Frederico Mariano de Almeida Marques

BSc in Computer Science

High-level Programming of Many-Core Architectures

Dissertação para obtenção do Grau de
Mestre em Engenharia Informática

Orientador: Professor Doutor Hervé Paulino, Professor Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Professor Doutor Artur Dias

Vogais: Professor Doutor Francisco Dias

Professor Doutor Hervé Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

June, 2016

High-level Programming of Many-Core Architectures

Copyright © Frederico Mariano de Almeida Marques, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

To my girlfriend for all the loving support. To my nephew and niece.

Acknowledgements

I would like to start by thanking Professor Hervé Paulino, without whom this work would not be possible. His dedication, patience and constant availability were fundamental to my thesis. I would like to single out Ana Pereira's indirect contributions to my thesis; the long hours spent discussing and sharing ideas, the patience and the continued support were the cornerstone that allowed me to reach this mark in my life. Obrigado.

ABSTRACT

Nowadays computational systems are heterogeneous typically containing CPUs and GPUs. Offloading computations to a GPU allows more computations per time unit, specially when GPUs are more efficient than CPUs for some type of computations, like matrix multiplication. There is, nonetheless, a specific complexity associated with writing portable code, specially when dealing with the integration of the programming of GPUs in mainstream programming languages, avoiding low-level complex frameworks like CUDA and OpenCL.

The proposals put forth to solve these questions can be divided into two fields: specialized skeleton frameworks with reduced expressiveness, and languages (or language extensions) that follow a best-effort strategy. Nonetheless, efficient usage of these languages requires the programmer reasons about code compatibility while considering the GPUs execution model and the limitations of the compilers themselves.

The herein documented approach takes a different route, offering a programming model in which the programmer expresses the GPU computations by manipulating data structures in a way that is equivalent to the computations designed for CPU. This expressiveness is achieved without prejudice to the guarantee that the expressed computations will run in GPU context.

The methodology chosen to implement this approach rests on an expression template layer. This expression template layer generates both the Marrow framework code, to launch the OpenCL kernels, and the OpenCL kernels to be run. The generated code is reused in subsequent executions being that the generation only happens at the first time a computation is to be computed. From the tests developed there was a significant reduction of lines of code. The technique's overhead has a reduced impact in the computation time, as shown by our results, being that the most noticeable feature is that the overhead is not proportional to the expressions complexity; our results point to an overhead of 9 to 10% in worst case scenario, and approaching 0% as the OpenCL computational time increases.

Keywords: Expression templates, Algorithmic skeletons, Heterogeneous Computation, Marrow

RESUMO

Actualmente os sistemas computacionais são heterogéneos, tipicamente contendo CPUs e GPUs. Executar o offload de computações para o GPU permite executar mais computações por unidade de tempo, especialmente quando se sabe que os GPUs são mais eficientes que os CPUs para alguns tipos de computação tais como a multiplicação de matrizes. Para evitar a complexidade associada a frameworks de baixo nível, tal como CUDA e OpenCL, procura-se a integração da programação de GPUs em linguagens mainstream. Contudo a escrita de código portável, abstraindo dos detalhes da programação de GPU tem uma complexidade própria.

As propostas feitas para solucionar estas questões podem ser divididas em dois campos: bibliotecas de skeletons muito especializadas e, por desenho, de expressividade muito limitada e linguagens ou extensões de linguagens que adoptam uma filosofia de melhor esforço. No entanto, a utilização eficiente destas linguagens requer que o programador raciocine sobre a compatibilidade do seu código com o modelo de execução dos GPUS e sobre a limitações do próprios compiladores.

A abordagem aqui documentada é diferente: oferece um modelo de programação em que o programador exprime as computações GPU manipulando estruturas de dados de forma equivalente às computações para CPU. Esta expressividade é atingida sem prejudicar a garantia que as computações são executadas em contexto GPU. Para a solução conseguir fazer o pretendido, utiliza-se a framework Marrow para executar os kerneis OpenCL.

A metodologia para implementar a abordagem assenta numa camada de expression templates. A camada de expression templates gera o código de biblioteca e o código de backend, que no nosso caso mapeiam para código Marrow e código OpenCL. O código gerado é reutilizado de execuções anteriores, sendo que a geração ocorre na primeira execução da computação. Dos testes efectuados detectou-se uma redução das linhas de código, reduzindo o código em OpenCL e Marrow em 100%. O overhead da técnica escolhida tem um impacto reduzido, tal como comprovam os testes efectuados, sendo que o overhead é não-proporcional à complexidade da expressão avaliada; os resultados apontam para que o overhead tenha um peso entre 9% a 10% no pior caso, e tendendo para 0% à medida que a computação OpenCL consome mais tempo.

