
**Fábio Miguel Cardoso Soldado**

*Nº 36564*

# Heterogeneous Computing with an Algorithmic Skeleton Framework

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

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

Júri:

| | |
|---|---|
| Presidente: | Doutor João Baptista da Silva Araújo Junior |
| Arguente: | Doutor Nuno Roma |
| Vogal: | Doutor Hervé Miguel Cordeiro Paulino |

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

**Março, 2014**

**Heterogeneous Computing with an Algorithmic Skeleton Framework**

*To my parents and my sister, for all this awesome years so far*

# Acknowledgements

I would like to thank, in first place, to my adviser Hervé Paulino, for all the support during the time we worked together, and for always being available to help me at any time. Secondly, I want to thank the FCT/UNL, for providing a good environment and working conditions.

A special thanks goes to my course friends, specially André Alves, Pedro Leonardo, Paulo Dias and Carlos Loureiro, for the mutual emotional support in the harder times.

I could not finish without thanking my family, specially my parents and grandparents, for being able to fully support me during my studies, both economically and emotionally.

# Abstract

The Graphics Processing Unit (GPU) is present in almost every modern day personal computer. Despite its specific purpose design, they have been increasingly used for general computations with very good results. Hence, there is a growing effort from the community to seamlessly integrate this kind of devices in everyday computing. However, to fully exploit the potential of a system comprising GPUs and CPUs, these devices should be presented to the programmer as a single platform.

The efficient combination of the power of CPU and GPU devices is highly dependent on each device's characteristics, resulting in platform specific applications that cannot be ported to different systems. Also, the most efficient work balance among devices is highly dependable on the computations to be performed and respective data sizes.

In this work, we propose a solution for heterogeneous environments based on the abstraction level provided by algorithmic skeletons. Our goal is to take full advantage of the power of all CPU and GPU devices present in a system, without the need for different kernel implementations nor explicit work-distribution.To that end, we extended Marrow, an algorithmic skeleton framework for multi-GPUs, to support CPU computations and efficiently balance the work-load between devices. Our approach is based on an offline training execution that identifies the ideal work balance and platform configurations for a given application and input data size.

The evaluation of this work shows that the combination of CPU and GPU devices can significantly boost the performance of our benchmarks in the tested environments, when compared to GPU-only executions.

**Keywords:** Algorithmic Skeletons, OpenCL, GPGPU, Heterogeneous Computing.

# Resumo

O GPU (Graphics Processing Unit) está presente em praticamente todos os computadores actuais. Apesar da sua finalidade específica, este processador tem sido alvo de crescente uso em computação de carácter mais geral, com resultados bastante promissores. Como tal, tem havido um crescente esforço para integrar este dispositivo na programação mais geral. No entanto, para que o potencial de sistemas constituidos por CPUs e GPUs possa ser explorado ao máximo, estes dispositivos devem ser apresentados ao programador como uma plataforma única.

Uma combinação eficiente do poder de CPUs e GPUs depende muito das características de cada um destes dispositivos, resultando em aplicações específicas para uma plataforma que não manterão a mesma eficiência quando portadas para sistemas diferentes. Além disso, o balanceamento de carga entre estes dois dispositivos depende também das computações executadas assim como dos respectivos tamanhos dos dados.

Neste trabalho, propomos uma solução para ambientes heterogéneos baseada no nível de abstracção fornecido pelos *Algorithmic Skeletons*. O nosso objectivo é tirar partido do poder de todos os CPUs e GPUs presentes num sistema sem que haja a necessidade de definir diferentes *kernels* ou dividir a carga explicitamente. Deste modo, estendemos o Marrow, uma *framework* de *Algorithmic Skeletons* para multi-GPUs, para suportar computações OpenCL no CPU e balancear a carga do trabalho entre os dispositivos de forma eficiente. A nossa abordagem é baseada num treino *offline*, que identifica o balanceamento de carga e a configuração da plataforma ideais para uma dada aplicação.

A avaliação deste trabalho mostra que a combinação do CPU e do GPU foi capaz de melhorar significativamente os resultados dos nossos testes nos ambientes de execução em que foram corridos, quando comparados com execuções apenas nos GPUs.

**Palavras-chave:** Padrões Algorítmicos, GPGPU, OpenCL, Computação Heterogénea.

# Contents

# List of Figures

# List of Tables

# Listings

# 1

# Introduction

## 1.1 Motivation

The Graphic Processing Unit (GPU) is a common component found in virtually every modern computer. Over the past years, this kind of processors experienced a great evolution, mainly driven by the gaming industry, reaching a point where they can deliver high levels of performance, unachievable with modern Central Processing Units (CPUs), when executing graphics-related computations. These performance levels grow from the particular characteristics of GPU hardware, formed by many streaming processors, with clock frequencies generally lower. By being a common component found in today's personal computer's system, and because of its purpose specific architecture, the cost of this processors are often labeled as low-priced, considering the performance levels they can output [1]. Also, due to their lower clock frequencies, they are also energy-efficient solutions, all characteristics that make this units desirable to use in fields other than graphics processing.

Using the GPU in general purpose computations (GPGPU), although desirable, raises several challenges related to its parallel and graphics targeted architecture. Nonetheless, the potential of this hardware has driven a lot of research with the goal of providing a friendlier way to program this kind of processors, by implicitly taking care of intricacies related to memory management and work parallelization (for example [2, 3, 4, 5]), leading to a more general acceptance and recognition of GPGPU.

The emergence of higher level programming tools to deal with GPUs, obfuscate some of the low level details exposed by the base GPGPU programming frameworks, OpenCL and CUDA. As such, this processor became available to a wider range of programmers. However, GPU programming still requires much more effort from the programmer than

the tools available for its CPU counterpart. Firstly, most of the current proposals only focus on providing a higher level interface for the GPU, leaving it to the programmer to identify which computations should be offloaded to the GPU and which ones should stay on the CPU. This will have an heavy impact on the system's overall performance as it will be dependent on the programmer's ability to identify and efficiently schedule computations on both processors, considering the intricacies of each of them. The offloading of an execution to the GPU must take into account the overhead introduced by the data transfers between the memory of both devices. On the other hand, when considering multi-core CPU computations, the memory is shared between different threads but there is still a need to partition the work among those threads, while avoiding race conditions.

Given the increasing utilization of the GPU in general purpose computing, our motivation grows from the belief that there is a need for tools that allow the programmer to address this heterogeneous system as a whole, delegating the work scheduling and efficiency concerns to the compiler/runtime system, providing a friendlier and generalized way to fully harness the available computing power of the underlying computational infrastructure.

## 1.2 Problem

Programming GPUs, even with the support of high-level programming platforms, is still a more complex task than addressing sequential or even multi-core CPU processors. This complexity increases when there is a need to address both CPUs and GPUs, due to their different execution models. For this reason, it is desirable for these processors to be presented to the programmer as a part of a single computing platform. In our opinion, CPU/GPU heterogeneity should be handled at system level, being it by the language's compiler, a dedicated run-time system or, even, by the operating system. However, to accomplish such grounds, some challenges have to be surpassed.

The CPU and the GPU have different associated execution models due to their different architectures. For them to be presented as a single platform, there is a need to find a suitable model to address both this processors. This problem is only partially addressed by most of the current proposals, as their focus is on the GPU programmability. Some solutions support both GPUs and multi-core CPUs, but these must be used in mutual exclusion, or the device scheduling and work distribution is left to the programmer.

There is, however, a new trend of platforms that aim to address the system's underlying heterogeneity internally, providing the user with a unified programming platform. Some of these solutions simply pick the best performing device to execute a computation, without achieving device cooperation. On the other hand, the solutions that do achieve device cooperation, they either require different implementations of the parallelizable computation for the various devices, or they are too purpose specific and cannot be used in a wider range of applications.

We believe that there is still work to be done, targeting implicitly schedule of different computations into the different processors in an efficient way, and without requiring extra work from the programmer to prepare the application to be executed on multiple devices.

In this work, we aim to exploit the abstraction power of the Algorithmic Skeletons to address this challenges. Skeletons, by definition, hide the implementation details of commonly found algorithmic patterns, providing a parameterizable interface to the programmer. Furthermore, since the skeletons encompass all the computation, they can convey, to the runtime system, all the information the latter requires to make the right decisions regarding work-load partitioning and work scheduling.

## 1.3 Proposal

Our proposal grows from Marrow[6, 7], an Algorithmic Skeleton Framework (ASkF) for the orchestration of complex OpenCL-based GPGPU computations. Among its most notorious features, there are the skeleton nesting and the task-parallel skeletons, features not found in other GPU ASkF. Also, it implements a set of optimizations for performance improvement, including communication and computation overlapping.

The main goal of this thesis is to extend the Marrow framework so that it may take advantage of both the multiple CPU and the multiple GPU devices present in a single node architecture. Our motivation is to fully exploit the computational power available, in an efficient way, while maintaining the current level of abstraction provided by the supported skeletons.

In order to achieve this goal, we have to start by introducing the support for CPU computations in Marrow and subsequently guarantee the cooperation and work-load between CPU and GPU devices without the programmers' intervention. The current version of Marrow already supports multi-GPU devices, being that the division of the work is performed when the skeleton tree is created, restricting the data dimensions to the values defined upon creation. We propose to lift this restriction by creating a more dynamic solution where the system can adapt itself to the dimension of the inputted data and also to the current system's work-load.

Like in the current version of Marrow, in the solution we propose, the work partitioning will be achieved prior to the execution offload, avoiding the need for work transfers during runtime. However, the current work distribution method is based on performance values acquired prior to the applications execution that are used to calculate a performance ratio between devices. This approach is not suitable for work distribution among CPU and GPU devices, due to the huge differences in their architecture and execution model. Therefore, we propose a solution based on an offline training where different work distribution configurations are tested, to find the ideal work-balance for each different application.

To address the possible work-load fluctuations of the CPU device, we aim to implement a lightweight monitoring of the running executions, comparing the performance of CPU and GPU devices and adjusting the work-balance when a significant performance gap is detected. Finally, we intend to explore the OpenCL device fission functionality to leverage locality in CPUs.

Our goal is to take advantage of the abstraction level provided by Algorithmic Skeletons to provide support for programming in heterogeneous systems, composed by multiple CPUs and GPUs, an approach that, as far as we know, is not addressed by the state of the art.

## 1.4 Contributions

The main contributions of this work are:

- The introduction of support for multiple OpenCL device types in Marrow;
- Backend for CPU OpenCL executions;
- A framework that combines offline application training with online performance monitoring with specific algorithms for both stages.
- An evaluation of the performance gains of this implementation when compared to GPU-only executions, as well as an evaluation of the execution of Marrow in a CPU-only environment.

The work also makes two secondary contributions which are:

- Dynamic adjustment of work distribution in successive executions of a given Marrow computation. This implied modifications on the execution model and on the frameworks API;
- An evaluation of the OpenCL device fission functionality.

## 1.5 Document Organization

The remainder of The remainder of this document is organized as follows:

- In Chapter 2, we analyze the current state of the art with a focus on GPU offloading frameworks, skeleton frameworks and automatic backend selection solutions;
- Chapter 3 describes the Marrow framework, in its multi-GPU version, which was the state of this framework prior to our work;
- In Chapter 4 we describe the modifications we implemented over Marrow in order to meet our goals.

- The evaluation of our work is presented in Chapter 5, focusing on the performance gains achieved by using both CPU and GPU devices, when compared to GPU-only executions;
- Finally, in Chapter 6, we revisit the our goals and introduce some ideas on how the Marrow framework could be further extended.

# 2

# State of the Art

Algorithmic skeletons have been a recurrent solution for abstracting parallelization intricacies in homogeneous and heterogeneous environments. The abstraction layer provided by skeletons has proved to be useful under different system configurations, from single node computers to clusters and grids. In this chapter, we analyze the existing work regarding skeleton frameworks (and similar solutions, like language templates), with a focus on tools directed to GPU and/or CPU parallel programming.

Before describing the existing tools related to our work, we start by introducing the concept of algorithmic skeletons and describing the skeletons more commonly found in the existing frameworks. We also introduce the concept of heterogeneous computing and characterize the most common approaches to this paradigm. Given our focus on both CPU and GPU programming, we also include some tools that target multi-core CPU computing through algorithmic skeletons.

## 2.1 Algorithmic Skeletons

The concept of algorithmic skeletons was proposed by Murray Cole, in 1989 [8]. The algorithmic skeletons model tries to identify commonly used patterns in parallel programming, namely concerning computation, communication and interaction [9], to provide a higher level of abstraction. Software developers need only to choose the skeleton that better fits the problem at hand. This greater level of abstraction enhances portability since the particularities of each architecture will be internally handled by the skeletons. This allows for each skeleton to be extensively optimized for each different architecture, resulting in levels of performance hard to achieve without this level of abstraction. Moreover, by abstracting the programmer of the parallel programming concerns, this model

also prevents typical programming errors derived from its extra complexity, resulting in a more efficient way to develop software.

Algorithmic skeletons were firstly thought as a solution for cluster computing. The distributed nature and hardware diversity of cluster computing demanded programmers to be aware of the underlying hardware's characteristics. This resulted in hardware-specific programs, with hard portability and probably poor performance in different systems, with distinct hardware specifications, since they were only optimized for the system they were developed to. Also, determining which part of a program could and should be parallelized was not always a trivial task and could also be dependent on the underlying hardware's idiosyncrasies, resulting in a solution that may not be optimal for that particular system. Moreover, the behavior of a system comprising several parts that work in parallel is harder to predict than the behavior of a system that works sequentially. As soon as general purpose computers began to incorporate parallel hardware, the concept was adopted to single machine computations, not only for multi-core CPUs, but also for GPUs.

Skeletons can be divided in three main categories: Data-parallel, Task-parallel and Resolution skeletons. Data-parallel skeletons handle problems with large data structures, dealing with challenges such as splitting the data among all running threads and merging the results after a routine is applied. Examples are *Map*, *Reduce* and *Zip*. Task-parallel skeletons deal with problems related to task interaction, like communication and synchronization. Examples are the *Pipeline*, *Farm* (also know as *Master-Slave*), *For* and *While*. Finally we have the Resolution skeletons, which define algorithms to solve a family of problems. Two examples are the *Divide & Conquer* and the *Branch & Bound*. We now describe the most popular skeletons included in these categories.

- Data-parallel
  - **Map** - Applies a function or a sub-skeleton to all elements of a set of data. This skeleton can be conceived as a Single Instruction, Multiple Data (SIMD) skeleton as parallel threads running the same instructions can be applied concurrently over different data elements.
  - **Reduce** - Scans a data-set from left to right applying a function to each pair of elements, resulting in a single element being produced. The concurrency potential will depend on the associativity property of the function applied. This skeleton is also called *Fold* by some frameworks.
  - **Scan** - This skeleton has a behavior similar to Reduce, the difference laying in the resulting type. Instead of a single element, Scan returns a data-set of the same size as the input, where each element is the result of applying the reduction function to all the elements in the input set with lower or equal indexes. This intermediate results are important to a subset of algorithms and that is why some frameworks support this skeleton.

  - **Zip** - This skeleton receives two data-sets and an operator as input, and outputs the resulting set of applying the operator over the input elements with the same position.
  - **Fork** - This skeletons works similar to map but a different function is applied to each element of the data-set, making it into a Multiple Instruction, Multiple Data skeleton.

- Task-Parallel

  - **Farm** - Also known as *Master-Slave*, consists in the creation of independent tasks, achieving parallelism through scheduling different tasks to distinct resource, so they can be simultaneously computed.
  - **Pipeline** - This skeletons consists in the notion of staged computations where the output of one stage is the input on the next one. Since each stage is dependent on the termination of the previous stage, parallelism can only be achieved through executing different stages simultaneously on different inputs.
  - **For** - Executes a sub-skeleton or a function for a specified number of times. Parallelism can only be achieved if the result of one iteration is independent of the result of the previous one.
  - **While** - Behaves like For but instead of iterating a fixed number of times, the cycle termination is decided by a condition.

- Resolution Skeletons

  - **Divide & Conquer** (D&C) - This skeleton can be viewed as a generalization of the Map skeleton. It works by recursively splitting the problem into subproblems and calculating partial solutions in parallel, merging the results in the upper levels of the recursion, ultimately reaching the global solution.
  - **Branch & Bound** (B&B) - B&B algorithms are applied on NP-hard problems where reaching the optimal solution is only possible with algorithms with exponential complexity because it requires all the possible solutions to be tested. Instead of looking for the ultimate optimal solution, B&B skeletons try to reach the best solution by recursively test several possible solutions until a bound is met. It works by finding an initial possible solution, calculate its optimality value (known as *fitness*) and generate alternative possible solutions based on that one. The same process will be recursively applied to every possible solution, but only the solutions with the best fitness value will be further explored. The algorithm will terminate when a determined bound is met. That bound can be either time-based, cycle-based or when a solution with a fitness value over some pre-established limit is met.

## 2.2 Heterogeneous Computing

Heterogeneous computing systems are not a new concept. For a long time, purpose-specific processors have been used to assist the CPU in some particular operations. The most notorious example of a purpose-specific processor is the GPU, a processing unit in charge of graphics processing in almost every nowadays computer. This kind of processors are very optimized for graphics related operations, outputting high performance levels, generally not achievable by the general-purpose processor when performing this type of tasks. GPUs were firstly design as fixed function pipelines [10], meaning that a pre-determined set of functions were specified in the hardware, and implementing a new function was only possible through hardware modifications. Nonetheless, the scientific simulation community identified an untapped computation source that could be leveraged in their highly resource demanding computations. The downside of these processors was that, being design with specific intents, it was really hard or even impossible for programmers to implement operations that differed from the processors original purpose. However, this new interest, conjoined with the evolution of GPU hardware, potentiated this kind of processors to become increasingly more programmable, allowing them to be applicable on a wider range of fields, ultimately leading to the rise of the GPGPU (General-Purpose Computing on GPU) concept. GPGPU is the designation of using the GPU for computations involving non-graphical tasks.

Despite being programmable, the existing APIs for GPUs (like OpenGL), were targeted to graphics processing and all the operations had to be addressed as graphical problems, meaning it was still really hard to program and take full advantage of this processors, when addressing problems other than graphical ones. This lead to an effort to reach more generic APIs. Brooks [11] was the first attempt, and although it was not adopted by the industry, it was an important proof of concept to show the potential of GPUs in different grounds. Consequently, it was not long until the industry itself recognized that potential and in 2006, NVIDIA, one of the main GPU manufacturers, unveiled CUDA [12], the first actual solution to GPGPU.

After NVIDIA released CUDA, other manufacturers wanted to join the race. Being created by NVIDIA, CUDA was only supported by NVIDIA graphical cards and the community craved for the specification of an open standard for GPGPU. Such necessity spawned the OpenCL [13] proposal, a standard specified by the Khronos Group and today is implemented by a large list of manufacturers, including NVIDIA, Intel, AMD and Apple.

The existence of an open standard further paved the path for the continuous growth of GPGPU field, but some challenges are yet to overcome, namely the increasing of the abstraction level. CUDA and OpenCL are too close to the hardware and the programmers still need to have some insight about the GPUs architecture to take full advantage of the parallelism they offer. This difficulty, together with the necessity to support other programming languages other than C or C++ (the languages supported by CUDA

and OpenCL), let to numerous independent efforts to bring GPGPU programming to an higher abstraction level. Our focus is on tools that allow orchestration of GPU and CPU computing, instead of just making GPU computations available to the programmer.

### 2.2.1 On the programming of Heterogeneous Systems

The popularization of the GPGPU led to the proliferation of GPU programming frameworks. On one hand, pre-existing parallel programming frameworks have been extended to benefit from the power of this processors. On the other, this device propelled the rise of new solutions developed from scratch with the GPU programming in mind. GPGPU has been tackled from different angles. Regarding the programming support, the most popular approaches are the directive-based frameworks, programming languages and high-level libraries.

**Directive-based frameworks** are tools that convert source code written in the host programming language (like C or Fortran) into code of another language (like CUDA or OpenCL). These, tools, also known as source-to-source compilers, aim to have a minimal impact on the sequential version of the code, requiring little to no modifications to the source, other than the directive annotations (also known as pragmas) on the parts of the code that can be parallelized. Therefore, despite providing a unified programming model, it still relies on the programmer to identify the parallelizable parts of the program and to specify how the compiler must interpret such code. Also, additional options can be specified, in order to increase the program's performance. Two examples of such compilers are hiCUDA [14](C-to-CUDA) and Bones [15] (C-to-CUDA and C-to-OpenCL).

Another way to provide GPU support in a higher level is through **native language level support**. Some programming languages directed to cluster computing, like X10 [16, 17] and Chapel [18, 19], have been adapted to keep up with the GPGPU trend, others, like Lime [20], have been design from scratch with GPU programming in mind.

GPU support can also be provided through **high-level libraries** that hide certain lower level aspects of GPU programming, like device communication and computation offloading. A common way to provide abstraction over the GPU parallelization is through algorithmic skeletons, or templates (like SkePU [3] and SkelCL [2]), identifying common parallelization patterns and providing a generalized implementation of such patterns, leaving it to the programmer to just parameterize them according to the specificities of the application.

In the context of our work we are more interested in classifying these frameworks according to their support for heterogeneous computing. Accordingly, we will divide them into three categories, which will drive the structure of remainder of this section:

- **GPU offloading tools:** These are the tools that focus solely on offloading computations to one or more GPU devices, without support for other backends;
- **Multi-backend with single or explicit offloading:** In this category we include

frameworks that provide support for both GPUs and multi-core CPUs (and possibly other backends), but each computation is explicitly offloaded by the programmer, or when only one backend is active at a time, the remaining working as a fallback in case the primary backend cannot execute the computation;

- **Automatic Backend Selection:** This category contains the tools that not only provide support for multiple backends (GPUs, CPUs and possibly other devices), but also provide mechanisms to implicitly distribute work-load among such devices.

