NVIDIA, 2009.

[Ope11] OpenACC. *The OpenACC Application Programming Interface (version 1.0)*. http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf. 2011.

[OHLGSP08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879 –899. ISSN: 0018-9219. DOI: 10.1109/JPROC.2008.917757.

[Pap07] M. Papakipos. *The PeakStream Platform: high-productivity software development for multi-core processors*. `http://download.microsoft.com/download/d/f/6/df6accd5-4bf2-4984-8285-f4f23b7b1f37/WinHEC2007_PeakStream.doc`. 2007.

[PSFW12] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch. "Rootbeer: Seamlessly using GPUs from Java". In: *14th IEEE International Conference on High Performance Computing & Communication, HPCC 2012, Liverpool, UK, June 25-27, 2012*. IEEE Computer Society, 2012.

[SABCCGKPT10] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. "The asynchronous partitioned global address space model". In: *Proceedings of the First Workshop on Advances in Message Passing*. 2010.

[SG96] M. Shaw and D. Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice Hall, 1996.

[SMV10] K. Spafford, J. S. Meredith, and J. S. Vetter. "Maestro: Data Orchestration and Tuning for OpenCL Devices". In: *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*. Ed. by P. D'Ambra, M. R. Guarracino, and D. Talia. Vol. 6272. Lecture Notes in Computer Science. Springer, 2010, pp. 275–286.

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

[TPO06] D. Tarditi, S. Puri, and J. Oglesby. "Accelerator: using data parallelism to program GPUs for general-purpose uses". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006*. ACM, 2006, pp. 325–335.

[TKA02] W. Thies, M. Karczmarek, and S. P. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by R. N. Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 179–196.

[UGT09] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. "Software Pipelined Execution of Stream Programs on GPUs". In: *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '09. IEEE Computer Society, 2009, pp. 200–209.