Palavras-chave: Esqueletos Algorítmicos, Computação Heterogénea, Marrow

Contents

Contents xiii

List of Figures xv

List of Tables xvii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Problem . . . 2
- 1.3 Work proposal . . . 4
- 1.4 Contributions . . . 4
- 1.5 Document structure . . . 5

2 State of the art 7
- 2.1 Introduction . . . 7
- 2.2 Algorithmic skeleton frameworks . . . 7
  - 2.2.1 What are algorithmic skeletons . . . 7
  - 2.2.2 Skeleton frameworks for heterogeneous computing . . . 9
  - 2.2.3 SkeTo . . . 9
  - 2.2.4 SkePU . . . 10
  - 2.2.5 SkelCL . . . 10
  - 2.2.6 Marrow . . . 11
  - 2.2.7 Bolt . . . 12
  - 2.2.8 Thrust . . . 12
- 2.3 Embedded linguistic support . . . 15
  - 2.3.1 Dandelion . . . 15
  - 2.3.2 Lime . . . 16
  - 2.3.3 StreamIt . . . 19
  - 2.3.4 OpenACC . . . 20
- 2.4 Polyhedral model . . . 22
- 2.5 Expression templates . . . 22
- 2.6 Discussion . . . 23

3 Marrow Expressions 25

3.1 General overview . . . 25
3.2 Marrow vector . . . 27
3.3 Abstract syntax tree . . . 28
3.4 Runtime system . . . 32
3.4.1 Code generation and Marrow dispatching . . . 33
3.5 AST submission and kernel execution . . . 35

4 Kernel fusion **37**

5 Evaluation **41**
5.1 Case studies . . . 41
5.2 Metrics . . . 42
5.3 Methodology . . . 42
5.4 Abstraction . . . 42
5.5 Overhead . . . 43
5.6 Final remarks . . . 44

6 Conclusion **47**
6.1 Objectives and results . . . 47
6.2 Future work . . . 48

Bibliography **51**

7 Appendix **55**
7.1 Marrow saxpy implementation . . . 55
7.2 Marrow expression saxpy implementation . . . 57
7.3 Marrow sqrt implementation . . . 58
7.4 Marrow expression sqrt implementation . . . 61
7.5 Marrow twolevels implementation . . . 62
7.6 Marrow expression twolevels implementation . . . 64
7.7 Marrow threelevels implementation . . . 65
7.8 Marrow expression threelevels implementation . . . 68

List of Figures

3.1 The application code generates orchestration code and back-end code. The generated orchestration code will execute the back-end kernel. 26
3.2 The application code, which contains the Marrow Expressions, generates **Marrow** and **OpenCL** code. The generated **Marrow** code will execute the **OpenCL** kernel. 26
3.3 Operation capture. 28
3.4 Runtime overview. 32
3.5 The compiler visiting a tree 34

5.1 Overhead of level-three. 43
5.2 Overhead of level-four. 44
5.3 Overhead of level-five. 44

List of Tables

2.1 Table with skeleton support from the frameworks discussed. 13

3.1 Examples of Marrow calls for some host code lines. 35

CHAPTER 1

INTRODUCTION

1.1 Motivation

The strategy chosen in the development of processing units over the years has been to augment the number of instructions executed by time-slice. The main approach to that effect has been the shrinking of components, compacting more processing power into less space. Moore's law was a decisive blow in this approach, stating that the circuit would shrink by half every two years. Heat generated by the semi-conductors in increasingly smaller environment would prove to be a problem to be dealt with, when attempting to pack more processing power in smaller ICs.

With the advent of increasingly powerful specialized programmable processing units came the opportunity of circumventing the heat problem by accelerating execution not by increasing a processor's processing speed but by having a section of the code running in both the central processing units and in auxiliary units present in the system.

FPGAs[1], GPUs[2] and other units that may be present in common desktop and laptop systems remain idle for a larger part of the time-slice, compared with the CPU, and became good candidates for code accelerators.

The power of a chip is based on the number of transistors clammed together on the chip area. The greater the number of transistors, the more powerful the chip is. The general purpose characteristic of the CPUs, however, do constraint the gain of computational power by transistor, since many are dedicated to caching and control flow and other usages.

GPUs can handle more workload, of certain tasks, on a given time-frame than CPUs,

[1]Field programmable gate array – an integrated circuit designed to be configured by the customer after manufacturing

[2]Graphics processing unit – a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display.

given that the latter must dedicate some of their time to thread control, operating system tasking and I/O interrupt handling. However, to harness a GPU's computing capabilities is not trivial since they feature a pure SIMT[3] architecture, having all of its processors executing instructions in parallel. Many research efforts tried to endow the programmer with a convenient way to harness the power of those devices, while hiding the differences of the contrasting execution models associated with those computational devices.

The work presented here aims to provide a high abstraction programming model with which the programmer can take advantage of devices with well-known algorithmic formulations, from a convenient, integrated, syntax.

GPUs dedicate the bulk of their computational power on matrix multiplications, through massively parallel computations. Performance comparison between CPUs and GPUs in scientific calculations identify the advantage of the GPU devices, since the latter are designed taking account floating point calculations. This makes the GPU very useful in data-parallel problems, with more computation than memory transfers. This identification came with the recognition that programming GPU's for other purpose than graphics is cumbersome and requires several specialized tools[1].