### 2.2.2 GPU offloading tools

**SkelCL [2]** is a C++ skeleton framework that provides support for multi-GPU executions through OpenCL. SkelCL provides implicit data exchange through an abstract vector data type and lazy copying to minimize communication overhead between devices. Every skeleton in SkelCL receives a vector as input and produces a vector as output. To generate an OpenCL kernel, every skeleton receives a function as a string parameter that will be merged with the skeleton's own source to form the OpenCL code. The kernel is compiled during runtime, but since it can be a time-consuming task, after compiled, the compiled kernel is stored on disk for future utilizations. To pass more arguments to the kernel function than the number defined by each skeleton, every skeleton can receive an additional argument, an object of the type `Arguments`. All the additional arguments must be packed inside this object and passed to the skeleton. When multiple GPUs are present, a vector can be distributed through all the available devices, either by completely copying them to every device, or evenly splitting the vector into different parts, to perform executions over the array simultaneously by all the the devices. SkelCL provides skeleton-specific distribution so the vector can be divided implicitly, although, the programmer can take control and explicitly set the Vector's distribution. Also, data exchanges between multiple devices is automatically performed by the library. The skeletons supported are Map, Zip, Reduce and Scan, without nesting capabilities.

**Marrow [6, 7]** is a C++ skeleton framework focused on orchestration and execution of OpenCL kernels on multiple heterogeneous GPU devices, as well as introducing optimizations to the overall execution. Besides the typical data-parallel skeletons, like Map and Reduce, provided by most of the skeleton frameworks, it also provides support for some task-parallel skeletons like Pipeline and Loop (While and For). Another advantage over other libraries, is that Marrow offers support for skeleton nesting, allowing for more complex algorithmic structures to be specified. The parallel computations are defined and submitted as native OpenCL kernels, although, low-level functionalities like error-handling and memory management are abstracted from the programmer. This skeletons are encapsulated in an object called `KernelWrapper` allowing skeletons to easily access them when orchestrating an execution.

The Marrow framework takes advantage of the modern GPU's capacity to perform

simultaneous bi-directional data transfers between host and device, while executing computations and applies a technique called overlap between communication and computation. This technique reduces GPU's idle time by optimizing the scheduling and issuing of operations with memory transfers between the host and the device.

The execution model in Marrow can be viewed as a Master-Slave pattern, where a task is submitted for execution and the application is allowed to continue to perform additional computations. When an execution is requested, it is queued and an associated *future* object is created and referenced to the application. This object allows the application to query the state of the execution as well as waiting for the execution to finish.

**OpenACC [21]** is an open standard for parallel programming, resultant of the effort of a group of vendors to improve code portability between different implementations. This standard defines directives similar to those found in OpenMP, an already well-accepted standard for parallel programming with multi-core CPUs. There are several compilers implementations of OpenACC, both commercial (PGI [22] and CAPS [23]) and open-source (accULL[24]), but since they all obey to the same specification, the same code can be compiled with the different compilers with no need for code modifications.

**X10 [16, 17] and Chapel [18, 19]** are two programming languages that use the APGAS (Asynchronous Partitioned Global Address space) [25] model, an extension of the PGAS model. The PGAS model tries to create a level of abstraction over the global address space of a distributed system, making any data (local or remote) directly accessible by any process regardless of its location, without never completely suppressing the notion of local and remote data. By being aware of the data location, it is possible to reduce the performance problems associated with the GAS solutions, where the programmer was completely abstracted from the data locality. The APGAS, as the X10 team as defined it, introduces two new concepts: *places* and *asyncs*. A place is the abstraction of an entity on which computations are executed. A place can be mapped to an x86 core, a multi-core processor or even a GPU. Places are not required to be single-threaded and a place can be defined hierarchically, allowing for the exploitation of the hierarchical design of an architecture. A place can be stored in a variable and passed to functions as an argument. This model allows for explicitly requesting an execution to take place on a determined place. The APGAS model goes even further and allows for that execution to be asynchronous, through the async statement.

X10 is a programming language developed by IBM as a solution for clusters of multi-core CPUs and has later been extended to support GPUs. In X10, a GPU is represented by a subplace of the hosting place. The memory allocation on the GPU is done in a way similar to CUDA or OpenCL. For memory transfers between the GPU and the host, X10 provides an API that mirrors Java's `System.arrayCopy` (`Array.asyncCopy`), but it allows one of the arrays to be a remote reference and the copy is made asynchronously. The native X10 synchronization mechanism, clocks, is also usable inside a GPU to provide

a barrier mechanism to synchronize shared memory access. In practice, the programming of the computation offloaded to the GPU is not very distant from the abstraction level of CUDA or OpenCL. The programmer must explicitly express which asynchronous activities must be executed on GPUs. Also, in such cases, the execution of said activities is specified inside two nested loops, iterating over each block and then over each thread of the GPU, requiring the programmer to be aware of the adjacent execution model.

Chapel is a programming language developed by Cray that aims to provide a higher level of parallel programming while allowing the programmer to drop to a lower-level specification allowing for specific algorithm tunes to be performed. Unlike X10, the GPU is not represented as a subplace of the host, it is however, an integrated parted of the host's *locale* (a place in Chapel). When allocating memory in Chapel, the *dmapped* keyword is used, followed by the keyword to specify the domain where the memory is to be mapped. Allocating a variable on the GPU memory is achieved by using the *GPUDist* as the domain keyword. Data parallelism is achieved through the *forall* loop declaration. If a forall loop is applied over a variable mapped to the GPU, the compiler will automatically generate a CUDA kernel with the body of that loop and that will execute that computation on the GPU. The memory transmissions between the host and the GPU is implicit by default but can be set to explicit when the memory is mapped by using the *GPUExplicitDist* keyword. Chapel also offers explicit access to specialized memory spaces of the GPU, like shared memory, constant cache memory and texture cache, by simply mapping the memory with respective keywords. Besides the forall operator, Chapel also offers support for the higher level operators *reduction* and *scan*, allowing those operators to also be defined by the programmers. Although GPU computing in Chapel is much more abstract than in X10, the programmer still has to decide what must the executed in each processing unit.

**Lime [20]** is a programming language for hybrid computation, comprising architectures involving multi-core CPUs, GPUs and also FPGAs. It is a language inspired in the Java programming language and attempts to achieve maximum compatibility with Java, in a way that code compilable with the Java compiler can also be compiled with the Lime compiler. The reverse can also happen if the Lime code does not make use of Lime exclusive features that do not exist in Java.

Lime allows for the creation of workflows that can be described as a direct graph of computations where each edge represent a task and the output of each task is the input of the next task in the graph. That being said, the two most important operators in Lime are the connect operator (=>) and the task operator (*task*). Each task represents a computational unit, equivalent to an OpenCL kernel, while each connector denotes the flow of the data between two tasks. In Lime, a task can be either *isolated* or *non-isolated*. An isolated task (also called *filter*) is a task that can only access its own address space and has no access to the global state. The job of each task is to apply a method (called *worker method*) as long as there is still input to be processed, and enqueue the result into

an output stream. Also, by explicitly declaring the communication between tasks (using the connect operator), it is possible for the compiler to optimize and synchronize the communication between tasks automatically, without the programmers intervention.

Lime also has support for the map-reduce model. The map function is represented by the @ token and applies a function to each element of an aggregate data structure, returning another aggregate data structure. On the other end, by using the token ! after an operator or method, the Lime compiler will treat it as a combinator to execute a reduction, as long as the method applies the computations to two arguments of the same type and returns a result of that type.

The Lime compiler implicitly determines a partitioning of the program between the CPU and the GPU. For a task to be electable to be offloaded to the GPU, it must not be the first nor the last task in the stream and it must be an isolated task, guaranteeing that it does not perform globally side-effecting operations. The compiler also searches for map and reduce operations within each filter to identify kernel-level data-parallelism opportunities. For a map function to be compiled for the GPU, it has to be static and local, and the arguments must be value types, guaranteeing that the function is side-effect free.

### 2.2.3 Multi-backend with single or explicit offloading

**Muesli [26]** is a skeleton library that started with a focus on multi-node clusters and has been continuously adapted to support multi-core processors and more recently, GPU processors, combining MPI, OpenMP and CUDA. Although it supports both multi-core CPUs and GPUs, Muesli does not support the combination of CUDA with OpenMP. Also, the platform does not choose where the execution will take place, meaning that it has to be specified in the code whether the execution will take place in the CPU or in the GPU. Like most of the skeleton libraries, Muesli takes care of memory transfers implicitly, through lazy-copy mechanisms to reduce the communication overhead. Kernels are defined in a CUDA-like way, without no higher level abstraction. Muesli implements both Data (Map, Zip, Fold and Scan) and Task-parallel skeletons (Farm, Pipe, D&C and B&B), although, GPU offloading is only available for Data-parallel skeletons. The same goes for skeleton nesting, as the library supports skeleton nesting but since you cannot nest skeletons inside data-parallel skeletons, it can be concluded that skeleton nesting is not supported for GPU computations.

**Thrust [27]** is a C++ template library of parallel algorithms and data structures for CUDA. In Thrust, memory management is explicitly declared, although, it is greatly simplified by vector containers. When specifying a vector container, either a `host_vector` or a `device_vector`, the memory will be automatically allocated in the specific device. Also, deallocation and dynamic resize will be taken care by the library. Memory transfers can easily be specified by using the = operator or the `copy` function. Kernel launching is abstracted by Thrust's interface as the programmer only has to call the template function. Thrust already provides most of the built-in arithmetic and comparison operators to be

used in the template calls, but it is also possible for the programmer to declare his own. Thrust also provides an OpenMP and a TBB backend that can be specified during compilation, without any modifications to the code, although, two different backends cannot coexist.

**Bolt [4]** is also a C++ template library, developed by AMD on top of OpenCL. Bolt is in many ways similar to Thrust. Device and host memory is also explicitly declared through a vector container (`std::vector` or `device_vector`), providing a simpler way to manage memory than in OpenCL declarations. Unlike Thrust, however, if the data is allocated in the CPU memory space and the computation will take place in the GPU, the data will be automatically copied to the GPU memory. Besides OpenCL, Bolt functions can also be executed with a TBB backend or in a serial way in the CPU, although, and like Thrust, it provides no support for different backends to coexist. By default, Bolt functions will try to run computations on the GPU with OpenCL, if it fails it will fall back to TBB and then to serial execution.

**Aparapi [28]** is the result of an effort aimed towards providing GPU computations using the Java programming language, by converting Java bytecode into OpenCL. It also aims to maintain the Java's principle of "Write Once Run Anywhere", meaning that if a GPU is not available, does not support OpenCL or, for some reason, the OpenCL code cannot be generated, the computation will be carried out by a Java thread pool. This characteristic makes Aparapi an interesting platform since code suitable for the execution on GPUs can also be executed on CPUs. Although, the sections of the program that are meant to be run on the GPU have to be explicitly declared meaning that the platform has no mechanism to implicitly schedule the working load to the different processors. Also, Aparapi offers no abstraction level over the kernel code. The code is written in Java, but the abstraction level is the same of an OpenCL kernel, as the programmers are aware that they are writting the code with the partial vision of each thread.

**Bones [15]** is a source-to-source compiler in which the compilation is achieved through algorithmic skeletons. In Bones, algorithms are classified into different classes and when the programmer is declaring a pragma to signal parallelizable code, it defines the class in which the algorithm falls into. This will allow the compiler to merge the programmer defined code with the generic code of the skeleton for that class. The resulting code already takes care of memory allocation and data transmission between devices. Bones has no support for multiple GPUs nor skeleton nesting. Supported backends are CUDA and OpenCL (both for GPU and CPU), although, not all the skeletons supported by the CUDA backend are supported by the OpenCL one.

**A Flexible Shared Library Profiler for Early Estimation of Performance Gains in Heterogeneous Systems [29]** is a work that addresses heterogeneous architectures through

a shared library interposing technique, that replaces shared library calls, by calls to a wrapper library that will choose among alternative implementations of the shared library, based on a performance prediction model. Given that the wrapper library and the alternative implementations have the same ABI (Application Binary Interface) of the original library, one of the advantages of this technique is that it does not require any modifications to the original application, nor to the original library.

The framework starts by analyzing the application's executable file in order to identify the shared library calls. It then generates a wrapper library that calls the original shared library functions and traces all the calls to each function while collecting profiling data. The next step is to execute the application with the wrapper library preloaded, for a significant number of operating conditions, so the profiling information can be collected. This profiling information is then combined with the performance models for each of the alternative implementations, allowing for a performance prediction during runtime, depending on the operating conditions like the input work size.

The downsides of this model are the need for the existence of different shared library implementations, and the fact that only one implementation is selected to execute, not taking advantage of possible performance gains by distributing the work-load between different implementations (that execute on different devices).

### 2.2.4 Automatic Backend Selection

**SkePU [3]** is a skeleton library that was originaly built with multi-core CPUs and multi-GPUs in mind. Like Muesli, memory transfers are achieved through lazy-copying mechanisms and memory locality is abstracted through a vector interface. The library supports both OpenCL and CUDA backends, for GPU executions, as well as an OpenMP backend for multi-core executions. All backends have the same interface, meaning that the programmer does not have to be aware of where the execution will take place while programming. Kernels are also not explicitly declared, instead, the programmer only defines the function to be applied by the skeleton via a small set of predefined C macros.

This framework has recently evolved to integrate automatic backend selection support[30]. Although there is no cooperation between the different devices, as only one implementation is selected to perform an execution, the control of which backend to offload an execution is no longer left upon the user. Instead, the platform implicitly determines the best performing device, based on the received data size and the performance information gathered during an offline training.

SkePU's offline training is based upon the premise that if an implementation is the best performing for a data size $i$ and for a data size $j$, then, all data sizes between $i$ and $j$ will also perform best with said implementation. The training algorithm starts by evaluating the best performance for the lower and the upper bound data sizes (default or user-specified). This information is stored on the root node of a tree. The remaining of the tree is then constructed by recursively dividing each dimension's subspace, until equal

winners of a subspace are found, considering the respective leaf node, a *closed node*. The training terminates when all the leafs of the tree are closed, or upon reaching a maximum tree depth or training timeout, specified by the user.

In parallel to the training project, there's a parallel project aiming to integrate SkePU with StarPU [31]. StarPU is itself a heterogeneous computing library that pays close attention to the efficient scheduling of tasks through the several available CPUs and GPUs, which combined with SkePU, gives the programmer the abstraction level of the skeletons as well as a dynamic scheduling of the tasks, resulting in lighter burden over the programmer and more efficient final results. However, there is still no work-load distributions among different devices, meaning that device concurrency is only possible with the submission of multiple tasks. SkePU only has support for data-parallel skeletons like Map, Reduce and some variants of those two and does not support skeleton nesting.

**StreamIt [32, 33]** is a programming language that was developed as a response to the crescent increase of parallel streaming applications. It attempts to provide the programmers with a more natural way to define a stream, while it performs stream-specific optimizations, achieving levels of performance close to optimized code written in a lower-level language. The basic unit of computation in StreamIt is the *filter*, which represents a computation that is executed over a stream of data. Each filter holds two types of communication channels, the input by which the data is received, and the output, where the results of the computations are sent to. A stream can be represented by a hierarchically organized graph and the constructs can be of three types: *pipelines*, *split-joins* or *feedback loops*. The pipeline is the most basic construct where each filter is connected to the next one. In a split-join construct, the flow of the data is divided into independent substreams by the *splitter* and later is rejoined by the *joiner*. A splitter can either apply each element to every of its substreams, to allow for different filters to be applied over the same data but it, can also apply each element to only one substream, allowing for weight distribution. A feedback loop allows for the existence of cycles in the stream flow graph.

StreamIt is another example of a language that has evolved to support GPU computations, although in this case, only the compiler was modified to be GPU aware. The stream application can be represented as directed acyclic graph (given that in this version, some advanced features like feedback loops are not supported). The platform processes the stream graph through a coarsening and a uncoarsening phase, grouping nodes (*filters*) into partitions, taking into account performance and communication time estimations, while also taking advantage of the on-chip GPU shared memory, in the most efficient way.

After the partitions are created, they are distributed among the present GPUs, with the exception of the partitions that maintain an internal state, that are pre-mapped to the CPU. Besides this exceptions, the CPU is used to control the GPU's execution (one dedicated thread per GPU device). The framework internally manages communications between partitions inside each GPU device, as well as inter-GPU communication.

**Adaptive Runtime Selection for GPU [34]** is a work addressing online performance prediction, based on an offline profiling stage. This solution consists in converting C source code, identified by pragmas, to CUDA code, through a source-to-source polyhedral compiler. During this code conversion, various optimizations are performed, like identifying data reusability, to take advantage of the GPU's shared memory. When the CUDA code is compiled, various versions of the code are generated, differing in some parameters, like tile and block sizes.

The next step is to perform an offline profiling stage. This profiling stage is divided into two stages. During the first stage, data communication between the host and the GPUs is evaluated, building a table that matches message sizes with the expected communication times. The second stage consists in the simulation of the different generated versions of the CUDA kernel, under different execution contexts.

Finally, at runtime, the framework estimates the total execution time of each kernel implementation, considering the execution and communication information gathered during the offline profiling stage. The version with the best predicted global execution time is selected to run. Simultaneously, the same computation is prompt at the CPU as well. When one of the executions finishes, it signals the other one to interrupt its execution. Although this solution performs concurrent executions in the CPU and GPU devices, there is still no cooperation between the devices, as the results of one of the executions will be discarded.

**Accelerating MapReduce on a Coupled CPU-GPU Architecture [35]** is a work that addresses the execution of MapReduce computations in nodes comprising CPUs with integrated GPU. This work proposes two approaches for using both these devices in a cooperative way: one where both the Map and the Reduce stages are executed by both processors (*Map-dividing Scheme*), and another, where each of those stages is executed in a different device (*Pipelining Scheme*). Both these devices rely upon a dynamic work distribution approach and implement continuous reduction.

In the *Map-dividing Scheme*, the scheduler is responsible for dispatching the input data to each worker. In a GPU, a worker corresponds to a workgroup. In the CPU, each core is a worker, except for the first one that is responsible for the dynamic schedule, based on the master-worker model. Inside each worker, the input data is divided among the working threads to execute the map function. Each worker has also a reduction object, responsible for the partial reduction of the output of each of its belonging thread. When the end message is received by the workers, one reduction object per device will perform a reduction of the reduction objects of its workers. In the end, these intermediate results are combined to produce the final result.

The *Pipelining Scheme* is based on a producer-consumer model. In this case, the map device performs the map function over the input data, in the same fashion as it happens in the *Map-dividing Scheme*, but instead of local reduction objects, the output of the map function is placed on a circular buffer (one for each worker), placed in the zero copy

buffers, as they are accessed by both devices. The reduction device is also composed by multiple workers, that perform a partial reduction function. When the scheduler identifies an idle reduce worker, it will assign it a full key-value block from one of the circular buffers. At the end, a global reduction object will reduce the results of all the partial reduction objects.

There is also a static load balancing approach for this scheme. This approach eliminates the need for a scheduler and therefore, all the CPU cores will work as workers. The input is evenly distributed among the map workers and the output is stored on the circular buffers (also one for each worker). Since there is no scheduler, the key-value blocks present in the circular buffers are evenly distributed among the reduction workers.

**Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems [36]** continues the work presented in [29], described previously. In this work, the authors extend the previously proposed framework, so, when not limited by the data/control dependencies, the system automatically distributes the work-load among different devices (through different library implementations).

Unlike their previous work, this works follows a dynamic performance model construction during runtime, instead of during an offline profiling. The task scheduling follows one of two policies: the *Best Performance Selection* policy, when the working data cannot be divided among different devices; and the *Load Balancing* policy, when work partitioning is possible.

In the *Best Performance Selection* policy, when the best performing implementation is not known for the given work size, the performance model is constructed by executing the application using all the implementations (simultaneously). The following executions will only be executed by the fastest implementation. Every time the application is executed, the performance model is updated so it may adapt to eventual alterations in the running environment, such as a system's load variation.

The *Load Balance* policy follows a load balancing approach that tries to balance the work-load in a way that all executing devices finish their execution at the same time. This process is also dynamic and based on a Functional Performance Model principle. For the first iteration, since there is no performance information, the work-load is evenly distributed among the devices. In the following iterations, the FPM of each device is updated and a new load distribution is calculated, gradually getting closer to the ideal work distribution.

**Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems [37]** is a solution for transparent orchestration of CPU and GPU OpenCL computations. They address device heterogeneity by executing a different number of work-groups in each device to achieve work-load balance. The work partitioning is performed by a dynamic compiler composed by three modules: the *kernel transformer*, the *buffer manager* and the *partitioner*.

The kernel transformer is responsible for converting the original kernel, preparing it to work only over a subset of work-groups. After this step, the buffer manager analyses the memory access patterns of each work-group. If the memory accesses can be determined statically, only the necessary data is transferred to each device. Otherwise, the input is replicated through all the devices and the output will be merged. The partitioner is responsible for deciding the number of work-groups to execute in each device, based on profiling information. That profiling information is gathered in a single time basis, by launching executions with s different number of work-groups to each device, while gathering performance results.

## 2.3 Skeletons and Template-based Frameworks for Multi-core Computing

In this section, we will visit some multi-core framework solutions that use algorithmic patterns.

**Skandium [38]** is a Java algorithmic skeletons framework (ASkF). It stands as a re-implementation of Calcium [39], a Java ASkF for cluster environments, but targeting multi-core computers instead. Skandium provides support for both Data (Map and Fork) and Task-parallel skeletons (Farm, Pipeline, for and while) as well as the resolution skeleton D&C. Skandium also allows for skeleton nesting. Besides skeletons, there is another important concept called *muscle blocks*. Muscle blocks are sequential blocks of code that provide the logical needed to transform a general skeleton into a specific application. Muscle blocks can be viewed as black boxes that are invoked by the skeletons and depending on the skeleton they can be executed either sequentially or in parallel.

In Skandium, all the communications between devices are implicit, however, it does not provide any higher level shared-memory protection mechanism, meaning that the programmer has to be aware that the muscle blocks may execute concurrently, and take the right measures to avoid race conditions, guaranteeing the correct results. This may have an impact on performance, since a part of the program has to be serialized and it is up to the programmer to identify the critical section.

Skandium offers a dynamic scheduling, based on a producer/consumer scheme. The initial task is inserted into a task queue. Multiple threads will consume and compute tasks from the ready queue. A task can generate sub-tasks that will be inserted into the task queue, and can be kept in a waiting state until all its generated subtasks have finished executing, allowing for other tasks to be executed. When the task exits its waiting state, it is re-introduced into the task queue to be scheduled to continue its computations.