Efforts were taken to uniform or simplify software development on such devices and two specialized frameworks called CUDA[2] and OpenCL[3] were developed. CUDA, from NVIDIA Corp is a proprietary technology that, in conjunction with NVIDIA graphic cards and specialized drivers, ease the development of applications that could partially execute on the GPU. The programming environment of CUDA exploited the well established C language, with some primitives added.

OpenCL, in contrast was a joint effort by consortium to produce an uniform an API that would allow a truly portable implementation. The philosophy behind OpenCL is that software built upon it should not be constraint by devices' environment or intricacies. This was an interesting premise since it could potentially solve the portability problems associated with programming to SIMD architectures, FPGA devices and CPUs, ushering a new co-computational heterogeneous programming mindset.

OpenCL, in contrary to CUDA, had several additions to the C language and became rather a subset of C. The potential of this framework was crippled by the added complexity; even an experience programmer would have to invest considerable time in learning the framework before knowing enough to produce consistent results.

One has to add a prologue and an epilogue to every **OpenCL** kernel in order to correctly call any kernel to solve problems.

1.2 Problem

The necessity to add both prologue and epilogue to run any OpenCL kernel hinders reusability of components since the programmer lacks a uniform methodology for writing

[3]Single Instruction, Multiple Thread – one instruction, potentially the same instruction, is executed on all the threads simultaneously

reusable kernels. Other differences remain between the two, for instance, dynamic memory allocation. While in CUDA memory allocation and management is possible, OpenCL specification strictly forbids memory allocation.

Research efforts to lower the complexity in writing software that took advantage of these accelerators was undertaken, and the results can be divided into two major fields: structured and unstructured.

Unstructured or free approaches are those where the programmer maintains the creative freedom in what can be written, however there are no guarantees that the code will execute in an accelerated environment. Structured approaches are those where the programmer loses some creative freedom in what can be written, for the added guarantee that the code will execute in an accelerated context. Examples of those are the skeleton-driven approaches. When writing a skeleton there are a great deal of common programming tasks that are not acceptable (i.e, issuing an output for the process's standard output).

While linguistic-based approaches were proved interesting, skeleton driven frameworks gathered the most adepts. Skeletons, while a compelling way to express algorithms still lacked an uniform methodology to be combined in a structured, abstract manner. In the majority of the works done, it can be observed that the approaches relate to the SIMD [4] property of these devices by using first and second class functions. The first class functions apply the second class ones upon data, which is itself partitioned into smaller chunks and then, attributed to computation units. The second class functions can be envisioned as functors; solutions can be calculated by applying them concurrently to each element of the partition.

Examples of the aforementioned programming model can be found in **SkeTo**[4, 5, 6, 7], **SkelCL**[8], **SkePU**[9, 10, 11], **Marrow**[12, 13, 14], **Thrust**[15] and **Bolt**[16], for skeletons. **Lime**[17, 18], **Dandelion**[19], **StreamIt**[20] and **OpenACC**[21] are examples for the linguistic support.

In all skeleton based approaches, both task- and data-parallel skeletons (or constructs in case of the linguistic approaches) were explored; all the frameworks discussed support data-parallelism, having *map* as a core skeleton. *map* applies a function to each element of a dataset, by executing in parallel each of the instructions on the supplied argument function operate in parallel upon every element of the dataset.

Task-parallelism, on the other hand, is defined by having different threads executing different functions. Examples of such support can be found on **Marrow**, which allows skeleton nesting; skeleton nesting allows for multiple skeletons to be called on the same data.

An approach which allows for a programming model familiar to the programmer is still lacking; most of the research efforts either offer a new programming language or a model that attempts to integrate kernel code with the host code. A high abstraction programming model must offer these characteristics, distancing itself from OpenCL's model, but maintaining the efficiency of hand written OpenCL code. The ideal model

[4] Single instruction Multiple Data

must also assure the programmer that the code written will be executed in an accelerated context, and as such, should be a structured approach.

The challenges presented with doing this is to find a common ground between both a common CPU programming model, like C++, and OpenCL, and integrate the two seamlessly as possible. The next step would be to further orchestrate the execution of the accelerated code in order to achieve a near optimal execution graph, taking advantages of the devices present.

Marrow programming model for GPGPUs[5] that is not as abstract as other approaches, allowing for the scheduling of skeleton execution and the integration of these skeletons, written in OpenCL, into a host software. The framework addressed the orchestration, presenting an API for submitting skeletons. As will be discussed, Marrow's accelerated code is strictly OpenCL code, and with an enhanced programming model, that bridges the gap between host and OpenCL, one can facilitate the deployment of computation to any accelerator supported by Marrow.

1.3 Work proposal

The premise of the present work is that efficient computation with recourse to GPGPU devices must be abstracted from the back-end framework and must instead be integrated with the programming language. The idea is that one can have container instances that can be converted and uploaded to GPU devices in order to accelerate the execution.

Marrow can accommodate for this by allowing for a computation over containers in device context; given an enhanced model with automatic code translation, from host to device, and an optimization engine. The programming model would be built upon the usage of ADTs[6] that can be themselves converted to OpenCL skeletons in compile time. The transformations from the ADT representation in C++ context, to skeleton format in Marrow's programming model will be handled by C++ expression template mechanism, through which all the parameters of a given skeleton can be specified.

The proposal is to add a layer to **Marrow** which will allow further abstraction from **OpenCL**. This layer will allow automatic generation of host and device code based on a specification of the computation to be made. As the computation will be applied to a container, the basis will be the **Marrow**'s *map* construct

1.4 Contributions

After delivery, it is conceived that Marrow will be endowed with the ability to accelerate many algorithms expressed with the new programming model; automatic OpenCL kernels

[5]GPGPU - General Purpose Graphics Processing Unit, a term for using graphical processing units as general purpose computational devices.

[6]Abstract Data Type – An abstract data type is a type with associated operations, but whose representation is hidden.

will be generated from the usage of the supported ADT's and primitives, and automatically offload to the accelerator.

This will allow for a more intuitive programming of the GPGPU device, and will be a more intuitive and expressive way to harness the power of the GPGPU device while maintaining the already known programming model of C++.

1.5 Document structure

This document is organized in six chapters, each with corresponding sections and, where applicable, subsections. Chapter 1 introduces the Motivation, Problem, Work proposal and Contributions, followed by this section with the document structure description. Chapter 2 introduces the State of the Art, listing relevant endeavors that relate to the work proposed, with some being skeleton based and others of a linguistic nature. Chapter 3 refers to the work done, being descriptive of the strategy chosen and how it was implemented. Chapter 4 deals with *OpenCL* kernel fusion, and when it applies. Chapter 5 deals with the evaluation done to the present work, and lastly chapter 6 is the conclusion.

CHAPTER 2

STATE OF THE ART

2.1 Introduction

This chapter aims to introduce into discussion the state of the art on High-level Programming of Many-Core Architectures. The main research efforts that are in direct correlation with the work proposed will be referenced and discussed, providing an overview into the field. The discussion is directed by approach, grouping related efforts together and presenting the approach itself in large strokes. First there is a presentation of Algorithmic Skeletons, then those approaches that directly build from that semantic model. Latter linguistic-based research efforts will be presented that also relate to the the current proposal.

2.2 Algorithmic skeleton frameworks

This section introduces the notion of skeleton, following with the frameworks that support both data- and even task-parallel skeletons.

2.2.1 What are algorithmic skeletons

Algorithmic skeletons, as defined by Cole[22], are a solution to the difficulties in parallel systems. The main problems associated with the traditional approaches are *problem decomposition*, *distribution*, *code* and *data sharing*, *communication* and *synchronization*. The most fundamental problem, *problem decomposition*, is associated with the identification of parallelism, or in other words, how a problem may be split into smaller, more tractable problems, that can be individually computed and aggregated in order to solve the original one.

The second most significant problem, *distribution* is concerned with the computation itself: how to distribute evenly the sub-problems onto computational units in such a way that those are computed concurrently. Cole[22] concluded that it would be improbable that *decomposition* and *distribution* of a given problem would lead to a situation in which a large number of processes would compute mutually independent sub-problems over equally independent data. Most problems would, therefore, be solved by applying similar or identical tasks to data that would be somewhat shared between the former.

This premise is followed by the recognition that even when a given problem is broken down into smaller sub-problems and, all of these are distributed over a given set of computational devices, sharing data between processes is still a problem to be reckon with. Existing parallel paradigms, until the formalization of algorithmic skeletons, lacked the universal characteristic necessary for heterogeneous programming, as Cole[22] stated "[...] the notion that the model of computation and its associated programming constructs should be “universal” in the sense that they may be employed unrestrictedly in solving any problem", the programming framework must be independent of device specific implementation and be as close to familiar programming paradigms as possible.

Most of the study in designing correct and efficient algorithms has been focused on sequential programming. This fact allows one to reason that, while other paradigms do exist, the framework should capitalize on the familiarity of programmers with the sequential programming paradigm, without too many specialized programming constructs . On the other hand, it is required that the resulting framework has the necessary abstraction mechanisms to allow the programmer to express her algorithms in a coherent, machine independent way.

Algorithmic skeletons, one of the strategies found, are, in a sense, high order functions that apply some other function to a collection of data. This allows skeletons to not be concerned with the lowest level details of the problem domain, but instead, to acquire higher computational structure of whole classes of algorithms. A skeleton-like solution to a given problem would, therefore, be a matter of supplying a correct function that is appropriate to the task in hand. This would allow a high versatility of the framework, in the sense that, one could retain the overall high-level abstraction structure between solutions, where applicable, and simply customize the solution itself with appropriate functions.

Let us explore what came to be a commonly implemented skeleton, *map*.

Let f be defined as a parametric function from a to b,

$f : a \rightarrow b$

then let *map* be defined as a high-order function that applies f to every element of a given collection $[a]$,

$(mapf) : [a] \rightarrow [b]$

then, map can be defined in function notation as follows,

$map : (a \rightarrow b) \rightarrow ([a] \rightarrow [b])$

The advantages identified in this paradigm are immediate: one can template any high abstraction structure like *map* while keeping *f* as a placeholder for a sequential, structured, high abstraction function that will solve the sub problem.