**Intel Threading Building Blocks [40]** (TBB) is a C++ template library developed by Intel for parallel programming on multi-core processors. Although it does not label itself as a skeleton framework, its pattern-based abstraction approach has evident similarities

| | GPU | | | | CPU | | GPU/CPU | |
|---|---|---|---|---|---|---|---|---|
| | SkelCL | Bolt | Thrust | Marrow | Skandium | TBB | Muesli | SkePU |
| Map | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ |
| Reduce | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ |
| Zip | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ |
| Scan | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✓ | ✗ |
| Fork | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ |
| Pipeline | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | CPU | ✗ |
| Farm | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | CPU | ✗ |
| For | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| While | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✗ |
| D&C | ✗ | ✗ | ✗ | ✗ | ✓ | ✗ | CPU | ✗ |
| B&B | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | CPU | ✗ |
| Sort | ✗ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ |

Table 2.1: Skeleton support by analyzed frameworks

with skeleton programming. TBB offers both implicit and explicit parallelism. Explicit parallelism is based on the notion of a task, a sequential blocks of code that can be executed in parallel with other tasks. A task may spawn additional tasks, allowing the creation of complex task hierarchies with specified task dependencies. For more common types of parallelism, TBB provides a set of templates (For, Reduce, Skan, While and Sort) that offer the same pattern-based abstraction as skeletons. These templates are implicitly converted into tasks.

All tasks are executed through a dynamic scheduler based on local task queues and task stealing, a scheduling strategy popularized by Cilk[41]. Each working thread has its own task queue, avoiding race conditions to a global task queue, maximizing concurrency. The problem with local task queues is that the work distribution between threads may be unbalanced, reaching a point in the execution when some threads become idle, waiting for other working threads to finish computations, not fully exploiting the parallelism. To work around this weakness, TBB implements a task stealing mechanism. When a working thread has no more tasks in its task queue, it selects a random working thread and tries to steal a task from it. Although this mechanism will keep more threads busy, ultimately improving performance, the random nature of the victim thread selections may not select the best victim to steal from, and since each steal will introduce an overhead on the execution, it would be desirable to steal a task from the busiest thread, reducing the overall number of steals.

## 2.4 Summary

Table 2.1 summarizes the skeletons implemented by the skeleton frameworks analyzed in this chapter.

Some of the GPU skeleton frameworks, like skeletons, are based on OpenCL, which means that they are able to execute on CPUs as well. However, given its GPU focus, there

is no reason to do it from a performance point-of-view, as all the framework was design with only OpenCL GPU executions.

Concluding, the current state of the art around the abstraction of a system composed by CPUs and GPUs as a whole is still very preliminarily. In the context of CPU and GPU skeleton frameworks, we did not find any proposal with the characteristics of the solution we are proposing.

# 3

# The Marrow Algorithmic Skeleton Framework

The work presented in this document aims to extend Marrow [6, 7], to offer support for CPU OpenCL computations along with the current support for multiple heterogeneous GPUs. As introduced in Section 2.2.2, Marrow is a C++ algorithmic skeleton framework that stands out for its focus on kernel orchestration and communication, rather than just focusing on the partitioning and distribution of data, like most of the current skeleton frameworks for GPU programming. This chapter presents the latest version of the framework that includes support for multiple heterogeneous GPUs.

Marrow's architecture is divided into four layers: *User Applications*, *Skeleton Library*, *Runtime* and *OpenCL Enabled Devices*, as shown in Figure 3.1. The flow of the communication goes downwards, meaning that each layer only has vision of itself and the layer immediately bellow. The upmost layer is the User Application layer and it represents the C++ applications that make use of Marrow's skeletal API. The downmost layer symbolizes the OpenCL enabled devices where the kernel computations are to be run, in this case, GPU devices. A more comprehensive description of the remaining two follows.

## 3.1 Skeleton Library

The skeleton Library encloses all the components of the Marrow framework that are accessible to the programmer. It includes the implementation of each skeleton offered by the platform, the *Kernel Data-types* and the `Vector` and `KernelWrapper` classes. The Kernel Data-Types enable the specification of a kernel regarding its type, size, whether or not it is partitionable (for data partitioning purposes) and the minimum size for a

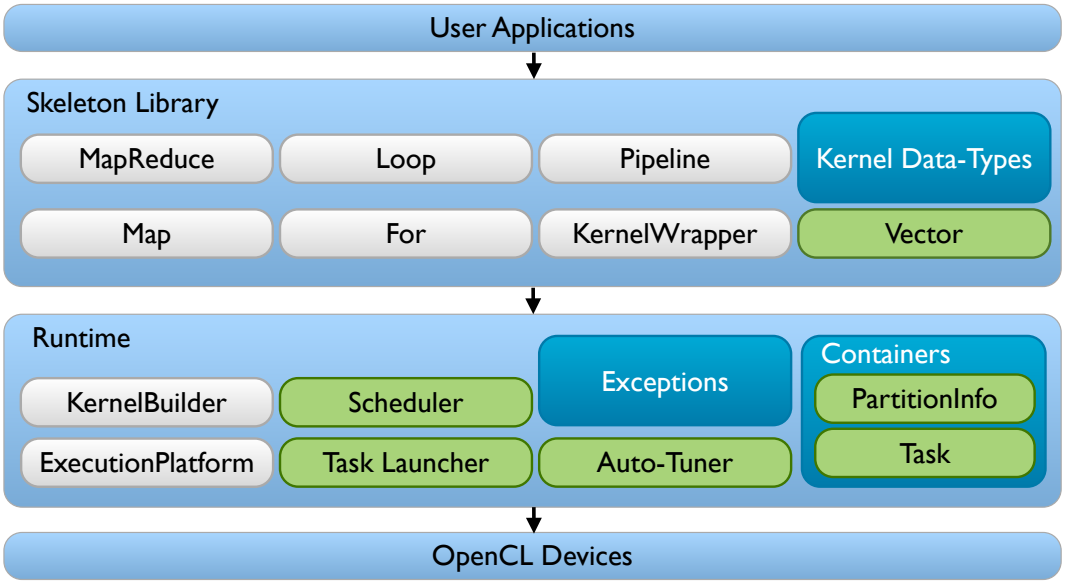


Figure 3.1: Marrow's System Architecture (taken from [7])

partition (*indivisible size*). The currently supported data-types are:

**BufferData** represents a contiguous memory region;

**FinalData** a constant single-element data defined on its creation and valid for all the executions;

**SingletonData** a single-element data but the value is defined for each execution;

**LocalData** represents a memory region allocated in the GPU memory.

The `KernelWrapper` object is an enclosure for an OpenCL kernel to be executed. The `Vector` represents a contiguous memory region and wraps any data buffer to be submitted to a skeleton, concealing any intrinsic memory management, namely data-partitioning, memory allocation and data-transmission between the host and the OpenCL devices.

### 3.1.1 Skeleton Nesting

When Marrow was introduced in Section 2.2.2, we highlighted the feature that allows the creation of structured computational trees (exemplified in Figure 3.2) by nesting skeletons inside each other, defining a complex behavior comprising the behavior of all the nested skeletons. The nodes of this trees can be divided into three categories: *root nodes*, *inner nodes* and *leaf nodes*.

In Marrow, a computational tree is composed by: exactly one root node, at least one leaf node and any amount of inner nodes. Any of the skeletons supported by the framework can become the root node of the computational tree but only the `Pipeline`, `Loop` and `For` skeletons are eligible to be inner nodes (the explanation follows in the

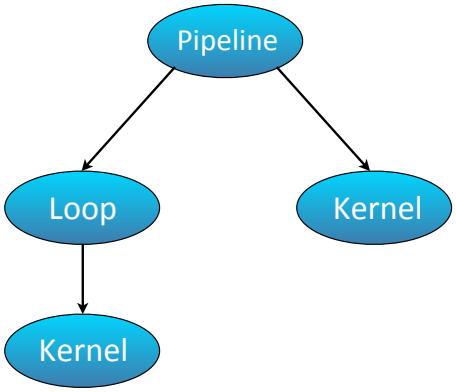


Figure 3.2: Example of a computational tree (adapted from [42])

Section 3.1.3). Since the skeletons only define an algorithmic behavior but they do not own any computational logic by themselves, the leaf nodes must be instances of a `KernelWrapper` object as they encapsulate the computational logic of the OpenCL kernels. The library regulates this restrictions through two interfaces: the `ISkeleton` and the `IExecutable`. The `ISkeleton` interface is implemented by all the components eligible to be a root node (all the skeletons). All the nestable nodes (inner and leaf nodes) must implement the `IExecutable` interface.

### 3.1.2 Multi-GPU support

In Marrow, the support for multiple heterogeneous GPU devices is achieved through a transparent decomposition of the application's data-set, based on the performance values of each individual device. This values are collected during the installation time and are used to calculate a performance ratio among the GPU devices present in the system.

Given that the input and output data sizes are specified during the creation of the computational tree and that those sizes do not change between skeleton submissions, the data is statically partitioned also during the initialization process and it stays valid for all the upcoming executions. Due to the data dependencies existent in some applications, Marrow allows the programmer to specify an *indivisible size* for each partitionable data type, hinting the framework to create partitions with a size multiple of that value. Other applications may require full access to a data-set. In this case, the full data-set must be copied to every device but each device will work over a different partition of that data. Listings 3.1 show the constructor of the `BufferData` type. By default, the `indivisibleSize` is 1 and the `partitioningMode` is set to `iWorkData::Partitionable`, however, the programmer can override this values, for example, set the `partitioningMode` to `iWorkData::Copy` to force a copy of the full data-set to every GPU device.

Marrow takes advantage of multiple GPUs through a data-parallel work distribution, meaning that the computational tree is replicated to every GPU device and it is fully executed over the assigned subsets of the input data, like depicted in Figure 3.3. This

Listing 3.1: BufferData constructor

```
1 BufferData(unsigned int numberOfElements,
2   IWorkData::partitionMode partitioningMode,
3   unsigned int indivisibleSize);
```

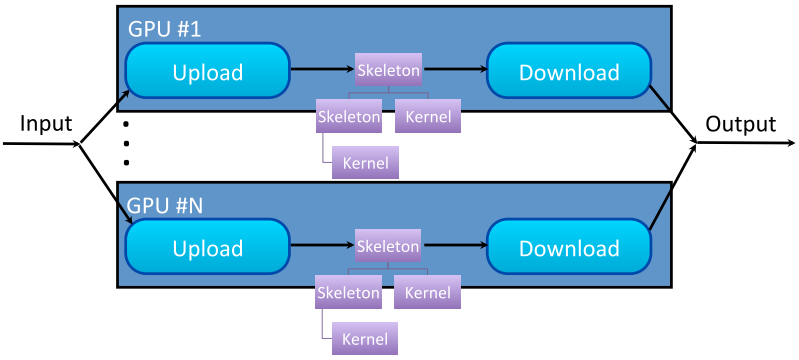


Figure 3.3: Computational tree replication (taken from [42])

allows the framework to take advantage of the data locality inside each device, reducing the data communication between the host and the GPU devices to the initial input upload and the final output download. Since the work-load is balanced based on the relative performance of each GPU, an efficient work-load balance is achieved prior to the tree's execution, eliminating the need for work-load transfers during runtime.

### 3.1.3 Supported Skeletons

Marrow currently supports the *Map*, *MapReduce*, *Pipeline*, *Loop* and *For* skeletons.

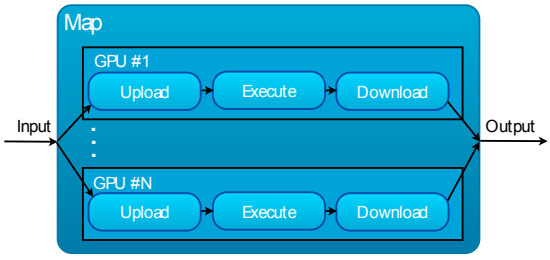


Figure 3.4: Map skeleton execution (taken from [42])

**Map** (Figure 3.4) is a data-parallel skeleton that applies the same computation over a given data-set. This skeleton does not introduce a new behavior to the computational tree but it plays an important role in a computational tree as it allows the framework to apply the work distributions among devices and overlapping partitions. Therefore this skeleton is useful to prompt the execution of a single kernel.

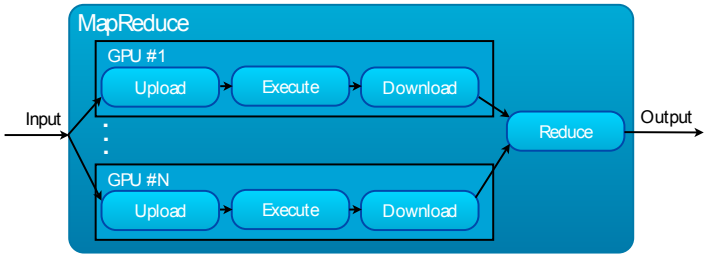


Figure 3.5: MapReduce skeleton execution (taken from [42])

**MapReduce** (Figure 3.5) is similar to the `Map` skeleton but it executes an additional computation over the output of the map resulting data, outputting a smaller data-set, usually a single data element. Since most of the reduction functions are hardly parallelizable and therefore inefficient to be executed on a GPU device, the `MapReduce` skeleton can be initialized in two ways: The programmer can either specify a C++ function that will be executed on the CPU after the `Map` execution has finished, or he can provide a reduction OpenCL kernel and the `MapReduce` skeleton will be initialized as a `Pipeline`.


Figure 3.6: Pipeline skeleton execution (taken from [42])

**Pipeline** (Figure 3.6) allows the programmer to specify a series of data-dependent stages where the output of one stage will be the input of the next one. In Marrow, the parallelism is achieved by issuing the execution of all the stages in all the present GPU devices, but over different partitions of the data (like detailed in Section 3.1.2). By doing so, there is no need for data communication between stages and the data communication between the host and the device is reduced to (a) the upload of the input of the first stage and (b) the download of the output of the last stage. Although this skeleton only supports two stages, by making use of the nesting mechanism, Pipeline skeletons can be nested inside each others to achieve the desired amount of stages.

**Loop (and For)** skeleton applies iterative computations over an input data-set. Due to the nature of a loop cycle, the condition that evaluates if the cycle should continue or break may be dependent on the full resulting data-set of one iteration. Since Marrow splits the global data-set among the various present devices to be independently processed, in cases where the computation requires access to the full data-set to evaluate that condition, a synchronization step has to be introduced between iterations. Since this

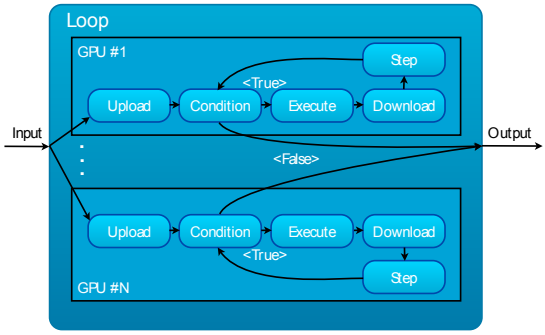


Figure 3.7: Loop skeleton with parallel step computation (taken from [42])

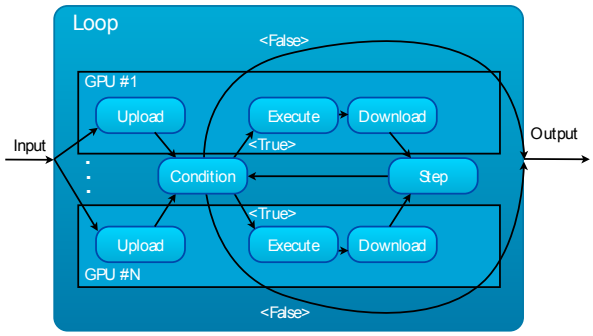


Figure 3.8: Loop skeleton with synchronized step computation (taken from [42])

step is certain to introduce an overhead in the computation (mainly because of the required communication), this skeleton can be instantiated in two ways: with a *parallel step computation* (showed in figure 3.7) or with a globally *synchronized step computation* (figure 3.8) when full synchronization cannot be avoided.

### 3.1.4 Programming example

In this section we present a programming example (in Listing 3.2) of a three stage image filter pipeline (one of the benchmarks we used for this work's evaluation) to illustrate the steps a programmer would take to instantiate a computational tree (from line 2 to line 34) and submit an execution request (from line 37 to line 47).

Lines 2 to 4 define the size of each dimension of the `globalWorkSize`. Since these kernels operate over two pixels of the same line of the input image, the first dimension of the `globalWorkSize` is set to half the width of that image. From line 8 to line 19 the input and output data-types for the first kernel are configured and the `KernelWrapper` object is created. The other two `KernelWrapper` objects are configured in the same fashion, from line 21 to line 30. In lines 32 to 34, the computational tree is initialized by nesting a `Pipeline` skeleton inside another. Since the `p2` pipeline is the root node of the tree, the number of GPUs and the number of overlapping partitions (`numBuffers`) are specified to overlap the default values.

The execution request stage starts with the creation of a `Vector` object for each input

and output data-type (line 37 to line 44). The execution is then submitted to the skeleton tree (line 46) and a `Future` object is returned. This object will be used to inform the application when the execution has finished.

## 3.2 Runtime System

The Runtime System of Marrow contains the modules that aggregate and export OpenCL functionalities that are used recurrently by the upper layer. The `ExecutionPlatform` is responsible for an OpenCL environment, shared by all the nodes in the execution tree. Since only GPU OpenCL devices are considered in this version, there is only one type of `ExecutionPlatform`, the one responsible for all the devices. `KernelBuilder` is responsible for managing all the stages associated with kernel compilation for all the present GPU devices. `OpenCLErrorParser` works as an interpreter for OpenCL error codes, transforming them into more representative C++ exceptions, thus providing a stronger error handling system, both internally and to the developer. A `Task` object is a container for all the information regarding a single submission to a computational tree. A `PartitionedInfo` object contains the partitioning information of a single argument of a kernel execution. The `Autotuner`, `Scheduler` and `TaskLauncher`, along with the `ExecutionPlatform`, constitute the persistent components of the platform, meaning they stay the same for all the computational tree's inside one application, allowing for a global scheduling of the available resources. `Autotuner` is the component that generates the partitioned information for all the arguments of a kernel during the creation of the skeleton tree. The `Scheduler` is responsible for receiving task submissions and enqueue them to the queue of each GPU device so the `TaskLauncher` can consume them and prompt the execution on the respective device.

### 3.2.1 Execution Model

Marrow's execution model can be divided into two stages, the *Skeleton Initialization*, performed only once when the computational tree is initialized, and the *Skeleton Execution Request*, performed every time an execution is submitted to the computational tree.

**Skeleton Initialization** is the stage when the computational tree configures itself to be ready to receive execution submissions. When the programmer initializes the computational tree, including the `KernelWrapper`, the data-types definitions (with respective data-sizes) and the skeletons, he may also specify the number of GPUs, as well as the number of overlap decompositions to be used (otherwise the framework will use all the GPU devices available and set the overlap decompositions to its default value). That information will serve as the configuration for all the submitted execution requests. Figure 3.9 describes the steps taken during this stage. When a `Skeleton` is initialized (step 1),

Listing 3.2: Image Pipeline implementation in multi-GPU Marrow

```
// Stage 1: Computation tree configuration
std::vector<unsigned int> globalWorkSize(2);
globalWorkSize[0] = uiImageWidth/2;
globalWorkSize[1] = uiImageHeight;

std::vector<std::shared_ptr<IWorkData>> inputData(2);

inputData[0] = std::shared_ptr<IWorkData> (new BufferData<cl_uchar4>(
  uiImageWidth * uiImageHeight, IWorkData::Partitionable, uiImageWidth));

inputData[1] = std::shared_ptr<IWorkData> (new FinalData<int>(factor));
std::vector<std::shared_ptr<IWorkData>> outputDataInfo (1);

outDataInfo [0] = std::shared_ptr<IWorkData> (new BufferData<cl_uchar4>(
  uiImageWidth * uiImageHeight, IWorkData::Partitionable, uiImageWidth));

std::unique_ptr<IExecutable> gaussKernel (new KernelWrapper(
  gaussNoiseKernelFile, "gaussian_transform", inputData, outputData,
    globalWorkSize));

inputData[1] = std::shared_ptr<IWorkData> (new FinalData<int>(threshold));

std::unique_ptr<IExecutable> solariseKernel (new KernelWrapper(
  solariseKernelFile, "solarise_transform", inputData, outputData,
    globalWorkSize));

inputData.resize(1);
std::unique_ptr<IExecutable> mirrorKernel (new KernelWrapper(
  mirrorKernelFile, "mirror_transform", inputData, outputData,
    globalWorkSize));

std::unique_ptr<IExecutable> p1 (new Pipeline(gaussKernel, solariseKernel));
std::unique_ptr<IExecutable> p2 (
  new Pipeline(p1, mirrorKernel, numDevices, numBuffers));

// Stage 2: Execution request
std::vector<std::shared_ptr<Vector>> inputData(1);
std::vector<std::shared_ptr<Vector>> outputData(1);

inputData[0] = new Vector(
  input, sizeof(cl_uchar4), uiImageWidth*uiImageHeight);

outputData[0]  = new Vector(
  output, sizeof(cl_uchar4), uiImageWidth*uiImageHeight);

IFuture *future = p2->write(inputData, outputData);
future->wait();
```

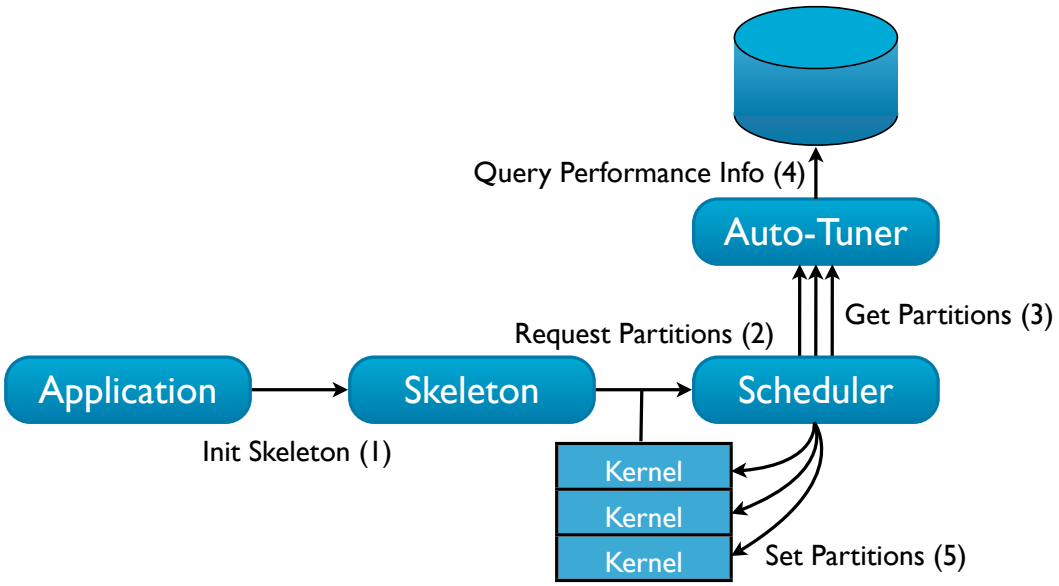


Figure 3.9: Skeleton Initialization (taken from [7])

it requests the `Scheduler` component for the partition information for all the kernels included in the computational tree (step 2). The `Scheduler` will use the `AutoTuner` component to calculate the partition information (step 3), based on the skeleton configuration arguments mentioned before, as well as GPU performance information (step 4), obtained previously for the underlying architecture where the program is running, allowing for a better work balance when heterogeneous GPU devices are present. Finally, the scheduler configures all the `KernelWrapper` objects with the partitioned information obtained (step 5). Since this process is only executed one time, the data decomposition will always follow the same pattern in all the upcoming execution requests. Such approach limits the tree's applicability, as all the execution requests must respect the data-sizes specified in this stage.

**Skeleton Execution Request** (depicted in Figure 3.10) is the stage when the computational tree receives an execution requests, prompts its executions to the GPU devices and returns the results back to the hosting application. This stage is triggered by the arrival of an execution request on the root node of the computational tree (step 1). The skeleton will create a `Future` object associated with that submission (step 2) and return it to the application (step 3) so it can be notified as soon as the execution finishes and the results are ready. The task is then submitted to the scheduler (step 4) that will enqueue multiple task submissions to each GPU device's task queue (step 5) (one for each overlap partition), associating a different portion of the data-set to each submitted task, according to the partitioning process performed in the initialization stage. The `TaskLauncher` is the component responsible to prompt the executions to the devices. To achieve the desired parallelism, the `TaskLauncher` is running one thread for each overlapping partition

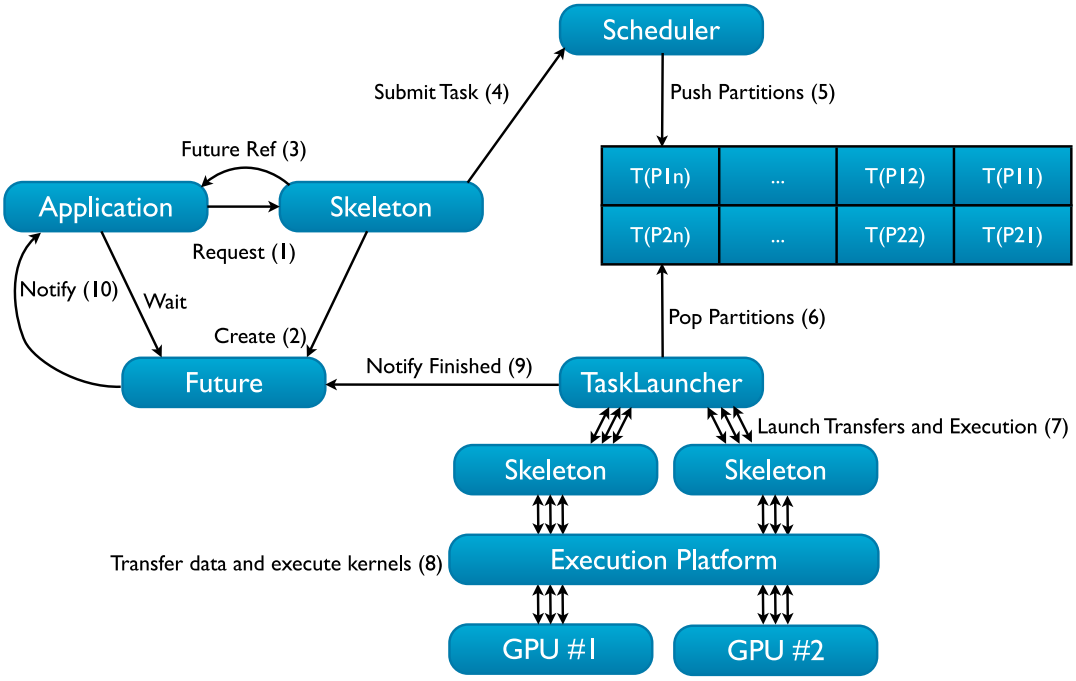


Figure 3.10: Skeleton Execution Request (taken from [7])

(number of devices X number of overlapping partitions) that will wait for tasks to be enqueued in their device's task queue. When a task is enqueued by the `Scheduler`, a free thread will dequeue it (step 6) and prompt the execution of the assigned partition on the assigned GPU (step 7), i.e, upload the input data to the GPU, prompting the execution of the computational tree and downloading the output data (step 8). By having multiple threads for each device, multiple executions will be prompt simultaneously to the same device, originating the desired overlapping effect. The `TaskLauncher` keeps track of all the subtasks in execution and when all the subtasks of a task have finished executing, it notifies the associated `Future` object (step 9) which, in its turn, notifies the hosting thread (step 10).

# 4

# Integrating the CPU in the Marrow Execution Model

The main goal of this work is to extend the Marrow framework [6, 7] from multi-GPU to provide support for a heterogeneous environments, composed by multi-CPUs and multi-GPUs, in a way that the optimal work-balance computation would be achieved without the programmer's involvement. This chapter highlights the modifications and improvements to the Marrow framework in order to accomplish our goals.

## 4.1 General Overview

In the previous version of Marrow, described in Chapter 3, the static distribution of the work-load already took into account the differences in performance among heterogeneous GPU devices. However, that distribution was accomplished with the assumption that GPU's architectures are similar enough that their performance can be compared simply by comparing performance measurements of each device, reaching a performance ratio between devices that would fit all applications. Although this method allows the framework to achieve pretty accurate work distribution among GPU devices, it falls short when the CPU enters the scene. The architectures of CPU and GPU devices are too distinct from each other to assume such preposition. Also, the use given by the operating system to each device is quite distinct. The GPU is a dedicated device that executes one task without interruptions, while the CPU is usually time-shared among multiple threads, meaning that the performance of a task is influenced by the system's overall work-load.

The huge design differences between CPU and GPU devices make it hard to find a

theoretical method to balance the work-load that would fit every application. Therefore, our approach was to devise a method that partitions the application's work-load based on the performance of each device, for that specific application, in a way that it stays transparent to the application itself. Recent work addressing CPU and GPU environments (analyzed in Section 2.2.4) approach this issue by performing a training session before actually executing the computation. Some of these solutions perform training executions with different input data sizes to determine, at execution time, to which device a computation should be prompted, based on the submitted input size and the collected information of the training. Our goal differs from cited works as our aim is to find the proper balance between devices and not which device performs better with a given input. Nonetheless, we also built upon a training approach, in order to find the desired efficient work balance between devices.

As detailed in Section 3.2.1, the scheduling in Marrow was static, meaning that the skeleton tree was partitioned upon creation. Executions with data-sets of different sizes required the tree's recreation. To fully exploit the underlying hardware, we altered the framework to allow for the work-load to be partitioned at execution request time instead. This modification not only lifts the restrictions on the size of the data-sets submitted upon a skeleton, as it avoids application errors due to size incompatibility of the configured data-types and the submitted `Vector` objects. Also, by delaying the partitioning to execution request time, the platform can now reconfigure itself before executing each request, which allows the framework to test different work distributions during a training session, and to accept input with different sizes without the need for an explicit computational tree recreation.

The work-load distribution process comprises two stages:

1. A computationally heavy, offline training process that tests different work distributions under different platform configurations, to find the configuration/work-balance combination that shows the best performance;

2. A lightweight online monitoring process that evaluates the performance of each device after each execution, detecting performance discrepancies and rebalancing the work-load for subsequent executions.

The training process (that is described in detail in Section 4.5.1), consists essentially in evaluating the performance of different work-load configurations and picking the one that performs better. This process, as a whole, may be computationally heavy, as such, it is supposed to be executed only once per application (during the first execution), being the cost justified and amortized by the efficient execution of the upcoming requests. Moreover, by having this "off the record" step before the actual execution starts, the framework can also test different values for parameters that were previously stated by the programmer. An example is the specification of the number of overlap partitions, whose ideal value depends of several factors, such as the amount of data to send, bus

contention, the weight of the computation, making the ideal values for this parameters, hard to predict without empirical knowledge.

As previously stated, a particularity of the CPU devices is that their performances will depend on the current load of the system. Consequently, if the load of a CPU device increases (or decreases) over time, the configuration found during the training process may not be the best for the current conditions. With that context in mind, we prepared the framework to constantly monitor its executions and detect excessive performance divergences between the devices. When these divergences are met, the platform tries to rebalance the load in a lightweight fashion. If it fails to find its balance after a number of attempts, the platform will compute a lighter version of the offline training to reestablish the balance under the current system's conditions.

When lifting the restriction over the input data sizes, we opened the door for applications to submit input data with different sizes over the same computational tree. If the amount of different data sizes is known to the framework user, he can choose to perform a training with each of those input sizes. All the trained data sizes are saved by the platform for future use. However, training a large number of input sizes can be unpractical, due to the amount of different work sizes (that can be unknown) or due to the excessive time consumed to perform different training executions. In this sense, when a skeleton receives a submission with an input size different from any data size previously received, the work distribution will be derived from the input size's partitioning information from previous iterations. This new work distribution information is also stored by the platform for future use, continuously populating the *Knowledge base* of the application. A derived work distribution may not have the same performance as a trained one, but they're performance is increased after some rebalance iterations.

## 4.2 Skeleton Library

The Skeleton Library layer (as presented in Chapter 3), suffered some modifications, motivated by the added dynamism we intended to implement in this version of Marrow. Since the platform is no longer restricted by the input data sizes specified during the computational tree's configuration, the *Kernel Data-Types* interfaces were simplified, so they no longer accept data size arguments. The *indivisible size* of a data-type may or may not be dependent on the respective data size, therefore, this value is still allowed in the data-types definition, but it can be overridden by the respective `Vector` object during the execution request. The interface of `Vector` objects has also been modified, in order to receive the size of each dimension of the respective data, so the `global_work_size` of each kernel can be inferred later, at runtime. Several of the configurations previously stipulated in the construction of the computational trees are now specified in a configuration file, examples are the number of GPUs to use and the number of overlapping partitions. Listing 4.1 shows the differences in the constructor of the `BufferData` datatype. Note that in this version, the `indivisibleSize` parameter is still acceptable, but

Listing 4.1: `BufferData` constructor alterations

```
1 //Multi-GPU BufferData constructor
2 BufferData(unsigned int numberOfElements, IWorkData::dataMode accessMode,
3   IWorkData::partitionMode partitioningMode, unsigned int indivisibleSize);
4
5 //New BufferData constructor
6 BufferData(IWorkData::dataMode accessMode,
7   IWorkData::partitionMode partitioningMode, unsigned int indivisibleSize);
```

Listing 4.2: `Vector` constructor alterations

```
1 //Multi-GPU Vector constructor
2 Vector(void* data, size_t elemSize, unsigned int nElements);
3
4 //New Vector constructor
5 Vector(unsigned int indivisibleSize, void* data, size_t elemSize,
6   unsigned int dim1Size, unsigned int dim2Size, unsigned int dim3Size);
```

as we can see in Listings 4.2, the `Vector` constructor can also receive this parameters, overriding the values defined at in the `BufferData` constructor (if they are specified). Note also, in the latter listing, that the new `Vector` constructor requires the individual size of each dimension instead of the total number of elements.

Other than the common modifications to all the skeletons, the `Loop` skeleton required some additional effort. Like explained in Section 3.1.3, the `Loop` may sometimes require a global synchronization step between iterations. With the dynamic behavior of the platform, some work-load distributions may create empty partitions (with a size of zero), due to the indivisible size restrictions, or to the fact that some applications may perform better that way. In these cases, the thread responsible for those partitions will not prompt an OpenCL execution. Therefore, during the synchronization step, the synchronizing thread must be able to determine which threads are actually executing and which are idle, to avoid deadlocks.

The skeleton's execution handler (common to all skeletons and implemented in the `Skeleton` abstract class, omitted in Figure 3.1 for simplicity) was modified to handle the execution requests depending on the current state of the platform: handling the request as a training execution if the training is active, or, prompting a device work-load rebalance before starting the execution, if the work-load was rebalanced in the previous execution (by the dynamic load balance). Also, upon receiving the execution request, this module will prompt the `Scheduler` to compute the work partitions of each device if the size of the input data differs from the previous request's input size. Due to the introduction of different execution platforms, all skeletons had to be adapted so they are aware of which execution platforms to prompt each partition, when an execution is requested.

Similarly to the skeletons interface, also the `KernelWrapper` interface had to suffer some changes due to the added dynamism. In the previous version, the OpenCL `global_work_size` array was defined as a constructor parameter of the

Listing 4.3: `KernelWrapper` constructor alterations

```
1  //Multi-GPU KernelWrapper constructor
2  KernelWrapper(const std::string kernelFile, const std::string kernelFunc,
3    const std::vector<std::shared_ptr<IWorkData>> &inputEntries,
4    const std::vector<std::shared_ptr<IWorkData>> &outputEntries,
5    const std::vector<unsigned int> &globalWorkSize),
6    const std::vector<unsigned int> &localWorkSize);
7
8  //New KernelWrapper constructor
9  KernelWrapper(const std::string kernelFile, const std::string kernelFunc,
10   const std::vector<std::shared_ptr<IWorkData>> &inputEntries,
11   const std::vector<std::shared_ptr<IWorkData>> &outputEntries,
12   const std::vector<unsigned int> &threadWorkSize,
13   const std::vector<unsigned int> &localWorkSize);
```

`KernelWrapper`. Now, given that the programmer no longer has to commit itself with data-sizes at the tree's construction time, this information is no longer required. However, in order to be later computed from the size of the vectors given as input to an execution request, we require a parameter called `threadWorkSize`, which is a vector that specifies the number of elements computed per dimension, by each thread. This parameter can be omitted if the kernel operates upon a single element and if no `localWorkSize` is specified. The information provided by the `threadWorkSize` vector is used to infer the `global_work_size` at a later time, based on the dimensions of the submitted `Vector` objects. The modifications to the `KernelWrapper` constructor are displayed in Listings 4.3

The added dynamism also demanded the `KernelWrapper` to be able to allocate and deallocate OpenCL devices memory in runtime, to cope with different work-load distributions. Also, different CPU/GPU configurations may require the `KernelWrapper` to reconfigure itself to address different platform configurations, like creating/freeing OpenCL resources to deal with the different number of partitions.

Moreover, like previously stated, some threads do not prompt OpenCL executions when they have no data to operate upon. The information of which threads are executing an OpenCL computation and which are idle is stored in the `KernelWrapper`. This is especially important for the aforementioned loop with global synchronization. Other than storing this information, the `KernelWrapper` also assists the `Loop` skeleton in the global synchronization process.

Finally, the `KernelWrapper` and its auxiliary module, the `KernelBuilder`, had to be adapted so they compile the OpenCL kernels for the CPU devices as well.

### 4.2.1 Programming example

In this section, we revisit the same programming example presented in Section 3.1.4 for the multi-GPU version of Marrow, highlighting the differences of the new interface. A programming example using the new interface is shown in Listing 4.4. The most notable

Listing 4.4: Image Pipeline implementation in Marrow

```
// Stage 1: Computation tree configuration

std::vector<std::shared_ptr<IWorkData>> inputData(2);
inputData[0] = std::shared_ptr<IWorkData> (
  new BufferData<cl_uchar4>(IWorkData::PARTITIONABLE));
inputData[1] = std::shared_ptr<IWorkData> (new FinalData<int>(factor));

std::vector<unsigned int> threadWorkSizes(2);
threadWorkSizes[0] = 2;
threadWorkSizes[1] = 1;

std::vector<std::shared_ptr<IWorkData>> outputDataInfo (1);
outDataInfo [0] = std::shared_ptr<IWorkData> (
  new BufferData<cl_uchar4>(IWorkData::PARTITIONABLE);

std::unique_ptr<IExecutable> gaussKernel (
  new KernelWrapper(gaussNoiseKernelFile, "gaussian_transform",
  inputData, outputData, threadWorkSizes));

inputData[1] = std::shared_ptr<IWorkData> (new FinalData<int>(threshold));
std::unique_ptr<IExecutable> solariseKernel (new KernelWrapper(
  solariseKernelFile, "solarise_transform", inputData, outputData,
  threadWorkSizes));

inputData.resize(1);
std::unique_ptr<IExecutable> mirrorKernel (new KernelWrapper(
  mirrorKernelFile, "mirror_transform", inputData, outputData,
  threadWorkSizes));

std::unique_ptr<IExecutable> p1 (new Pipeline(gaussKernel, solariseKernel));
std::unique_ptr<IExecutable> p2 (new Pipeline(p1, mirrorKernel));

// Stage 2: Execution request

std::vector<std::shared_ptr<Vector>> inputData(1);
std::vector<std::shared_ptr<Vector>> outputData(1);
unsigned int indivisibleSize = uiImageWidth;
inputData[0] = new Vector(
  indivisibleSize, input, sizeof(cl_uchar4), uiImageWidth, uiImageHeight);
outputData[0]  = new Vector(
  indivisibleSize, output, sizeof(cl_uchar4), uiImageWidth, uiImageHeight);

IFuture *future = p2->write(inputData, outputData);
future->wait();
```

difference is the absence of any reference to the size of the input data during the configuration of the computational tree (lines 3 to 31). In the definition of the `BufferData` objects (lines 4-5 and 13-14), only the partitioning option is defined, although, in this case, this option could have been omitted, since the data is considered partitionable by default. The indivisible size is also not specified here, since, for this application, it is dependent on the input size. The *globalWorkSize* is also undifined for now, instead, the `threadWorkSizes` array is configured (lines 8 to 10), defining the number of elements per dimension that each thread works upon. In this example case, each kernels operates upon two pixels of the same line (first dimension) for a single execution. Also, with the introduction of the configuration file, the platform configuration options, like the number of GPUs and the number of overlapping partitions is not defined in the skeleton constructor anymore (lines 17-18, 21-23 and 26-28).

In the execution request, the most visible difference lie in the `Vector` definitions (lines 38-41). For this application, the indivisible size is defined here, since it matches the width of the image. Also, unlike the multi-GPU version, the size of each dimension now has to be specified as different arguments rather than as the full size of the input buffer, so the *global_work_size* is properly calculated.

## 4.3 Runtime System

The Runtime layer was the layer that suffered the larger number of modifications, at an architectural level, with the introduction of new modules, but also at a behavioral level, with the modification of the behavior of some of the already existing modules. Figure 4.1 shows the new architecture of this layer. Two new modules have been introduced, the `Configuration` and the `WorkDistributionBase`. The `ExecutionPlatform` is now an abstract class (omitted in the figure for simplicity), with the common behavior of all the execution platforms. For this work, we introduced two execution platform specifications: the `GPUExecutionPlatform` and the `CPUExecutionPlatform`, to address GPU and CPU devices, respectively.

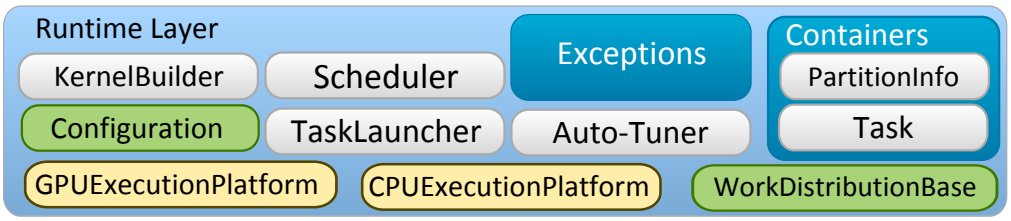


Figure 4.1: Marrow's new Runtime Layer

The `Configuration` is a new module introduced due to the increased number of configurable arguments in this version: like whether or not to execute the training session, the number of training iterations, among others. This new module is responsible for reading all this configurations from a file, configured by the user (an example of a

Listing 4.5: Configuration File example

```
# number of GPUs to use
# (0 to use all)
num_of_gpus = 1

# Use CPU
# (1 = true, 0 = false)
use_cpu = 1

# Number of training executions to perform
training_iterations = 10

# Number of executions performed by each device
# to compute average execution time
training_reiterations = 5

# Activate dynamic load balance
# (1 = true, 0 = false)
dynamic_balance = 1

# Performance ratio to activate rebalance
# (must be between 0 and 1)
balancing_ratio = 0.6

# Select the training mode to execute:
# 0 = default
# 1 = 50/50 split
# 2 = GPU steal
training_mode = 1

# Activate the training for the first execution
# (1 = true, 0 = false)
training_first = 1
```

configuration file is shown in Listing 4.5), and provide the remaining modules with this information, when requested. Also, parameters that were previously passed as skeleton arguments, like the number of GPUs to use and the number of overlapping partitions, are now configured in this file, allowing for a more configuration independent application programming. Furthermore, since the remaining modules are now subject to reconfiguration (unlike the previous version where some modules were static), this new module is now the only static part of the framework and it is responsible for keeping the current state of the framework between platform reconfigurations.