Universality, which was identified previously as a major concern, can be achieved by having the first class citizens of the framework (i.e, *map* and meta-functions) as a device specific, high performance, implementation. Then a problem's solution may be a template specification that takes the place of *f* and is *ad-hoc* implemented in a high abstraction language suitable to the specification of the algorithm and to the framework itself.

It comes that, in the programmers' perspective each skeleton is a higher order function, or a template. The systems' implementer, on the other hand, sees a skeleton as a computational pattern for which an equivalent, efficient parallel "harnesses" must be defined[22]. A drawback of this approach, however, is that each skeleton will require a different implementation for each different parallel machine. The gains are that the *distribution* is handled by the structure of each skeleton, and the mere selection of a skeleton (or template) will signify a commitment from the programmer to the *distribution* strategy employed.

In the case of the work presented in this proposal, **Marrow** framework serves as the underlying skeleton framework, and both *problem decomposition* and *distribution* are continuously being developed.

2.2.2 Skeleton frameworks for heterogeneous computing

Many research efforts have been in the development of skeleton based parallel programming. The approaches differ from skeleton deployment strategy onto skeleton specification methods. In all the following frameworks the large spectrum programming methodology is similar; the programmer is required to provide functions, in one form or another, that when called upon by skeleton implementations, operate over input data to provide a sub problem solution. The programming models in these frameworks, however, have little in common besides the existence of skeletons to work with, and their semantics.

Whilst exploring the programming models, one encounters very disparate concepts, and an overview of notable work is ensued. There are four remarkable research efforts that warrant one's attention, **SkeTo**[4, 6, 7, 23], **SkePU**[9, 10, 11, 24, 25, 26], **SkelCL**[8] and **Marrow**[12, 13, 14], and two major initiatives from AMD and NVIDIA, respectively **Bolt** and **Thrust**.

2.2.3 SkeTo

In **SkeTo**, the programming model closely follows Constructive Algorithms' theory[22], bringing the idea of fusing the successive application of functions. Constructive Algorithms views algorithms as composable units of logic. This opened the path to the fusion mechanism that is the core of **SkeTo**, which is, in essence, a strategy to execute a sequence of skeletons; the idea is simple: given a skeleton application, the fusion process brings about a single, composed, application. **SkeTo** authors achieved this by creating a sort of

```
1 double var = reduce(plus, map(square, map(sub(ave), x)));
```

an application descriptor, a structure that holds references to the functions called by the skeletons and, groups those functions by skeleton, creating a list of successive callbacks. The framework's main data transfer vehicle is a STL vector-like collection that deals with data transfer between host and device memories. The reason why this is important rests on a simple analysis of a simple skeleton statement.

Reproduced from [27]

In the implied semantic skeleton model the aforementioned logic encompasses three local loops, since both *reduce* and *map* require access to all of elements of x. **SkeTo**'s fusion mechanism allowed for the compilation of the implied execution graph of the three loops into a single loop, with one global communication between host and device.

2.2.4 SkePU

SkePU authors took another approach, making heavy usage of the macro mechanism present on C/C++ languages. A function, in **SkePU**, is defined with recourse of these macros that allow one to associate a name with the definition, and a definite number of arguments (which also determines the macro used in the process). The macro used will result in the expansion of the pre- and post-amble required to write a skeleton with the function implementation. This meant for a less cumbersome integration process, with fewer steps than those required by **SkeTo**'s, but did not abstract from OpenCL's syntax. **SkePU**'s functions may have arbitrary arity using the *map* skeleton, and this directly relates to the elements of a dataset that can be referenced directly by the function itself.

2.2.5 SkelCL

SkelCL is another implementation of Cole's[22] skeleton concept. In this framework, the specification of the skeleton is written in a string-like fashion onto the host's source, and is translated into skeleton form whenever the host decides to execute the skeleton. The actual implementation is not based on macros, as in **SkePU**, or with expression templates like **SkeTo** but, instead, in **SkelCL** the authors provide a transparent way for the programmer to write the kernel code and have it adapted to the back-end in use, at an appropriate time.

In **SkelCL** the arguments to the skeletons are in array form. The *distribution* of workload between GPU units takes place issuing disjoint data sets, part of the original input vector. The configuration of the partition of data is configured by the distribution itself, tieing the concept of work-load *distribution* to that of input *partioning*. To achieve ease of use, **SkelCL** offers three notable distribution strategies which are *single*, *block* and *copy*.

The *single* strategy maintains the input vector in a single GPU and, there is not partitioning of data. *block*, on the other hand, splits the data into contiguous and mutually

disjoint data sets, formed from the original vector. With *block*, each GPU stores a contiguous, disjoint part of the vector. The *copy* strategy copies vector's entire data to each device, and with single a vector's is assigned to only one GPU.