The `CPUExecutionPlatform` and the `GPUExecutionPlatform` are two implementations of the `ExecutionPlatform`, which has been refactored into an abstract module that factorizes the behaviors common to all execution platforms. Previous versions of Marrow were only focused on a single device type, the GPU, which were all managed in the same way. The introduction of the CPU as a new device, required a distinct treatment from the GPU's, so they can each be managed in a more device specific, and therefore, efficient way. The main differences between these modules reside in the

fact that in the CPU environment, we will not take advantage of the overlapping functionality since we are working with local memory. On the other hand, in OpenCL, a CPU device can be divided into distinct sub-devices, a functionality called *Device Fission* [43], giving the framework more control over the work distribution among a device, as each sub-device can be addressed independently. This functionality allows a CPU device to be partitioned in different ways:

- **Equally** Given the number of computational units per sub-device, the device is partitioned into as many sub-devices as possible;
- **By Counts** Allows for the specification of the number of computational units per sub-device to be created, resulting in a heterogeneous set of sub-devices;
- **By Affinity Domain** Creates the sub-devices based on memory affinity. OpenCL supports affinity fission by NUMA node and L1 to L4 caches, although, some devices only provide support for a sub-set of this set.

In this work, we only considered fission by affinity domain, since we are not interested in creating heterogeneous sets of sub-devices and dividing the device equally can yield a large number of combinations, depending on the total amount of computational units of the device. However, since only the `ExecutionPlatform` is aware of the fission level, the fission method can be easily replaced.

The execution platform modules are also responsible for creating the OpenCL contexts for each device. To fully exploit asynchronous OpenCL submissions, one different context is created for each GPU overlap level [42] and for each CPU sub-device resultant of the device fission. Finally, to maintain future extensibility and adaptability, it is the responsibility of the each execution platform to offer an iterator over its possible configurations.

The `Scheduler` module was restructured to coordinate the reinitialization of the `Auto-tuner`, `TaskLauncher` and of all the `KernelWrapper`'s when a platform reconfiguration is needed. This module is also responsible for the management of the task queue of each device, one for each level of overlap (as in the previous version) and one for each CPU device (recall that each sub-device resultant of the OpenCL fission is treated as a normal CPU device).

The `TaskLauncher` is responsible for launching the OpenCL executions and data communications between the host and the devices. In Marrow, the OpenCL execution of each partition is controlled by dedicated threads, the task launching threads. The modifications to this module include the initialization of the CPU devices' task launching threads (one for each device), along with the GPU devices' threads (one for each level of overlap). Also, the existence of different execution platforms require each thread to be aware upon which platform to prompt its executions. This is achieved by passing that information to the executing threads. Given that the new dynamic behavior of the framework requires a later allocation of the device's memory, it is now the responsibility

of each thread to, when necessary, prompt the allocation on the associated device, of the memory required to store the assigned the data partition.

The task launching threads are also responsible for measuring the duration of each OpenCL execution and hand those values to the `Auto-tuner` (through the `Scheduler`), to be globally compared. To minimize the effect of possible performance discrepancies, during a training iteration, each thread submits the same execution multiple times (the amount of submissions is specified in the configuration file) and the submitted duration value is the average of the duration of all those executions.

The `Auto-tuner` is in charge of computing the partitions of each device. Given that, in this version of Marrow, the partitioning may change between iterations, this module received the extra responsibility of calculating those partitions, based on the execution time measurements received from each task launching thread during the training process. Moreover, during *normal* executions, this module is also responsible for comparing the execution durations of each partial execution, and determine, whether or not to rebalance the partitions or even when to reverts to a partial execution of the training process.

Finally, the `WorkDistributionBase` is a new module, introduced to deal with different work-distributions for different work sizes, in a modular way. This module is responsible for storing the partitioning information of the training executions, making such information available for every upcoming execution request upon data-sets of the same size. Also, when there is no information regarding a specific inputed data size, the `WorkDistributionBase` derives the partitioning information from the work distribution information of previous executions. This process is further detailed in Section 4.5.2.

## 4.4 Execution Model

The execution model of the Marrow framework was subject to a considerable amount of modifications in this new version, mainly due to the necessity of adding dynamism to some of its modules (that previously implemented static behaviors). This added dynamism allows these modules to reconfigure themselves at runtime, based on the current state of the system, and the computation's input data size. As previously stated in Section 4.1, the work partitioning process is no longer performed at skeleton initialization time, but rather when the execution request is submitted. Accordingly, the framework's new execution model can be regarded as a three stage process: *Skeleton Creation*, *Skeleton Work Partitioning* and *Skeleton Request Execution*. The execution model is slightly conditioned by the training process. We will detail this impact in Section 4.5.1.

**ion** stage, is the stage responsible for the creation of the `KernelWrapper` and `Skeleton` objects. From the application's point-of-view, this stage replaces the old *Skeleton Initialization* stage, however, in this new version, the `Skeleton` initialization is postponed until the tree's first execution. Figure 4.2 shows the steps taken inside this

stage: First, the application creates the `KernelWrapper` objects (step 1) and compiles the OpenCL kernel for all the present devices through the associated execution platform (step 2). Then, with the `KernelWrapper` objects initialized, the application creates the `Skeleton` objects that stay uninitialized until they receive the first execution request.

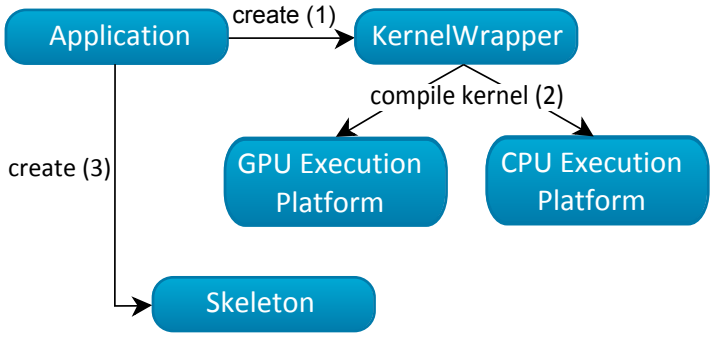


Figure 4.2: Skeleton Creation stage

**Skeleton Work Partitioning** stage is responsible for the work partitioning before the actual execution requests can be performed. This stage is only executed before the first execution request and whenever there is a modification in the dimensions of input (and output) arguments, as the dimension of each partition has to be readjusted. Figure 4.3 illustrates the steps taken during this stage, starting with the application execution request over the `Skeleton` (step1). Then, the `Skeleton` will initialize (or reinitialize) itself with the argument's data dimensions (step 2), associating them with the *Kernel Data-Types* specified in the previous stage (passed as arguments to the `KernelWrapper` constructor) and setting this dimension information in every kernel nested in the computational tree (step 3). Afterwards, the `Skeleton` requests the `Scheduler` to create the partition sizes over the new data-set dimensions (step 5). The `Scheduler`, in turn, will resort to the `Auto-Tuner` module to compute those partitions, based on the performance information of each present GPU (step 6) and of the results of the training session (step 7), kept by the `WorkDistributionBase` module. Lastly, the `Scheduler` stores the calculated partition sizes to the respective kernels (step 8), finalizing this stage, as the platform is now able to perform the requested execution.

**Skeleton Request Execution** stage, depicted in Figure 4.4, follows from the *Skeleton Work Partitioning* stage, or directly from the execution request, when the former stage is not required (step 1). As in the previous version of Marrow, the `Skeleton` creates a `Future` object (step 2) and return its reference to the application (step 3). Then, it submits the task to the `Scheduler` (step 4) that uses the partition information, previously stored in each `KernelWrapper`, to split the work-data among the GPU and CPU devices' queues. Concurrently, the `TaskLancher` consumes from each of these queues (step 6) and prompts the execution of the computational tree, over the correspondent

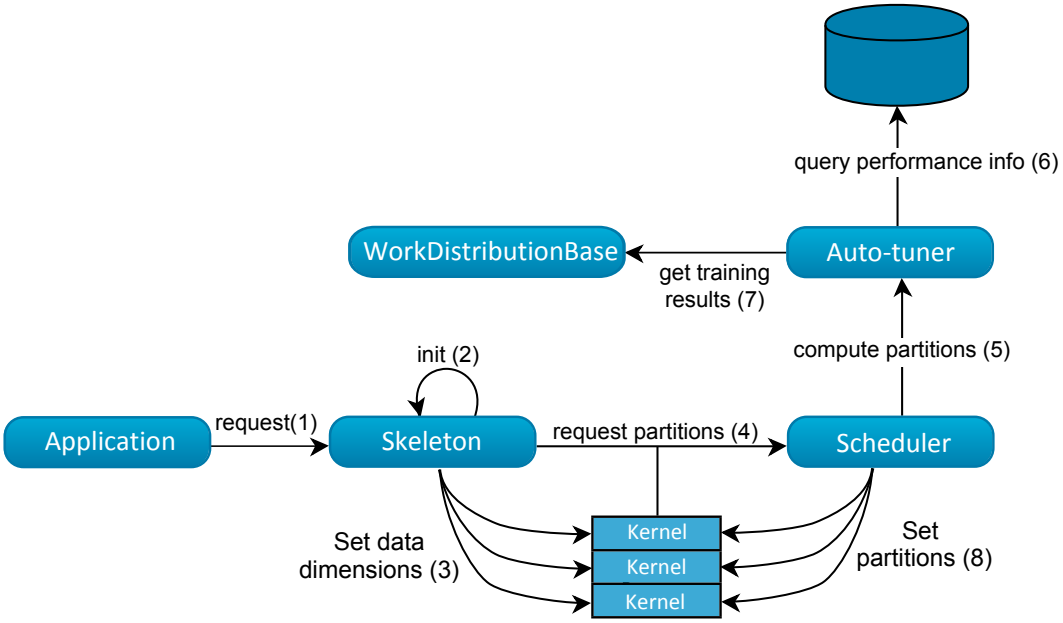


Figure 4.3: Skeleton Work Partitionging stage

data partition (step 7), by directing the execution of each device to the respective execution platform (step 8). When all the partitioned tasks have finished, the `TaskLauncher` notifies the `Future` object (step 9) which in its turn, notifies the requesting application (step 10).

## 4.5 Work-load Distribution

The main focus of this work is to distribute the work-load of an application among the CPU and GPU devices present in a system. As previously stated, finding an efficient distribution among heterogeneous devices is not a trivial task, and the difficulty increases when architectural differences between processors are more perceivable, like with CPUs and GPUs. This version of Marrow is specially focused on applications with a recurrent submission of different data-sets (with possibly different data sizes) over the same computational tree. Our solution is based on the offline training of an application, with one or more different data sizes, and an online partitioning derivation for data sizes not trained offline. All the trained and derived work distribution information is store in a *Knowledge Base* and updated every time a distribution is rebalanced.

Figure 4.5 illustrates the decision process taken by Marrow upon receiving an execution request. In the first execution, if the Knowledge Base is empty, a training is executed for that given input size, regardless of the training flag status. The resultant partitioning is stored in the Knowledge Base. Subsequent iterations are only trained if the training flag is active (specified in the configuration file). In the following execution requests, if the data size remains the same, the request is immediately prompt for execution, unless a work distribution is needed. In this case, a quick partition recalculation is performed

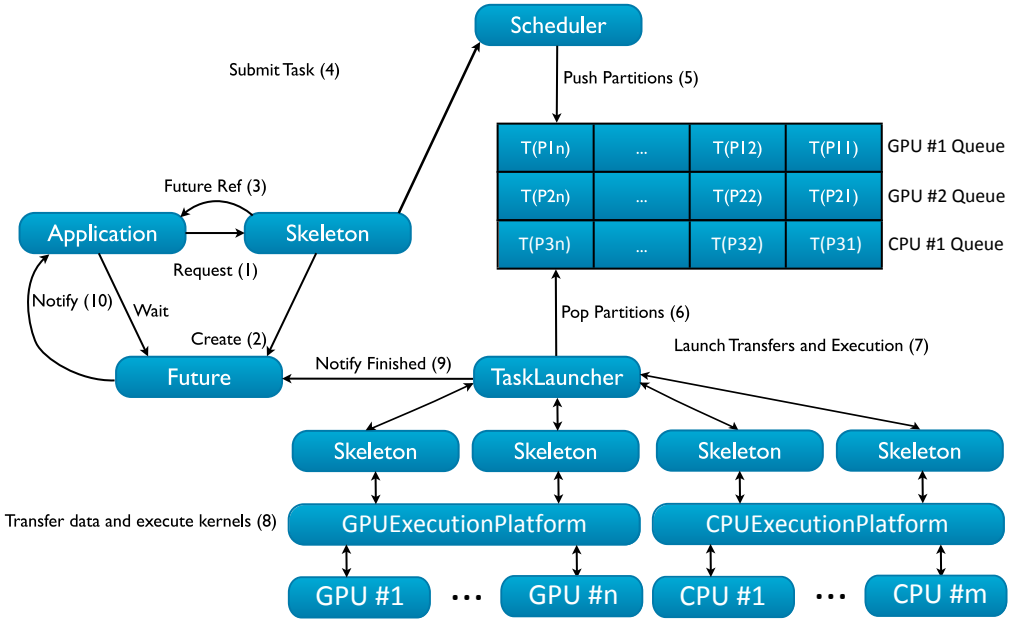


Figure 4.4: Skeleton Request Execution stage

before prompting the execution.

When the submitted work data has a size different than the size of the previous iteration's data, Marrow checks the Knowledge Base to see if it stores the partitioning information for the submitted input size. If it does, the partitioning is loaded and the execution is prompt. Otherwise, Marrow checks the stats of the training flag. If the flag is active, a full training is executed and the resultant partitioning is added to the Knowledge Base. Else, the work partitioning is derived from the Knowledge Base. In this scenario, there is a chance that the derived work distribution is not the ideal for this data size, but in this case, after some unbalanced iterations, the rebalancing will be triggered, the balance will be found and the Knowledge Base updated.

### 4.5.1 Training

When we introduced our goal of incorporating CPU OpenCL computations in a multi-GPU framework, we identified the architectural and execution model differences between CPU and GPU devices, as the main obstacle to a static balanced work partitioning. This drove us to look for a more empirical method to reach a balanced work distribution among those devices. Our solution is based on performing a set of offline training executions, during the application's first execution, where different work distributions for a given data-set are tested and the one that presents a best overall performance is selected for all the subsequent executions. Since it is unbearable to exhaustively train every single possible work distribution combination, we designed two distinct training modes that builds on the performance results of the previous iterations to pick the next distribution to test.

In generic lines, both training modes group the devices into two groups: the GPU

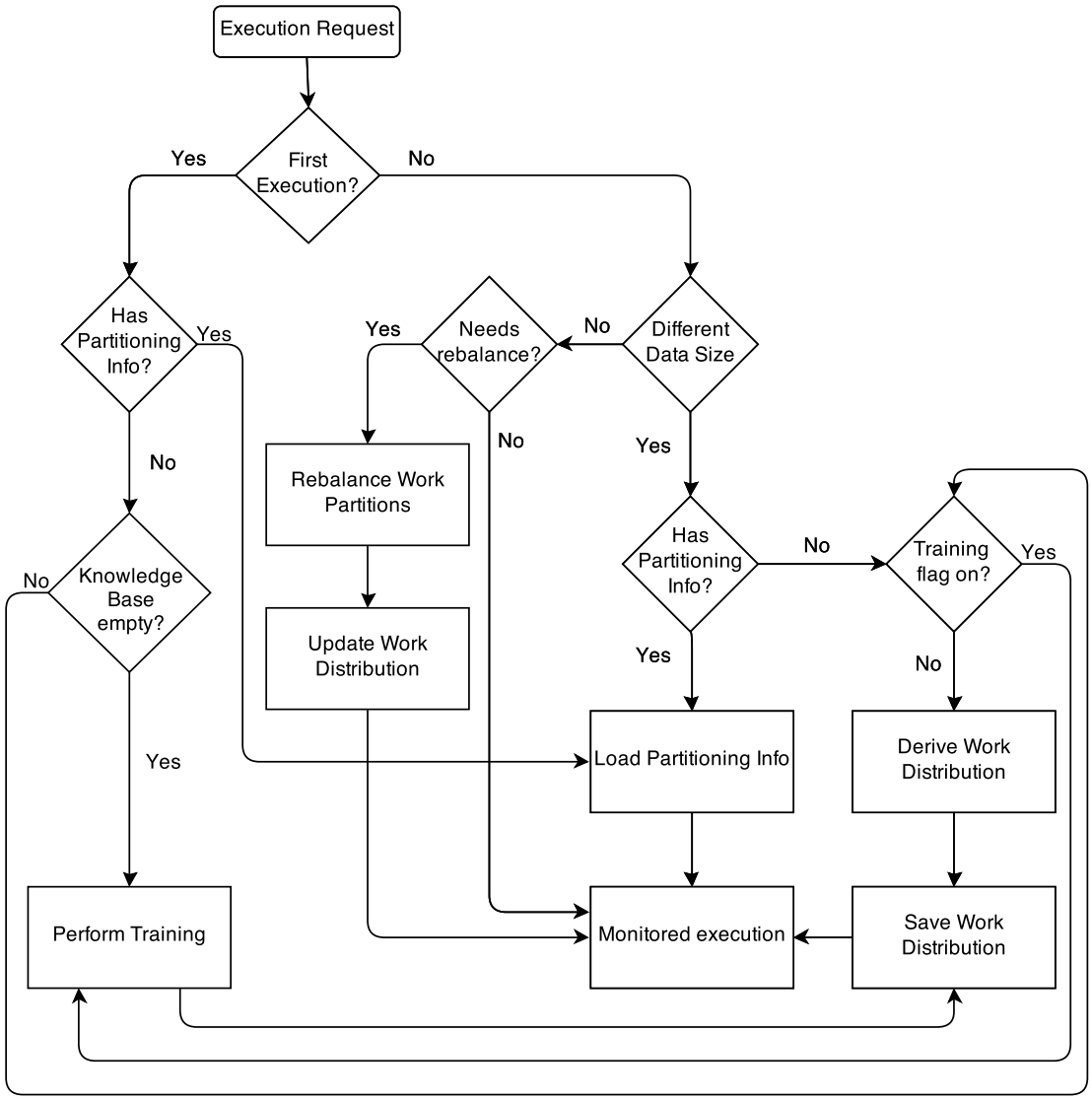


Figure 4.5: Work distribution decision process

devices and the CPU devices. During the training, Marrow tries to find the best work balance between these groups by testing out different work distributions across them. Inside each group, the partitioning between devices is done in a static way: each GPU's partition size is calculated based on the performance values acquired prior to the execution (identical to the multi-GPU version), and the CPU's group partition is divided equally among all the CPU cores, as all cores are homogeneous among themselves. The next sections detail the training process mechanism and present the algorithms for the two implemented training modes.

#### 4.5.1.1 Training Execution Model

Section 4.4 describes the execution model of a normal execution request on Marrow. To prepare the framework for a specific application to run at its best performance, under a specific system, the application needs to perform a one-time-only heavy execution with a slightly different execution model. In this training execution, one of the implemented training modes is performed under different combinations of configurations of both the execution platforms. Algorithm 1 explains how the process is accomplished.

The process starts when the computational tree receives an execution request and the *isTraining* flag is set to *true*. Then, each of the execution platforms provide an iterator over all its possible configurations (lines 3 and 5). One of the training algorithms is then executed for every configuration combination, in this case, fission/overlap combinations (lines 9 to 14). The training algorithm is defined by the user via the configuration file. For each configuration, the execution platforms needs to reconfigure itself (lines 4 and 6) and the training is reset so the partitioning ratios are set to their start values (line 7). These starting values depend on the training algorithm used. After this process, Marrow is ready for a new training cycle. Afterwards, for a given number of iterations (also specified in the configuration file), a new task is created and submitted to the `Scheduler` (lines 11 and 12). Each submission will perform one iteration of the selected training algorithm. Internally, given that the *isTraining* flag is active, the platform executes the submitted task in training mode. When in training mode, each executing thread prompts the same execution multiple times (the amount of executions is specified in the configuration file) to the associated device. The duration of each execution is measured and the average duration is used by the `Auto-tuner` to compare all the threads durations and compute the partitions for the next training iteration. Performing the same execution multiple times allows for a more reliable evaluation, attenuating possible discrepancies. Since the device groups' ratios are modified at each training iteration, the partitioning needs to be recomputed before submitting another task to the `Scheduler` (line 10). After all this training iterations, the current work distribution is expected to be the best performing one. However, to make sure that the best tested distribution is chosen, the platform internally saves the ratios that showed the best global performance and loads them at the end of the training cycle (line 15).

To make sure that we compare the different configurations' performances in an environment closer to the expected in a normal execution, after the training cycle, the platform temporarily deactivates the training and submits a few normal executions (this number can also be configured by the user) with the training mode disabled, while measuring the time they take to execute (lines 20 to 27). The average duration of those executions will be used later to compare the performance of the different configuration combinations (line 35).

After all the combinations of fission and overlap configurations have been tested, the one that performed better is picked (line 35), the platforms reconfigure themselves for

**Algorithm 1** Training execution

**Require:** $isTraining = \textbf{true}$
1: $devicesRatios \leftarrow \emptyset$
2: $performances \leftarrow \emptyset$
3: **for all** $fission \in$ CPUExecutionPlatform.configurations() **do**
4: CPUExecutionPlatform.reconfigure($fission$)
5: **for all** $overlap \in$ GPUExecutionPlatform.configurations() **do**
6: GPUExecutionPlatform.reconfigure($overlap$)
7: scheduler.resetTraining()
8: {Compute best CPU/GPU ratio for the current fission/overlap configuration}
9: **for** $numberTrainingIterations$ **do**
10: scheduler.computePartitions()
11: $task \leftarrow$ new Task
12: scheduler.submit($task$)
13: wait for Task to finish;
14: **end for**
15: scheduler.setBestTrainingRatios()
16: scheduler.computePartitions()
17: $isTraining \leftarrow \textbf{false}$
18: $totalTime \leftarrow 0$
19: {Compute the execution time for the best CPU/GPU ratio for the current fission/overlap configuration}
20: **for** $numberPerformanceIterations$ **do**
21: $start \leftarrow$ currentTime
22: $task \leftarrow$ new Task
23: scheduler.submit($task$)
24: wait for Task to finish;
25: $end \leftarrow$ currentTime
26: $totalTime \leftarrow totalTime + (end - start)$
27: **end for**
28: $performances[overlap][fission] \leftarrow totalTime/numberPerformanceIterations$
29: $devicesRatios[overlap][fission] \leftarrow$ scheduler.getCurrentRatios()
30: $isTraining \leftarrow \textbf{true}$
31: **end for**
32: **end for**
33: $isTraining \leftarrow \textbf{false}$
34: {Reconfigure the framework according to the best overall fission/overlap configuration}
35: $(bestOverlap, bestFission) \leftarrow$ pickBest($performances$)
36: CPUExecutionPlatform.reconfigure($bestFission$)
37: GPUExecutionPlatform.reconfigure($bestOverlap$)
38: scheduler.setDevicesRatios($devicesRatios[bestOverlap][bestFission]$)
39: scheduler.computePartitions()

that configuration (lines 36 and 37), the associated partitioning ratios are restored (line 38), and the work distribution is recalculated (line 39). From this instant, Marrow is ready to execute the upcoming requests with the best configuration and the best work distribution found for the current application.