2.2.6 Marrow

Marrow allows the programmer to specify the skeleton on a side file, that is later compiled by an OpenCL backend. Sequential skeletons over the same data are pipelined in similar fashion to **SkeTo** fusion algorithm, although the former uses a derivative of abstract syntax trees to structure execution while the latter uses a control structure; however the main advantage of Marrow's strategy is that one can combine the application of different skeletons while **SkeTo** is limited to a *zip* skeleton with only two functions. **Marrow** is concerned, still, with efficiency. Like **SkeTo**, with fusion mechanism, or **SkePU** with its Vector implementation, **Marrow** handles data transferring from the CPU context to the GPGPU device by applying a taxonomy to data; data can either be intermediary or final. In **Marrow** only final data is uploaded to the device.

The programmer is allowed the same degree of control over the addressable data unit in skeletons, because the skeletons are written in pure **OpenCL**. The programmer can and must specify in **OpenCL** code, not only the function called by the skeletons, but also the granularity of the access associated with the input arrays. One can address a 1024 bytes array, with long-, int-, or byte-scheme. The data partitioning happens at the host and device level, being that the programmer has the responsibility to write that partitioning herself, both in the host source and in the kernels code.

The data, on the host code, is represented through a `vector` specialized construct. **Marrow**'s `vector` is a continuous memory region that holds the elements. It also has the concept of geometry and size embedded within. A `vector`'s geometry has 3 dimensions: x, y and z. This allows **Marrow** to decide the `clEnqueueNDRangeKernel` parameters, threads and calls to be made. For instance, both `vector`s with `<1000,1,1>` and `<500,2,1>` are executed in one dispatch to **OpenCL** but a `vector<500,2,1>` has a two dimensional grid while the `vector<1000,1,1>` grid is uni-dimensional.

Marrow has four basic constructs: *Pipeline*, *Map*, and *MapReduce* and *Loop*.

`Pipeline` allows the programmer to run several kernels in a sequence over the same data. This results in lowering the data transfers between the host and device environment. The `pipeline` construct can be seen as a function composition construct. All kernels that are pipeline are applied in sequence over the result of previous kernels.

`Map` is a construct that allows the application of a kernel over a vector. Each thread on the device will run the kernel and execute its instructions on the context of an element of the vector.

`MapReduce` construct follows the **MapReduce** programming model in the sense that it allows a kernel to be applied to the data, in the `map` phase, and the data to be aggregated on the `reduce` phase.

`Loop` construct allows for a loop notion in the sense that it allows a kernel to be applied to the data repeatedly.

2.2.7 Bolt

There are other interesting approaches that follow these methodologies like AMD's **Bolt** and NVIDIA's **Thrust**. **Bolt** is an library C++ template library optimized for GPUs[1]. AMD's focus, in developing **Bolt**, was to provide a high performance container library, following to C++'s STL[2] conventions, that would contain high performance implementations of algorithms like *scan*, *reduce*, *transform* and *sort*, to use with skeleton programming. This library was developed as precursor to HSA[3], an architecture that will allow a complete transparent memory model. Bolt has the ability to access host memory through direct interface with the memory itself. **Bolt** also allows the programmer to allocate and manage device-local memory and, as such, **Bolt** API's accept either host memory or device vector. Given **Bolt**'s architecture and design, usage is simple; assuming the existence of an already created and filled STL vector, sorting it using **Bolt** is simply calling an already defined sort algorithm. When the call happens, the **Bolt**'s underlying system kicks in, and the call is converted to a task submission. The runtime selects the appropriate device to run the sort potentially running it on the CPU, if the GPU is unavailable or the vector size does not justify GPU acceleration. Memory related issues, like data transfer between contexts are all taken care of by the runtime itself.

Bolt allows for customized algorithms. The algorithm must be expressed in a functor pattern, on an abstract data type, on OpenCL dialect. **Bolt** itself has a macro, *BOLT_FUNCTOR*, that allows the programmer to define such ADTs[4] in concise, OpenCL string-like, fashion. These defined ADTs are then made available to the OpenCL compiler which is called with *clBuildProgram()*. With the introduction of a new **Bolt**'s primitive, *restrict* any experienced C++ developer can now write algorithms to run on a plethora of supported devices by using side-effect free algorithms.

2.2.8 Thrust

Thrust is NVIDIA's approach to heterogeneous computing and is built on top of NVIDIA's proprietary CUDA technology. NVIDIA states that, by using **Thrust**, one can achieve 5 to 100 times the performance seen with STL (C++ Standard Template Library) and TBB (Intel's Thread Building Blocks). While this comparison is hardly fair, since STL's developers focused on a more general, easy-to-use, library for C++ and Intel focused on an user friendly, almost domain specific language, way to express parallel computations, NVIDIA's **Thrust** focused on approximating the CUDA's already established technology to a more unified, user friendly approach to better productivity. **Thrust**, like **Bolt**, allows

[1] Information from _hsa-libraries/bolt_2014 retrieved at 6/2/2015
[2] STL – Standard template library, containing many useful implementations
[3] Heterogeneous System Architecture
[4] Abstract Data Type - an object oriented terminology for data encapsulated on a logic wrapper.

Framework	Skeletons supported
SkelCL	AllPairs, Map, MapOverlap, Reduce, Scan, Zip.
SkePU	map, reduce, mapreduce, maparray, mapoverlap, scan, farm.
SkeTo	map, map_with_index, zip, zipwith, reduce, scan, scanr, shift, postscan, postscanr, gscanl, gscanr, shiftl, shiftr.
Marrow	Map, MapReduce, Loop, Pipeline.
Bolt	Multiple implementations that can be viewed on Bolt's documentation.
Thrust	Multiple implementations that can be viewed on Thrust's documentation.

Table 2.1: Table with skeleton support from the frameworks discussed.

one to configure the device on which to ran the skeletons. Where **SkelCL**[8] used strings with OpenCL definitions, **SkeTo** and **SkePU** used C++ templated expressions and macro defined skeletons, and **Bolt** used *BOLT_FUNCTORS*, **Thrust** uses *algorithms*.

These *algorithm*s are essentially skeletons themselves, and are expressed in C++ functor syntax, overriding the *()* (apply) operator. As back end, **Thrust** supports the host, where the execution originally begins, Intel's Threading Building Block, OpenMP, CUDA and C++ threading model. To maximize customization of the execution model, Thrust provides an parallel execution policy interface, that is implemented for all the already mentioned back-ends.

The default parallel execution policy is *thrust::device* that is associated with all the *algorithms* called without an explicit execution policy. Decision of witch back end to use is based on the tags of the system's iterators; to have an *algorithm* dispatch a given functor on a specific back end, one has to specify the associated execution policy.

The basic memory container is *device_vector*, but some execution policies enjoy performance benefits if specific containers are used, to prevent data copying between host to back end infrastructure. *thrust::ttb::par* is the execution policy of Intel's TTB (with *thrust::tbb::vector* as a specialized container), *thrust::omp::par* for OpenMP (with *thrust::omp::vector*), *thrust::system::cpp::par* for C++, *thrust::seq* for a sequential algorithm, and finally *thrust::cuda* for NVIDIA's CUDA backend. In this last execution policy, to each functor is attributed a *cudaStream* in which the executor can achieve parallelism either with host as with other CUDA kernels. Synchronization among streams can be done with *cudaStreamSynchronize* primitive, that allows data interdependent stream coordination. **Thrust**, while offering the same expressiveness as the other frameworks, offers also a richer set of first order functions, available for consult in the API documentation[5].

[5] Information from https://developer.nvidia.com/Thrust retrieved at 6/2/2015

In table 2.1 there is an a mapping from the frameworks described earlier, to the supported skeletons. To complement the information in the table, a description of the referred skeletons is presented, grouped by meaning.

allpair - SkelCL Applies an user defined calculation over pairs of rows and column vectors.

farm - SkePU Independent task with dynamic load balancing.

gscanl, gscanr - SkeTo Accumulates the values of a list with the function from left to right.

map - SkelCL, SkePU[6] **SkeTo, Marrow** Applies an unary function to each element on a vector.

maparray - SkePU A map with some replicated operand arrays.

mapreduce - SkePU, Marrow Applies a mapping operation and a reduce operation in sequence.

mapoverlap - SkelCL, SkePU Applies an unary function to each element, with the ability to access more than just a single element of the container.

map_with_index - SkeTo A map with the addition of a 0-based index.

loop - Marrow A skeleton for repeated application of a function, given that the condition that controls the application of the function is on the host.

pipeline - Marrow Allows for ordering the application of other skeletons.

postscanr - SkeTo Computes accumulation of the values form right to left.

scan - SkelCL, SkePU, SkeTo, Bolt, Thrust Applies a prefix sum calculation.

scanr - SkeTo Applies an associative binary operator that computes a prefix-sum.

shiftl, shiftr - SkeTo Shifts the elements of a list from left to right (right to left respectively).

reduce Applies a binary function that performs an accumulation.

zip - SkelCL, SkeTo Applies an unary function that operates on a tuple formed with elements from two sets.

zipwith - SkeTo This skeleton takes two lists of the same size and applies an argument function f to each pair, formed with elements from each list with the index.

2.3 Embedded linguistic support

Embedded linguist support implies that the programming model includes primitives that allow one to express algorithms in a way that allows for heterogeneous computation. This is achieved by the added primitives – or in some cases, a whole programming language – designed with multi-device computing in mind; the compiler or runtime can, as will be discussed in future subsections, compile the source to GPU instructions and, in some cases, the runtime can even decide on the optimal computational device on previous execution profiles.

2.3.1 Dandelion

Dandelion[19] is a project to provide a framework that will facilitate the development of scalable heterogeneous systems, where logic can be ran on multiple different architectures, with high level abstraction from the intricacies of the different contexts. A context, by definition, is the environment of execution that encompasses the processing unit – where the code runs – the memory schema associated with the unit, and the whole hardware and software stack that allows complex operations to be performed.