#### 4.5.1.2 Training Modes

In this version of Marrow, we introduced two distinct training modes:

- The *50/50 split* mode, that is based on direct comparison between the different device's execution performance;
- The *CPU assisted GPU execution* mode, based on incremental transfers of work-load from the GPU to the CPU and global performance comparison.

In this section we detail each of these modes.

**50/50 split mode** The *50/50 split* training mode is built under the assumption that, if a device group's performance is lower than the performance of its counterpart, the work-load should be readjusted by moving some of the work from the worst performing group to the best performing one. With this premise, in any given training iteration, the framework evaluates the performance of both device groups individually. In the subsequent iteration, the group that performed better (had a lower execution time) will receive a portion of work from the group that performed worse, and both performances are reevaluated. It this sense, both performances are expected to come closer to each other after each training iteration, reaching a balanced work partitioning after a few iterations.

To perform this training, the framework will keep track of the amount of work that can be transferred between the groups (the *transferable partition*) and the amount that is bound to each group (the minimum amount of work that the training has identified that will be performed by that group). Before the first training iteration, all the work is considered to be transferable and no work size is bound to any group. At each iteration, the transferable partition is divided evenly between the two device groups. To determine to which group it must be bound, a training round is executed and the the haft of the transferable partition sent to the device group that performed better becomes bound to that group. The remaining transferable partition becomes the transferable partition for the next training iteration. In this sense, the transferable partition can be regarded as the portion of the work data that is still under training. Because it is divided by 2 at each iteration, the size of the transferable partition after n iterations follows the function:

$$transferableSize = \frac{globalSize}{2^n} \tag{4.1}$$

and since,

$$\lim_{x \to \infty} transferableSize = 0 \tag{4.2}$$

the impact of each training iteration in the resultant work distribution is likely to decrease with each iteration.

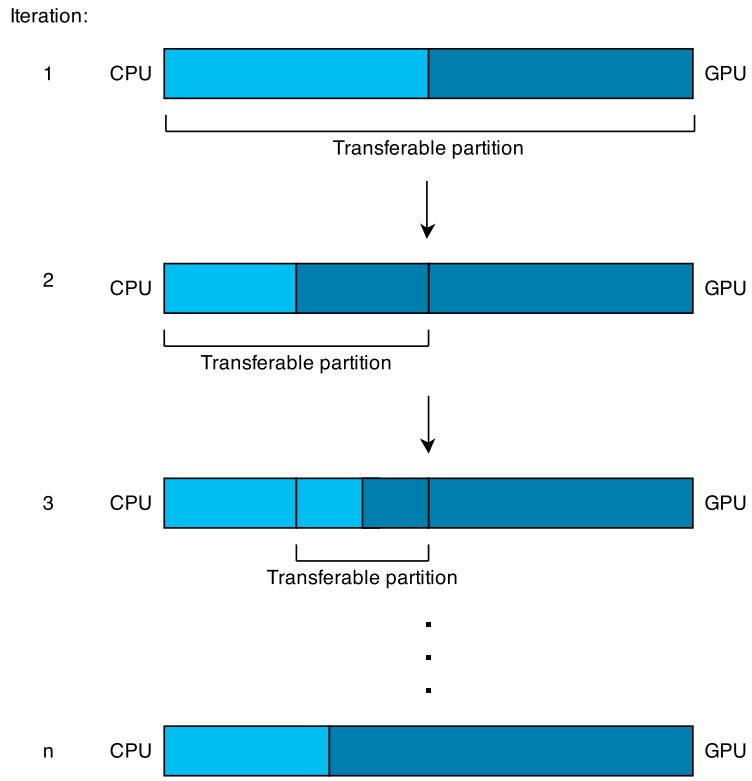


Figure 4.6: 50/50 split training example

Figure 4.6 exemplifies how this method works. Initially, since the transferable partition encompasses all the working data, that data will be divided evenly between the two device groups. After the first iteration, since the GPU group outperformed the CPU's, the GPU's partition is bound to that group as the remaining partition will be used as the transferable partition for the next execution. In the second iteration, the same process is executed: the transferable partition is divided equally between the two groups, resulting in a work distribution with $1/4$ of the work data to the CPU group and the remaining $3/4$ to the GPU group. The same process is repeated for the number of iterations specified by the user in the configuration file, as the transferable partition size will tend to zero.

During the evaluation process, we noted that in some particular cases, some devices did not perform better when the size of their partitions was reduced. This can happen if the programmer does not specify a *localWorkSize* for a given kernel, which will make the OpenCL implementation pick one in runtime. Given that the localWorkSize values must be a divisor of the *globalWorkSize*, if the Auto-tuner does not have a localWorkSize restriction, the biggest divisor of the size of a created partition can be a very low value, and it may have a negative impact on the device's performance. As an extra measure to prevent this training mode to output a low performing work distribution, Marrow keeps track of the best performing training iteration so far, and after the last training

iteration finishes executing, if its performance is worse than the best performance found, the cooresponding work distribution is pick instead.

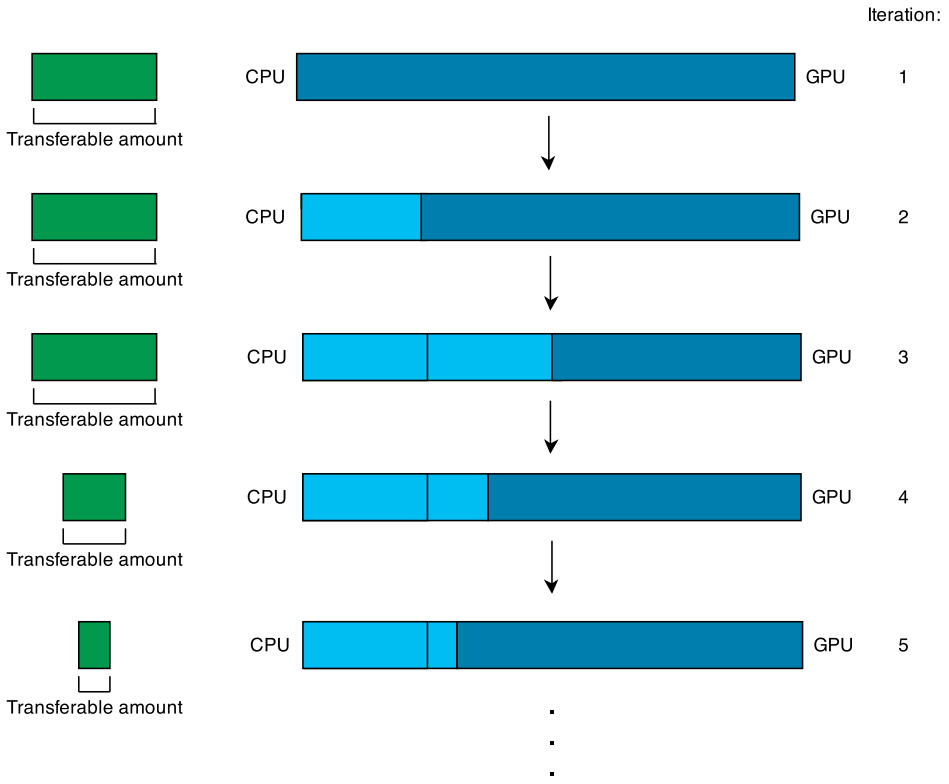


Figure 4.7: CPU assisted GPU execution training example

**CPU assisted GPU execution mode** The *50/50 split* training mode was designed with the assumption that by approximating the execution times of both device groups, the best performance possible (theoretically) would be attained. Even though we got the expected results in most of our benchmarks (evaluation in Chapter 5), this may not always be the case. Also, there are some circumstances under which the execution time of each device group is not measured correctly. One of this situations is when the computational tree has a `Loop` skeleton with a globally synchronized step computation (introduced in Section 3.1.3). In this particular case, all the task launching threads that control the OpenCL executions need to wait for each other to finish, so that the `step` condition can be computed. Therefore, and since the performance of each device is calculated based on the time it takes to execute the full computational tree, the faster devices' performances will be highly adulterated by the time they are waiting for the remaining devices.

To work around these issues, we implemented a training mode that instead of comparing the performances of the two device groups, it compares the global performance of one iteration with the global performances yield by the previous iterations. In this training mode, exemplified in Figure 4.7, the work-load transfers form one device group to the other is also guided by a different method than in the 50/50 split mode. In the CPU

assisted GPU execution mode, the initial execution is performed only by the GPUs, to obtain a first performance value. From the second iteration onward, a percentage value (configured by the user) called the *transferable amount*, is taken from the GPU group and added to the CPU group partition. The process repeats itself until the result yield by an iteration is worse than the iteration before. From then on, the transferable amount will be consecutively divided by 2 after each iteration. To determine the next work distribution to test, the algorithm will selects the distribution with the best performance so far and increase the CPU devices partition by the transferable amount. Similarly to the 50/50 split training, the transferable amount will tend to decrease over iterations and its impact on the partitions will also decrease.

#### 4.5.1.3 Dynamic load balancing

When the main differences between CPU and GPU devices were identified, it was pointed out that the performance of a CPU is hard to predict due to their multi-threading operating model, as it is affected by the current work-load of the system. When a training session is executed, it will be affected by the work-load of the CPU at that time, and the resulting best configuration and work distribution will reflect that. However, the global work-load of a system can change during the application's execution time, thus, the current work distribution may no longer be the optimal one.

To try to compensate the possible work-load fluctuations during execution time without introducing a noticeable overload, Marrow enables the dynamic balance of the work distribution. For that purpose, it comprises a lightweight online monitoring process, which detects continuous performance descrepacies between the two device groups and performs a quick partitioning rebalance, transfering a fixed small percentage of the work-load of the worst performing device group to the best performing one. After a couple of rebalancing attempts, if the balance is still not satisfactory, Marrow performs a lighter version of a training execution, where only the current configuration is trained.

To monitor the runtime execution of a submitted task, each task launching thread measures the duration of its partial execution over the computational tree. Upon completion, each thread sends their execution duration time to the `Auto-tuner` module, so that the latter may evaluate the need for a rebalance. To that end, the Auto-tuner, when in the possession of all the measured execution times, picks the durations with the longest running execution time of each device group and it calculates the ratio between the shortest and the longest running durations through the following function:

$$performanceRatio = shortestDuration/longestDuration \tag{4.3}$$

This value is then compared against the *balancingRatio* parameter, a ratio (between 0 and 1) defined by the user in the configuration file. If the *performanceRatio* is lower than the *balancingRatio*, this execution is considered to be unbalanced. However, this discrepancy can be an isolated case, so, instead of triggering the partition rebalance after one or a

| t | *lbt*$(t-1)$ | isUnbalanced($dev$) | *lbt*$(t)$ | Balance load? |
|---|---|---|---|---|
| 0 | - | - | 0.00 | |
| 1 | 0.00 | 1 | 0.66 | |
| 2 | 0.66 | 1 | 0.88 | |
| 3 | 0.88 | 0 | 0.30 | |
| 4 | 0.30 | 1 | 0.76 | |
| 5 | 0.76 | 1 | 0.92 | |
| 6 | 0.92 | 1 | 0.97 | |
| 7 | 0.97 | 1 | 0.99 | Yes |
| 8 | 0.99 | 0 | 0.34 | |
| 9 | 0.34 | 0 | 0.11 | |
| 10 | 0.11 | 0 | 0.04 | |
| 11 | 0.04 | 1 | 0.67 | |
| 12 | 0.67 | 1 | 0.89 | |
| 13 | 0.89 | 1 | 0.96 | |
| 14 | 0.96 | 1 | 0.99 | Yes |

Table 4.1: Example of the evolution of $lbl$ for $weight = 2/3$

fixed number of unbalanced executions, the decision of whether or not to perform a rebalance is delegated upon a function that we called the *Load Balancing Threshold function* (*lbt function*). This function's value is influenced by the historical information of previous executions, as its value increases with each unbalanced execution and decreases when the execution is balanced. When increasing, the *lbt* value will get closer to the value 1, therefore, we set the rebalance to trigger when the *lbt* value reaches 0.99. This allows for a better judgment on the decision of whether or not to rebalance the partitions, since the function's value only increases with a predominance of unbalanced executions over the latest executions. The *lbt* function is defined in the following way:

$$lbt(t) = isUnbalanced \times weight + lbt(t-1) \times (1 - weight) \tag{4.4}$$

The *weight* parameter is defined in the configuration file and represents the weight with which an unbalanced execution affects the *lbt* function. The bigger the weight value is, the sooner the rebalance is triggered. The *isUnbalanced* value can be defined as:

$$isUnbalanced = \begin{cases} 0 & \text{if } performanceRatio \leq balancingRatio \\ 1 & \text{otherwise} \end{cases}$$

Everytime the rebalance is triggered, the *lbt* value is slightly decreased (the equivalent of a balanced execution). By not reseting its value, if the work distribution remains unbalanced, the rebalance will be triggered sooner, expectedly in every execution until the balance is reestablished. Table 4.1 exemplifies the evolution of the *lbt* function along a few iterations with both balanced and unbalanced executions.

Also, everytime the rebalance is triggered, a **rebalance counter** is incremented. After a number of rebalancing failures (the number is defined in the configuration file), if the

work distribution remains unbalanced, the platform reverts to a partial execution of the training process that only considers the current fission/overlap configuration. This training mode, is similar to the offline training process described above. However, since it will be executed during the normal operation of the application, there is a need to minimize the overhead introduced by this process. Thus, when this process is executed during a normal execution, only the current configuration is going to be trained. Given that the unbalance was most likely introduced by changes in the system's overall load or an input with a size that differs from one of the already present in the Knowledge Base, there may be a different configuration that suits better this new conditions. However, the possible performance gains from choosing a better configuration are not significant enough (according to our evaluation) to justify the added overhead of a full training execution.

### 4.5.2 Partitioning Derivation

Some applications have an undefined number of possible input sizes. Even when this number is limited and known during the application's install time, training every distinct input data size may be unbearably heavy, even considering it is a onetime only, offline execution. To that extent, Marrow keeps a *Knowledge Base* of every trained and derived work-load distribution configurations. When a skeleton receives a request with an input data size different from the sizes of the executions previously executed (meaning that the partitioning information is not present in the Knowledge Base), the work-load distribution is derived from the partitioning information of the previously executed data sizes.

In this version of Marrow, the derivation consists in an interpolation between the new data size dimensions and the stored data sizes. The interpolation is achieved using the Euclidean distance, therefore, depending on the number of dimensions of the data, the distance between the inputed data $i$ and the stored data info $s$ is calculated through the function:

$$distance(i, s) = \sqrt{(s_{dim1} - i_{dim1})^2 + (s_{dim2} - i_{dim2})^2 + ... + (s_{dimn} - i_{dimn})^2} \quad (4.5)$$

This derivation process is built under the assumption that the closer the data sizes are, the more similar the ideal work-distribution configuration is. In this sense, a derived work-distribution is expected to have an acceptable performance at its first execution. If the work distribution is still considered unbalanced by the online system monitoring, the work distribution is rebalanced over the next couple of iterations without a noticeable overhead. In cases that the ideal work distribution (for the current system configuration) is too far from the derived distribution, a light training is executed. The resultant partitioning of the rebalance process updates the partitioning information on the Knowledge Base for the current data-size, allowing the future requests with this data size to be executed in a balanced distribution.

## 4.6 Summary

In this chapter, we presented the modifications to the Marrow framework performed during this work. We distinguished different execution platforms and introduced a work distribution mechanism to balance the load between GPUs and CPUs. We also explored the OpenCL fission functionality to take advantage of data locality in the CPUs.

Furthermore, we simplified the programming model of the framework by eliminating the possibility for inconsistencies between the configured and the submitted data dimensions, as we also lifted the restrictions over the submittable data sizes, previously stipulated upon the creation of the computational tree.

Our work-load distribution approach is based on an offline training, where the ideal number of overlapping partitions, as well as the ideal fission level is selected, and at the same time, we reach a balanced work-load distribution among GPU and CPU devices, for the selected configuration and inputted data size. To achieve such balance, we implemented two different training modes with two different approaches.

To maintain the balance over continuous executions while addressing possible system work-load fluctuations, as well as different performances for different input sizes, our implementation also includes an online system monitoring that detects performance discrepancies and corrects the work-load balance in a lightweight fashion.

# 5

# Evaluation

In this chapter, we present the evaluation of our approach to multi-CPU/multi-GPU support in the Marrow framework, presented in the previous chapter. In this study, we analyze the performance of the framework when executed on a CPU-only environment (Section 5.4). We then compare GPU-only executions of our benchmarks with executions of the same benchmarks using both CPU and GPU devices (Section 5.5). We took advantage of the dynamic load balance mechanism to evaluate the precision of the *50/50 split* training method (Section 5.6) and finally, we analyze the behavior of the dynamic load balance (Section 5.7).

## 5.1 Methodology and Metrics

The devised evaluation process aims to answer the following questions:

1. Can we take advantage of the OpenCL fission feature to increase the performance of CPU OpenCL computations?
2. Can we take advantage of the CPU to increase the performance in a heterogeneous system composed by GPUs and CPUs?
3. How accurate is the our training method approach?
4. How does the framework respond to alterations to the input data sizes and the system's work-load?

To analyze the OpenCL fission we compare the execution times achievable with fission partitioning with the execution times without fission. To evaluate the impact of

the CPU in the system's performance, we analyze the speedups achievable for a set of benchmarks. To evaluate the accuracy of the training, we compare the measured execution times of each device group (CPU and GPU). Finally, we performed two test cases to evaluate the dynamic adaptation behavior of the framework: one that attests the efficiency of the configuration derivation process and another that analyzes the behavior of the work-load partitioning when the system's work-load is intentionally altered, with an application design for this effect.

The benchmark executions consist in submitting the same input over the computational tree for 500 runs and the duration of the computation is measured. From those 500 runs, the highest and the lowest thirds of the execution times are ignored and the average is computed over the inner third (167 runs are considered). The first of this executions is the training execution which duration will be ignored with the highest third. The overlap partitions configurations tested are from 1 to 4 and the OpenCL fission partitioning tested are from caches L1 to L3 and without any fissioning.

## 5.2 Case-Studies

To conduct our evaluation, we adapted five of the benchmarks already available in the Marrow benchmark suite, namely: Image Filter Pipeline, FFT transformation, N-body-simulation, Saxpy computation and Image Segmentation.

**Image Filter Pipeline** is a benchmark that consists in the sequential application of three image filters over an inputted image. Some of the filter kernels were adapted from the AMD's OpenCL Samples. The filters applied are the Gaussian Noise, Solarize and Mirror Image. The computational tree contains two Pipeline skeletons, one nested inside the other and the three kernels are the leaf nodes. In every one of these kernels, each thread computes over two non-contiguous pixels of the same line. Therefore, some restrictions have to be specified. Firstly, when initializing the `KernelWrapper` for each kernel, the `threadSize` of the first dimension needs to be 2. Secondly, when creating a `Vector` for submission, the indivisible size must be the width (size of a line) of the input image.

**FFT** benchmark was initially adapted from the Shoc Benchmark Suite [44]. It consists in a set of Fast-Fourier Transformations with 512kbs each, being the computational tree a two-stage pipeline that composes the FFT with its inverse. The *indivisibleSize* of the input arguments is expressed as $numFFTs \times numElementsPerFFT$ and the *threadSize* of each kernel is 512 (the size of a FFT).

**N-Body** simulation is a classical problem that simulates the position and the velocity of celestial particles based on the interactions among them. This benchmark was initially adapted from the AMD's OpenCL Samples and follows the direct-sum algorithm, of complexity $O(n^2)$. The computational tree is composed by one Loop skeleton and the n-body

kernel. Given that in this simulation, each particle (body) is affected by all the other particles in the set, a full synchronization between iterations is required between iterations so all the computing devices have a global vision of the previous iteration result.

**Saxpy** stands for "Single-Precision AX Plus Y" and is part of BLAS (Basic Linear Algebra Subroutines). This benchmark consists in the multiplication of a matrix with a scalar value and the result is then summed with another matrix. This computation ($z[i] = ax[i] + y[i]$) can be completely parallelizable since each thread only operates over a single element of each matrix. The computational tree consists in a single Map skeleton with the Saxpy computation kernel and no partition restrictions are specified.

**Segmentation** benchmark performs a transformation over a gray-scale three dimensional image, changing its value to either white, gray or black. The computational tree is expressed as a single Map skeleton with a nested kernel. Although there is no algorithmic dependencies between pixel elements, the indivisible size is set to the size of the first two dimensions so the partitioning is performed only over the last one.

## 5.3 Systems

To evaluate different characteristics of our prototype solution, we used the two different systems described in Table 5.1.

**System** $S_1$ does not possess any GPU device but it contains 4 Opteron 6272 CPU processors with 16 cores each, summing a total of 64 cores. Both L1 and L2 caches are shared between two cores[1] (32 caches total) and the L3 caches are shared between 8 cores (8 L3 caches total). This systems characteristics provide an interesting environment to evaluate the OpenCL fission feature, as well as the performance of the Marrow framework in a CPU-only environment.

**System** $S_2$ holds two identical AMD HD 7950 GPU devices connected to the motherboard with two dedicated PCIe x16 lanes (one per GPU), allowing the system to scale better when the two GPUs are used by reducing the communication overhead. This system is also equipped with a single hyper-threaded 6-core Intel i7-3930K totaling in 12 CPU threads. Each core as its own L1 and L2 caches (total of 6 caches) and a single L3 cache is shared among all the cores. This system is used to evaluate the CPU and GPU interoperability and work distribution of the Marrow framework.

[1] Actually each core has its own dedicated L1 data cache, only the L1 instruction cache is shared, but this is the L1 cache identified by OpenCL Fission.

| | CPU | RAM | #CPU threads | Drivers | GPUs |
|---|---|---|---|---|---|
| $S_1$ | 4 x AMD Opteron 6272 @ 2.20GHz | 64GB | 64 | 1214.3 | N/A |
| $S_2$ | Intel i7-3930K @ 3.20GHz | 64GB | 12 | 1113.2 | 2 x HD7950 |

Table 5.1: Systems characteristics

| Benchmark | Input type | Input argument | Fission | number of subdevices | Execution time | Execution time (no fission) |
|---|---|---|---|---|---|---|
| Filter pipeline | Image size (pixels) (pixels) | 1024x1024 | L3 | 8 | 8.5 | 9.8 |
| | | 2048x2048 | L2 | 32 | 22.0 | 34.8 |
| | | 4096x4096 | L2 | 32 | 65.1 | 120.3 |
| | | 8192x8192 | L2 | 32 | 222.8 | 377.1 |
| FFT | Size of data-set | 128MB | L2 | 32 | 34.7 | 103.7 |
| | | 256MB | L2 | 32 | 56.5 | 197.9 |
| | | 512MB | L2 | 32 | 106.4 | 423.8 |
| NBody | Number of bodies | 8192 | L2 | 32 | 35.8 | 138.4 |
| | | 16384 | L3 | 8 | 99.0 | 284.0 |
| | | 32768 | L2 | 32 | 383.4 | 1116.2 |
| | | 65536 | L2 | 32 | 1499.0 | 4433.6 |
| Saypy | Number of elements | $1\times10^6$ | L2 | 32 | 2.2 | 7.4 |
| | | $10\times10^6$ | L2 | 32 | 23.9 | 72.1 |
| | | $50\times10^6$ | L2 | 32 | 102.9 | 270.8 |
| Segmentation | Number of elements | 1MB | L3 | 8 | 1.1 | 2.2 |
| | | 8MB | L3 | 8 | 4.3 | 11.8 |
| | | 60MB | L2 | 32 | 31.0 | 61.5 |

Table 5.2: CPU only executions in system $S_1$

## 5.4 CPU-only Execution

In this work we introduced the CPU as an OpenCL device in the Marrow framework, previously only supporting GPU devices. In this section, we analyze how the framework performs in a CPU only environment. Also, since we did not find any studies regarding the performance of OpenCL device fission feature, we started our investigation by analyzing the possibility of increasing the performance of an application by explicitly partitioning the CPU device and independently controlling the resulting sub-devices.

Table 5.2 shows the obtained results of the execution of the benchmarks on system $S_1$, with the execution time after the offline training and the expected execution time if no fission partitioning were used. These no-fission execution times were retrieved from the training performance measurements. To be noted that these performance measurements, showed to be a pretty accurate estimation of the real execution times. For a better

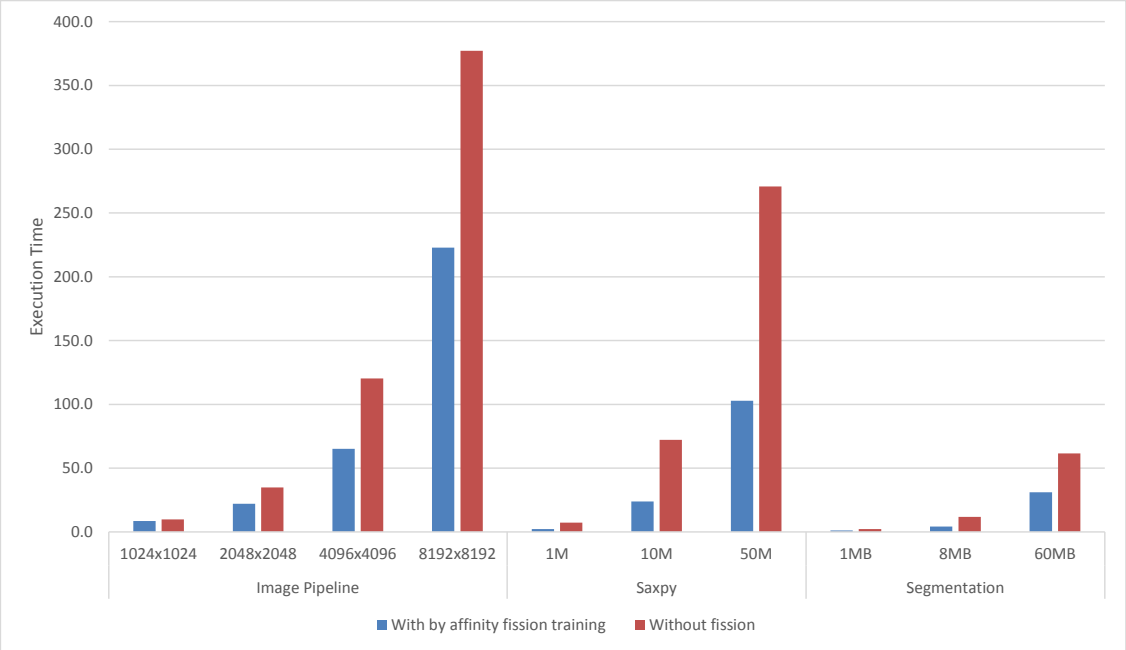


Figure 5.1: Comparison of the execution times with and without fission for Image Pipeline, Saxpy and Segmentation benchmarks on system $S_1$

comparison, the results are also graphically represented in Figures 5.1 and 5.2. The training execution consisted in testing the performance with fission over the caches L1, L2 and L3 and also without any fission. To be noted that OpenCL identifies the full set of CPU processors as a single device, meaning that if no fission is used, the 4 processors are presented to the application as a unified device.

By analyzing all the benchmark executions, it can be noted that the performance was highly improved with the use of the OpenCL fission functionality. Note that, however, the ideal fission level varies from benchmark to benchmark, and it is also affected by the size of the input data. Despite that, the biggest performance difference is between the no-fission configuration and the remaining. The reason for these numbers is predictably bounded to locality, as we are in presence of a NUMA architecture, where each CPU has its own memory slots. Therefore, the access times of each device to the same memory address is different for each CPU. As an example, Figure 5.3 shows the execution time for each configuration, as measured by the training process of the FFT with a 256 MB input size. The differences in the performance of the no-fission configuration and the remaining configurations is the most noticeable, but the performance differences among the latter cannot be overlooked.

## 5.5 Comparison against GPU-only executions

In this section we compare, from a performance perspective, Marrow executions that resort only to GPU executions, to the executions that also distribute the work-load among

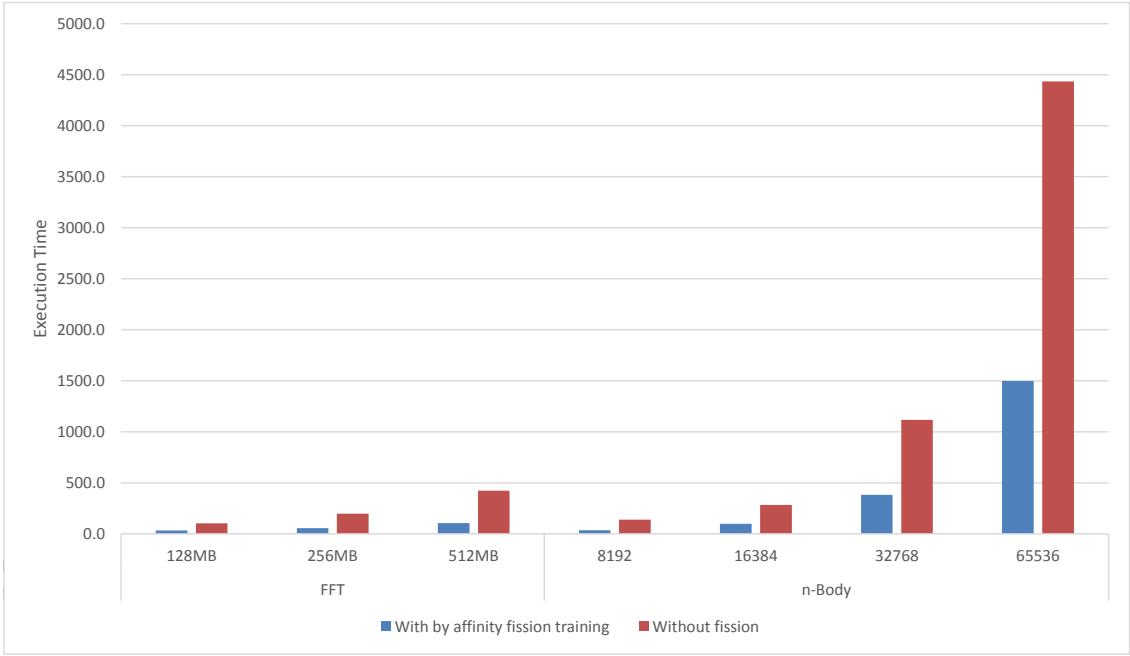


Figure 5.2: Comparison of the execution times with and without fission for FFT and N-Body benchmarks on system $S_1$

the available CPUs. We perform our analysis for 1 and 2 GPUs using system $S_2$, in order to access the gains obtained by the CPUs' assistance in either case.

For this evaluation, we executed of all the benchmarks using these different configurations:

- GPU-only execution, with a single and both GPUs. Internally, only the different configurations of overlapping partitions are trained;
- CPU plus GPU executions with the *50/50 split* training method (T1);
- CPU plus GPU executions with the *CPU assisted GPU execution* training method (T2).

In the two later configurations, where the CPU was used, the trained fission configurations were from caches L1 to L3 and with no fission. Tables 5.3 and 5.4 show the obtained results, for 1 and 2 GPUs, respectively. In the Figures 5.4 and 5.5, we display the speedup obtained with both training modes, when compared to GPU-only executions.

**Image Filter Pipeline** This is a benchmark where the GPU excels the CPU performance and it can be observed in the resultant work-load distributions where the biggest CPU ratio is 8.2% for the smallest input, and it shows a tendency to decrease as the input size increases. Despite the small work-load, the CPU still proved to be useful, achieving significant speedups in the performance of this benchmark, for both 1 and 2 GPU executions.

| Benchmark | Input type | Input argument | 1 GPU | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Execution time (ms) (GPU Only) | 50/50 Training | | | CPU assisted GPU Training | | |
| | | | | Configuration (fission/overlap) | Execution time (ms) | Distribution (GPU/CPU) | Configuration (fission/overlap) | Execution time (ms) | Distribution (GPU/CPU) |
| Filter pipeline | Image size (pixels) | 1024x1024 | 1.97 | L2/3 | 1.10 | 91.8/8.2 | L1/3 | 1.08 | 92.5/7.5 |
| | | 2048x2048 | 5.10 | L3/4 | 3.17 | 92.9/7.1 | none/4 | 3.17 | 93.8/6.3 |
| | | 4096x4096 | 16.80 | none/4 | 12.59 | 93.8/6.3 | none/4 | 12.38 | 93.8/6.3 |
| FFT | Size of data-set | 128MB | 35.28 | L2/3 | 12.42 | 32.8/67.2 | L2/4 | 11.82 | 37.5/62.5 |
| | | 256MB | 67.83 | L2/4 | 25.01 | 31.3/68.7 | L1/4 | 23.40 | 30.0/70.0 |
| | | 512MB | 88.93 | L1/3 | 51.28 | 37.1/62.9 | L2/1 | 51.06 | 15.0/85.0 |
| NBody | Number of bodies | 16384 | 37.17 | - | - | - | L1/1 | 35.75 | 95.0/5.0 |
| | | 32768 | 101.56 | - | - | - | L2/1 | 101.55 | 97.5/2.5 |
| | | 65536 | 356.85 | - | - | - | L2/1 | 356.78 | 98.8/1.2 |
| Saypy | Number of elements | $1\times10^6$ | 2.56 | L1/2 | 0.87 | 41.4/58.6 | L2/2 | 0.91 | 37.5/62.5 |
| | | $10\times10^6$ | 14.91 | L1/2 | 8.15 | 45.3/54.7 | none/4 | 8.20 | 67.5/32.5 |
| | | $50\times10^6$ | 72.86 | L1/3 | 40.34 | 43.8/56.3 | L1/3 | 37.31 | 47.5/52.5 |
| Segmentation | Number of elements | 1MB | 0.79 | none/2 | 0.36 | 59.9/40.1 | none/1 | 0.35 | 55.0/45.0 |
| | | 8MB | 2.88 | none/4 | 1.32 | 81.3/18.7 | L3/4 | 1.32 | 78.8/21.2 |
| | | 60MB | 16.70 | none/4 | 9.42 | 82.6/17.4 | L1/4 | 9.27 | 78.8/21.2 |

Table 5.3: Benchmark execution on system $S_2$ using 1 GPU

| Benchmark | Input type | Input argument | 2 GPUs | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | Execution time (ms) (GPU only) | 50/50 Training | | | CPU assisted GPU Training | | |
| | | | | Configuration (fission/overlap) | Execution time (ms) | Distribution (GPU/CPU) | Configuration (fission/overlap) | Execution time (ms) | Distribution (GPU/CPU) |
| Filter pipeline | Image size (pixels) | 1024x1024 | 1.12 | L3/2 | 0.79 | 94.6/5.4 | L1/3 | 0.78 | 98.8/1.2 |
| | | 2048x2048 | 3.84 | L3/4 | 1.90 | 96.1/3.9 | none/3 | 1.92 | 98.8/1.2 |
| | | 4096x4096 | 11.76 | none/4 | 6.63 | 96.9/3.1 | none/4 | 6.67 | 97.5/2.5 |
| FFT | Size of data-set | 128MB | 23.76 | L1/4 | 9.42 | 59.8/40.2 | L2/3 | 10.09 | 52.5/47.5 |
| | | 256MB | 43.12 | L1/4 | 19.07 | 58.6/41.4 | L2/4 | 19.47 | 55.0/45.0 |
| | | 512MB | 77.21 | L1/4 | 42.93 | 56.3/43.8 | L1/4 | 136.71 | 57.5/42.5 |
| NBody | Number of bodies | 16384 | 29.87 | - | - | - | L3/1 | 29.44 | 98.8/1.2 |
| | | 32768 | 69,63 | - | - | - | L2/1 | 69.61 | 98.8/1.2 |
| | | 65536 | 200.76 | - | - | - | L2/1 | 200.81 | 98.8/1.2 |
| Saypy | Number of elements | $1\times10^6$ | 1.59 | none/2 | 0.78 | 75.0/25.0 | L1/2 | 0.69 | 67.5/32.5 |
| | | $10\times10^6$ | 10.97 | L3/4 | 5.20 | 87.5/12.5 | none/4 | 5.10 | 88.8/11.2 |
| | | $50\times10^6$ | 46.84 | L3/4 | 28.10 | 85.2/14.8 | L1/4 | 28.86 | 77.5/22.5 |
| Segmentation | Number of elements | 1MB | 0.72 | none/1 | 0.31 | 69.5/30.5 | none/2 | 0.35 | 85.6/14.4 |
| | | 8MB | 1.87 | none/3 | 0.97 | 88.3/11.7 | none/2 | 1.01 | 88.8/11.2 |
| | | 60MB | 10.75 | none/4 | 5.43 | 93.0/7.0 | L1/4 | 5.84 | 88.8/11.2 |

Table 5.4: Benchmark execution on system $S_2$ using 2 GPU

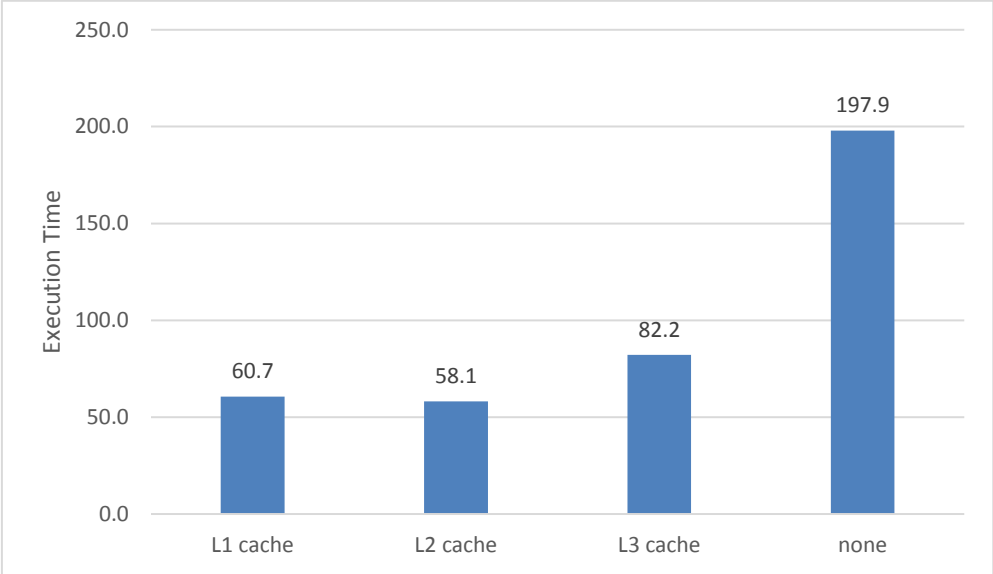


Figure 5.3: Execution times measured during the training of FFT with 256 MB input

**FFT** This benchmark is one of the cases where the CPU outperforms the GPU, possibly caused by the fact that it is the benchmark with the biggest input size, increasing the cost of the communication with the GPU, but also the nature of the computations, containing exponential and trigonometric operations, computations where the CPU excels as well. For 1 GPU executions, only a bit over 30% of the work-load is assigned to the GPU which explains the speedups over 2.5 when executing with the CPU. With 2 GPUs, the amount of work assigned to the GPUs grows until almost 60%. In both 1 and 2 GPU executions, the speedup shows a tendency to decrease as the input size increases. The fact that each GPU has its own PCIe bus improves the scalability from one GPU to two GPU devices. Given the increased amount of data communication that this benchmark requires, if the GPU devices shared the same PCIe bus, it would be expected that the dual GPU training partitions would show a smaller work-load assigned to the GPUs.

One interesting fact lies in the 512MB executions. The work-load distribution is very different in the two training modes, however, the resulting execution times are close to each other. This shows that, for some applications and input data sizes, there may be more than one viable work-load partitioning to achieve the same level of performance.

**N-body** This benchmark is not suitable to be trained under the *50/50 split* training mode, due to the existence of a synchronized loop. Running this benchmark under this training mode, would take a lot of time and the performance would be too degraded, as the tendency (shown during the development) was to keep transferring work-load to the CPU group, as the global performances decay. Therefore, we opted to not include those results in the evaluation. Under the *CPU assisted GPU execution* training mode, the results show no significant speedups when the CPU enters the scene. The fact that the


Figure 5.4: Speedup for 1 GPU executions

global synchronization is always performed in the CPU, explains the low work-load assigned to this device. Since the CPU has little to no impact in the OpenCL executions, the speedup is, as expected, close to 1, meaning that for this benchmark, we could not profit from the CPU computational power. However, no overhead was introduced with the use of the CPU for OpenCL computations. Despite the fact that we did not profit from using the CPU in this particular benchmark, this cannot be generalized to every globally synchronized loop skeletons. We expect that, with some lighter computations, the CPU may be of use to help improve the performance by taking care of a part of the computation, as long as it does not compromise the synchronization process.

**Saxpy** For 1 GPU, we achieved close to linear speedup for the bigger inputs. The results are as expected, since the GPU/CPU work-load ratio is close to an even work-load distribution (approximately 45%/55% for the T1 training mode and slightly further apart in the T2 mode), meaning that the work-load of the GPU was reduced by more than an half (except in one case in T2). The execution with the smallest input, achieved a speedup higher than 2.5 in both training modes. A little over the expected, but this occurrence can be justified by the fact that Saxpy is more data-bound than computation-bound and due to the architectural nature of the GPU, some work size dimensions may fit the device better. This shows us that this empirical approach can be also useful to find work size partitions that are more GPU friendly.

For 2 GPUs, the speedup achieved for the smaller inputs is a little higher than 2, which is a little unexpected, since only 25% or less of the work-load is performed by the


Figure 5.5: Speedup for 2 GPU executions

CPU (expect in one case in T2). Nonetheless, as the number of elements decreases, the speedup also decreases to more expected values. Similarly to the 1 GPU execution, this can be justified by Saxpy being a data-bound problem and due to the architecture of the GPU devices.

Like in the FFT benchmark, also in Saxpy with the 10 million elements input, with 1 GPU, both trainings identified two different configurations and work distributions, but the yielded execution time were also pretty close.

**Segmentation** This is another case where the GPUs outperform the CPU, but a significant speedup can be achieved by taking advantage of the latter device. Compared to the Image Filter Pipeline benchmark, this benchmark has a higher data-to-computation ratio, meaning that the GPU communication overhead has more impact in the GPUs' overall execution time. Akin to the Saxpy benchmark, we believe that the training is balancing the GPU work-load to avoid prompting executions with few data to compute, where the communication overhead would not be worth, thus, explaining the higher than expected speedup.

**Final Remarks** These results confirm that it can be profitable to exploit the combination of CPU and GPU devices in the execution of parallel computations, when compared to GPU-only computations. It may also be observed that the best work distribution configuration is application dependent, and even data-set size dependent within applications. This fact validates our application-targeted training approach, since having a general, application oblivious approach, requires an empirical method like the offline training we implemented. Another interesting fact about these results is that in some data-bound

benchmarks, the speedup achieved was even better than the expected, considering the amount of work-load that was distributed to the CPU. This fact is more noticeable for smaller inputs, where the PCIe transmission overhead has more impact on the performance, given the small amount of data to compute. In general, this overhead is mitigated for bigger inputs and the GPU executions prove themselves more advantageous. We assign this to the fact that, some partition sizes are more friendly of the GPU's specific architecture, achieving shorter execution times. As a final remark, we can observe that the number of overlapping partitions that display the best performance is not the same for different input sizes of the same benchmarks, but its tendency is to increase with the increase of the input size. This observation can be useful for the future development of a performance model that is able to infer the performance of a computation when in the presence of alterations to the input data size. As for the OpenCL device fission level, the ideal fission level does not follow a perceptible pattern. Even for the same benchmark, the training picked different fission configurations for the same input, in the different training modes.

By comparing the results of each training mode, there is no absolute winner on the best approach, as they both achieve close performance levels for the same benchmarks and data-sizes. Although, for each execution of the benchmarks, one of the training methods had a slightly better performance, we cannot conclude that one approach is better than the other, as the best performing training method is not always the same, even for the same benchmark with different input sizes. The only exception is in the N-Body benchmark and this can be generalized to every application with a `Loop` skeleton in its computational tree, as the *50/50 split* training method is not suitable for these skeletons, making the *CPU assisted GPU execution* the best training approach for this applications.

## 5.6 Training Evaluation

From the results of the previous section, we can conclude that we can achieve significant speedups by combining the computational power of CPU and GPU devices. Our *50/50 split* training method achieves these speedups by balancing the execution times of each device group. In this section, we evaluate the accuracy of that balancing. To do so, we slightly modified the framework to announce the average value of the *performanceRatio* (Equation 4.3, part of the dynamic load balancing, detailed in Section 4.5.1.3), at the end of all executions. We do not consider the *CPU assisted GPU execution* training method for this evaluation, as it lies on a different approach, based on the global performance. Therefore, the individual performance of each device group is not a useful information to compare the effectiveness of the method. All the benchmarks were executed using only one GPU device (with a *50/50 split* training execution) and the average *performanceRatio* is expressed in Table 5.5.

As the results show, the ratio between the two devices was always greater than 0.8, going from 0.825 to 0.919. We consider that these values validate the premise that this

| Benchmark | Input parameter | $performanceRatio$ |
|---|---|---|
| Saxpy | $1\times10^6$ | 0.885 |
| | $10\times10^6$ | 0.919 |
| | $50\times10^6$ | 0.874 |
| Segmentation | 1MB | 0.979 |
| | 8MB | 0.863 |
| | 60MB | 0.832 |
| Filter pipeline | 1024x1024 | 0.842 |
| | 2048x2048 | 0.855 |
| | 4096x4096 | 0.846 |
| FFT | 128MB | 0.846 |
| | 256MB | 0.825 |
| | 512MB | 0.841 |

Table 5.5: Average benchmark duration for each

training method lies upon. From this values, we can also infer that 0.8 is a nice value for the configurable *balancingRatio* parameter, to identify unbalanced executions. For some benchmarks like Saxpy, even higher values can be used since lowest *performanceRatio* average value we obtained was 0.874.

## 5.7 Work-load Derivation and Dynamic Balancing

To evaluate the capacity of our dynamic work-load balance approach, we performed two tests with our framework:

- A test where we performed a training execution for one input size and the remaining input sizes are subject to the work-load derivation and the dynamic work-load balance;
- A test where we intentionally increase and decrease the CPU's work-load during a benchmark's execution, and analyze the changes in the application's work-load distribution.

### 5.7.1 Image Pipeline work-load distribution derivation

For this test, we used the Image Pipeline benchmark and we performed a training for one of the input sizes and submitted different inputs without activating the training. Our goal is to compare the performance of the work-load distribution derived from the previously executed input sizes (with possible dynamic rebalances or retraining), with the performance achievable with a training for each of the inputted sizes. The *balancingRatio* was set to 0.85. Recall that, once derived, the work-distribution information is stored in the *Knowledge Base*, and some inputs may derive from the previously derived work-load distributions.

| Image id | Image size | Training result | | | | |
|---|---|---|---|---|---|---|
| | | Fission | Overlap | GPU (%) | CPU (%) | Execution time |
| Image 1 | 1024x1024 | L3 | 3 | 90.8 | 9.2 | 1.10 |
| Image 2 | 512x512 | L3 | 2 | 87.5 | 12.5 | 0.54 |
| Image 3 | 1024x2048 | L1 | 4 | 91.5 | 8.5 | 1.74 |
| Image 4 | 2048x512 | L2 | 3 | 89.8 | 10.2 | 1.06 |
| Image 5 | 2048x2048 | none | 4 | 92.9 | 7.1 | 3.17 |
| Image 6 | 4096x4096 | L3 | 4 | 93.8 | 6.3 | 12.59 |

Table 5.6: Filter Pipeline: performance obtained from the training's results

| Image id | Image Derivation Nearest neighbor | Required balancing | | Resulting distribution | | Execution time | Relative performance |
|---|---|---|---|---|---|---|---|
| | | load balance | training | GPU (%) | CPU (%) | | |
| Image 2 | Image 1 | 6 | 1 | 81.0 | 19.0 | 0.64 | 84% |
| Image 3 | Image 1 | 2 | 0 | 90.7 | 9.3 | 1.87 | 93% |
| Image 4 | Image 1 | 4 | 0 | 90.6 | 9.4 | 1.07 | 100% |
| Image 5 | Image 3 | 1 | 0 | 90.6 | 9.4 | 3.48 | 91% |
| Image 6 | Image 5 | 0 | 0 | 91.8 | 8.2 | 13.41 | 94% |

Table 5.7: Filter Pipeline: performance obtained from the derivation of the work-load distribution from past executions

As a comparison point, Table 5.6 shows the execution times of each input, when executed with a previous training execution. Table 5.7 shows the results of this evaluation test. The images were submitted in the order they appear in the table, meaning that each input could derive its work-load distribution from the input sizes that were submitted before.

As we can see from the relative performance values, although the derived work-load distributions do not reach the same performance levels achieved with a training, the execution times of both executions came pretty close, even taking into consideration that the derived fission/overlap configuration is not always the ideal one, found during training. Most of the derived work-load distributions performed with more than 90% of the performance of its ideal distribution and configuration. Also, the rebalancing mechanism proved to be effective in this test-case, as a balanced distribution could be found after a few rebalance attempts and only one of the cases required a retraining to find the work-load balance.

### 5.7.2 Reaction to system's load changes

For this test case we intentionally loaded and unloaded the CPU during the execution of a benchmark and observed the changes in the work-load distribution. It required some minor changes in the framework so the new work-load is announced anytime the dynamic load balance is triggered.

To make the system's load fluctuate, we designed a simple application (that is further referred to as LFA), that spawns multiple threads that runs a loop of algebraic computations. For this test case, we used 12 threads, to match the number of the CPU's hardware

threads. The selected benchmark was the Saxpy, with the 50 million elements input, because it is one of the benchmarks that distributed a big part of the work to the CPU (more than 50%).

We started this test case by training the benchmark using one GPU and without any kind of user submitted system load. The obtained partitioning was 49.61% to the GPU and 50.39% to the CPU, with L1 cache fission and 4 overlapping partitions. The average *partitioningRatio* was 0.89, therefore, we configured the framework to trigger the dynamic load balance at 0.85. We then started the benchmark, still without any system's load interference, and observed that the dynamic load balance was not triggered. Then, still with the benchmark running, we started our load fluctuation application and waited until the work-load was rebalanced. After some iterations without the dynamic load balance being triggered, we terminated the load fluctuation application and again, waited until the balance was reestablished.

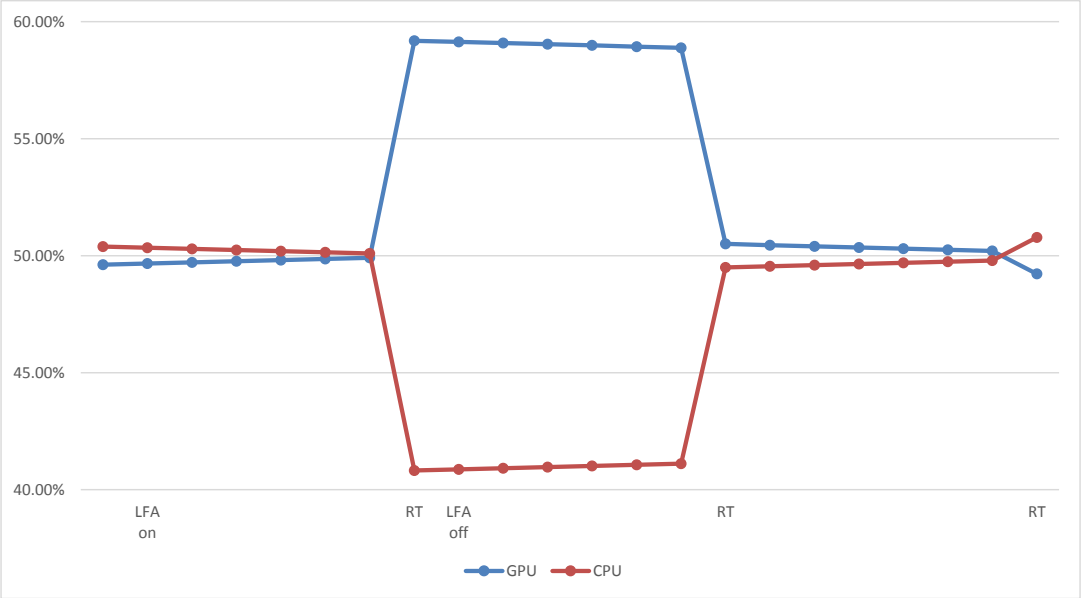


Figure 5.6: Dynamic rebalance to system's load fluctuations

Figure 5.6 displays the work-load modifications along this test case. The points labeled with *LFA on* and *LFA off* represent the first load-balance after the load fluctuation application was executed and terminated, respectively. The points labeled with RT are the result of a retraining, being the remaining results of a simple load adjustment.

As the results show, the application reacted to the load variation and executed 6 load rebalances until the retraining triggered, stabilizing the work-load distribution at 58.18% to the GPU and 40.82% to the CPU. After the retraining, the work-load remained balanced until the load fluctuation application was terminated. Similarly to the previous case, the framework tried another 6 consecutive load balances until the retraining triggered. However, the resultant work-load distribution of this retraining was not sufficiently balanced

because the dynamic load balance kept triggering, demanding another retraining iteration to kick in and bringing the work-load to a balanced state with 49.22% of the work assigned to the GPU and the remaining 50.78% to the CPU.

## 5.8 Final Remarks

In Section 5.1, we introduced the questions that we wanted to answer with the evaluation of our work. In this section, we answer those questions based on the results we obtained in our evaluation.

1. Can we take advantage of the OpenCL fission feature to increase the performance of CPU OpenCL computations?

The results of Section 5.4 show that the OpenCL fission can be profitable to achieve higher performance levels, not only to take advantage of data locality in the different CPU's local memory, but also to profit from cache level locality inside each CPU. This result is further validated by the results of Section 5.5, by the fact that the training process selected, fission configurations over the no-fission configuration in some of the benchmarks. With that in mind, it is safe to say that we can take advantage of the OpenCL fission functionality, but when to use it and for what dimensions the CPU should be partitioned is still difficult to predict.

2. Can we take advantage of the CPU to increase the performance in a heterogeneous system composed by GPUs and CPUs?

Section 5.5 compares the performance of the benchmarks using just the GPUs with the performance of combined executions of GPUs and CPUs. For both one and two GPUs executions, the speedups obtain were pretty satisfying, yielding good speedups in every benchmark, except for the N-Body. We consider our approach as a good method to achieve balance over the devices, as all the work distribution concerns are delegated on the platform.

3. How accurate is the our training method approach?

We analyzed the *50/50 split* training method in Section 5.6, to compare the execution times of each device group. The results were pretty close, with ratios over 0.8 when comparing the fastest group with the slowest. We did not evaluate the *CPU assisted GPU execution* training method, as it relies on a different premise, but given the similar speedups obtained in Section 5.5, we consider both approaches equally valid, as we cannot reject one over the other, except when the computational tree has a `Loop` skeleton, with global synchronization, in which case, the *CPU assisted GPU execution* is clearly the best approach.

4. How does the framework respond to alterations to the input data sizes and the system's work-load?

In Section 5.7 we evaluate how the platform reacts to inputs of different sizes and to system load fluctuations. The platform reacted pretty well to inputs with different sizes on the Image Pipeline benchmark, achieving balance after a few rebalancing iterations and, in one case, after rerunning the training for the current configuration. Although not optimal, the performance achieved by the dynamic rebalancing mechanism was considerably close to the achievable performance with the best fission/overlapping configuration.

When under the influence of load from external applications, the framework quickly detected the load fluctuations and triggered the work-load rebalance mechanism that balanced the load into a more stable state.

# 6

# Conclusion

## 6.1 Goals and Results

The main goal of this thesis was to extend a multi-GPU algorithmic skeleton framework, to combine the power of the GPUs with the power of the CPU for the skeletal computations, improving the performance of a system while maintaining the same level of abstraction provided by the skeletons. To this extent, we extended the Marrow framework, introducing the support for CPU computations as well as dynamic work balance among devices.

Unlike the previous version of Marrow where the work distribution among the GPU devices was achieved statically, based on performance benchmarks results obtained during the install process. In this version, we had to adopt a more dynamic way to distribute the work-load among CPUs and GPUs, due to the inability to achieve optimal work balance before actually executing the target application. To guarantee that the application runs with the best execution and platform configuration possible, the work balance is achieved through a number of offline training executions, where the performance of various work distributions, overlapping partitioning and OpenCL fission combinations, are measured to prepare the platform to execute the applications requests with the best performing configuration found.

The work-load distribution is achieve through the replication of the computational trees over the different devices and the data is partitioned among the devices based on the training results. This way, the data locality is guaranteed between different kernel executions on the same tree, minimizing device communication. During the training stage, the data partitioning is based on one of the two training modes that we designed, that select the work distribution to test, based on each devices performance on the previous

training iteration, aiming to achieve a better partitioning along the training process.

The evaluation of our prototype shows that CPU and GPU devices have different performance ratios depending on each benchmark computations and data-sizes. Despite that, the training process was always able to find a balanced work distribution for each tested application, significantly improving its performance when compared to GPU-only executions.

Overall, we fulfilled all our goals, achieving integration and cooperation between CPU and GPU devices in Marrow, while abstracting the programmer from the underlying heterogeneity.

A paper presenting part of the work developed in this thesis has been accepted for presentation at the International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2014), and will be subsequently published in EuroPar's workshop companion [45].

## 6.2 Future Work

In this work we addressed heterogeneity between GPU and CPU devices. One interesting next step could be to consider support for other heterogeneous devices which would introduce another work balance dimension, increasing the complexity of our current training approach. This added complexity may derail the current training approach and a more refined approach may be necessary.

Currently, the kernel functions offloaded to the devices are expressed in native OpenCL language. To maintain the compatibility of those kernels among devices, we had to abdicate from kernel-level device specific optimizations which would imply different implementations for different devices. In the future, this limitation can be surpassed by creating a higher level layer for kernel programming, allowing the framework to internally perform those device specific optimizations over each generated kernel, while, at the same time, providing a friendlier programming interface to the programmer. This added abstraction layer can also open the way for the support of different execution backends, other than OpenCL.

In this version, we only focused on applications with a single computational tree. To further increase the range of programmable scenarios, given that some applications may require the definition of more than one tree, in a future work it would be interesting to provide support for work-load adaptation considering multiple computational trees.

Our approach to the work-load distribution derivation has a linear complexity, which means that with the increase of the number of work-load distributions for different data sizes in the *Knowledge Base*, the amount of comparisons during the interpolation will also increase. This module can be improved in the future, by reducing the complexity of the derivation, and also by exploring other approaches to derivate the work-load distribution, other than the Euclidean distance.

# Bibliography


[1] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in heterogeneous computing," *Sci. Program.*, vol. 18, pp. 1–33, Jan. 2010.

[2] M. Steuwer, P. Kegel, and S. Gorlatch, "Skelcl - a portable skeleton library for high-level gpu programming," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '11, (Washington, DC, USA), pp. 1176–1182, IEEE Computer Society, 2011.

[3] 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, (New York, NY, USA), pp. 5–14, ACM, 2010.

[4] AMD, "Bolt c++ template library," Last visited in June 2013.

[5] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU Computing Gems, Jade Edition, Edited by Wen-mei W. Hwu*, 2011.

[6] R. Marques, H. Paulino, F. J. M. Alexandre, and P. D. Medeiros, "Algorithmic skeleton framework for the orchestration of gpu computations," in *Euro-Par 2013 Parallel Processing - 19th International Conference, Euro-Par 2013, Aachen, Germany, August 26-30, 2013. Proceedings* (D. a. M. Felix Wolf, Bernd Mohr, ed.), no. 8097 in Lecture Notes in Computer Science, pp. 874–885, Springer-Verlag, 08 2013.

[7] *On the Support of Task-Parallel Algorithmic Skeletons for Multi-GPU Computing*, (Gyeongju, South Korea, March 24-28, 2014), ACM, 2014.

[8] M. Cole, *Algorithmic skeletons: structured management of parallel computation*. Cambridge, MA, USA: MIT Press, 1991.

[9] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Softw. Pract. Exper.*, vol. 40, pp. 1135–1160, Nov. 2010.

[10] NVIDIA, "Cuda 5." http://www.nvidia.com/object/cuda_home_new.html, May 2013.

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, pp. 777–786, Aug. 2004.

[12] NVidia, "Nvidia cuda." http://www.nvidia.com/object/cuda_home_new.html, May Last visited: June 2013.

[13] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.2*, 8 December 2012.

[14] T. D. Han and T. S. Abdelrahman, "hicuda: a high-level directive-based language for gpu programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, (New York, NY, USA), pp. 52–61, ACM, 2009.

[15] C. Nugteren and H. Corporaal, "Introducing 'bones': a parallelizing source-to-source compiler based on algorithmic skeletons," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, (New York, NY, USA), pp. 1–10, ACM, 2012.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, Oct. 2005.

[17] 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, (New York, NY, USA), pp. 8:1–8:10, ACM, 2011.

[18] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, Aug. 2007.

[19] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzarán, and D. Padua, "Performance portability with the chapel language," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, (Washington, DC, USA), pp. 582–594, IEEE Computer Society, 2012.

[20] C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink, "Compiling a high-level language for gpus: (via language support for architectures and compilers)," *SIGPLAN Not.*, vol. 47, pp. 1–12, June 2012.

[21] C. NVIDIA, CAPS and PGI, "Openacc." http://www.openacc-standard.org, Last visited: June 2013.

[22] PGI, "Pgi accelerator compilers with openacc directives." http://www.pgroup.com/resources/accel.htm, Last visited: June 2013.

[23] CAPS, "Caps compilers." http://www.caps-entreprise.com/products/caps-compilers/, Last visited: June 2013.

[24] uccULL, "The openacc research implementation." http://accull.wordpress.com/, Last visited: June 2013.

[25] 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," tech. rep., Toronto, Canada, June 2010.

[26] S. Ernsting and H. Kuchen, "Algorithmic skeletons for multi-core, multi-gpu systems and clusters," *Int. J. High Perform. Comput. Netw.*, vol. 7, pp. 129–138, Apr. 2012.

[27] J. Hoberock and N. Bell, "Thrust: A parallel template library," Last visited in June 2013. Version 1.7.0.

[28] AMD, "Aparapi." http://developer.amd.com/tools-and-sdks/heterogeneous-computing/aparapi/, Last visited: June 2013.

[29] A. Matoga, R. Chaves, P. Tomas, and N. Roma, "A flexible shared library profiler for early estimation of performance gains in heterogeneous systems," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pp. 461–470, July 2013.

[30] U. Dastgeer, L. Li, and C. Kessler, "Adaptive implementation selection in the skepu skeleton programming library," in *Advanced Parallel Processing Technologies* (C. Wu and A. Cohen, eds.), vol. 8299 of *Lecture Notes in Computer Science*, pp. 170–183, Springer Berlin Heidelberg, 2013.

[31] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 187–198, Feb. 2011.

[32] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, (London, UK, UK), pp. 179–196, Springer-Verlag, 2002.

[33] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, "Scalable framework for mapping streaming applications onto multi-gpu systems," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 1–10, ACM, 2012.

[34] J.-F. Dollinger and V. Loechner, "Adaptive Runtime Selection for GPU," in *42nd International Conference on Parallel Processing*, (Lyon, France), pp. 70–79, IEEE, Oct. 2013.

[35] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 25:1–25:11, IEEE Computer Society Press, 2012.

[36] J. Colaço, A. Matoga, A. Ilic, N. Roma, P. Tomás, and R. Chaves, "Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems," in *Parallel Processing and Applied Mathematics* (R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, eds.), Lecture Notes in Computer Science, pp. 693–703, Springer Berlin Heidelberg, 2014.

[37] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 245–256, IEEE Press, 2013.

[38] M. Leyton and J. M. Piquer, "Skandium: Multi-core programming with algorithmic skeletons," in *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, (Washington, DC, USA), pp. 289–296, IEEE Computer Society, 2010.

[39] D. Caromel and M. Leyton, "Fine tuning algorithmic skeletons," in *Proceedings of the 13th international Euro-Par conference on Parallel Processing*, Euro-Par'07, (Berlin, Heidelberg), pp. 72–81, Springer-Verlag, 2007.

[40] G. Contreras and M. Martonosi, "Characterizing and improving the performance of intel threading building blocks," *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 57–66, 2008.

[41] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, pp. 212–223, May 1998.

[42] F. Alexandre, "Runtime Support for Multi-GPU Computations in an Algorithmic Skeleton Framework," Master's thesis, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2013.

[43] B. R. Gaster, "OpenCL device fission." `https://www.khronos.org/assets/uploads/developers/library/2011_GDC_OpenCL/AMD-OpenCL-Device-Fission_GDC-Mar11.pdf`, 2011. Published by Khronos Group.

[44] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, (New York, NY, USA), pp. 63–74, ACM, 2010.

[45] F. Soldado, F. Alexandre, and H. Paulino, "Towards the transparent execution of compound opencl computations in multi-cpu/multi-gpu environments," in *Euro-Par 2014: Parallel Processing Workshops*, Lecture Notes in Computer Science, Springer, 2014. to appear.