To achieve the objective, development was geared towards a dataflow design: every cluster element is controlled by a dataflow engine designed specifically to control the given element. The system is, therefore, composed by several execution engines: a distributed cluster engine, a multi-core CPU engine and a GPU engine. As documented, each engine represents its computation as a dataflow graph and the edges between engines are asynchronous communication channels; data transfers among engines are managed by Dandelion and are transparent to the programmer.

In Dandelion[19] the authors identified the urgency of merging the high abstraction capabilities of modern programming languages with the finer grain programming details that are involved in programming for different contexts. The premise put forth in *Dandelion* was that a "single machine" abstraction, where the programmer writes sequential code in a high-level programming language like C# or F#, and the system automatically executes the logic utilizing all the available parallel computational resources in the execution environment. The main difficulties identified were the need to encapsulate the system components, rendering them transparent to the programmer, and integrate multiple runtimes – given the different contexts – efficiently to enable high performance for the overall system.

Dandelion enjoyed success in integrating a high-level programming language with the already mentioned accelerators. The programming model chosen was LINQ (Language-Integrated Query) [7], which is a mechanism that endows .NET infrastructures with the capabilities to query and update data with potentially any kind of data store. LINQ inter-operates seamlessly with .NET Framework collections, SQL Server databases, XML

[7]Information from https://msdn.microsoft.com/en-us/library/bb397926.aspx retrieved at 6/2/2015

Documents and other available data stores. Three primitives were added to LINQ, by Dandelion: *.AsDandelion(gpuType)*, *.DoWhile(body,cond)*, *.Apply(f)*.

The first primitive, *.AsDandelion(gpuType)*, is applicable to LINQ collections. When the method is called upon a subclass of *Collection*, the compiler attempts to automatically detect the collection size in order to reserve space on different execution contexts; in GPU, for instance, allocation mechanisms are limited due to the particularities of the memory layered schema. Even though allocation is supported in some cases, it is not widely supported by all the device manufacturers, and therefore is not standard. This is seen in NVIDIA's CUDA and Khronos's OpenCL specification. According with Khronos's [28], the kernel cannot perform any dynamic allocation of memory. CUDA developers took a different approach, creating *cudaMalloc** primitives, that allow for in-device specific memory allocation. FPGA's, on the other hand, being dynamic reconfigurable logic chips do not have any ABI [8] and as such do not provide basic memory management infrastructures.

The parameter provided to *.AsDandelion* specifies the collection type to use when the code is ran on GPU. The specification of the collection size is optional, with Dandelion's compiler infrastructure capable of inferring the native LINQ collection size automatically, however the subsystem performance is improved when advance knowledge of the size is available.

2.3.2 Lime

Liquid Metal Programming Language by IBM [17] is general purpose language whose main goal was to provide a high abstraction programming model. While similar to Dandelion in respect to their objectives, and approach, IBM chose to develop a linguistic approach in such a way as to allow large portions of a program to be realized in hardware via direct synthesis into a programmable logic fabric like FPGA.

This is a goal that directs the project to a truly co-computational model, where one can offload some algorithm onto a system's FPGA or GPU devices, while other parts can run on the CPU. While FPGA programming is traditionally a field of domain-specific languages with their own constructs, syntaxes, and meta-modeling like Verilog or VHDL, this project shorten the gap between the dramatically different programming models of CPU and FPGAs, allowing one to develop truly reconfigurable systems.

While **Dandelion**, which built upon .NET and CCI[9], **Lime** is itself composed of a language, a compiler and a runtime that work together to offer the ability to compile a single language to so many different architectures. Taking from Java's machine independent semantics, and dynamic execution model, which means adaptive recompilation as needed, **Lime**'s authors seek to achieve a programming model that allowed one to increase the domain of devices on which parallel code could be concurrently ran. One

[8] Application Binary Interface with which a software may communicate with an underlying system.
[9] Common Compiler Infrastructure – a project to ease the development of compilers.

of Lime's authors main concern was the expressiveness of the language. One had to be able to express a parallel algorithm in multiple ways, with the runtime taking to itself responsibility of dividing the programs dynamically as the system's elements increase or change. This, as we have seen already on Dandelion, and as we will see on the next linguistic approaches, strips the control of the locality of execution from the programmer, transferring this responsibility to the compiler. This, of course, means a semantic model that accounts for transparent data transfers within the system elements, each with specific context requirements.

The programming model must address a common denominator in these contexts: static code. While CPU architecture allows for static and not static code, FPGA and GPU characteristics and, more importantly, the project's goal of moving code in chip dictate some requirements that the languages had to adhere to. This implied a functional model since the lack of side-effects, of direct memory addressing, alongside with the ease of compiling such code to hardware(given functional programming model's proprieties) are all proprieties that allowed for heterogeneous programming.

To achieve a balance between object and imperative oriented paradigms, Lime includes micro and macro functional features. The micro functional features center about recursively immutable value types. Value types are, essentially, immutable; a type object can only be composed of other equally immutable value types. Lime's value classes may be moved freely across memory boundaries, thanks to these proprieties, without requiring remote accesses or code relocation calculations. Value types are define with the keyword *value* which indicates the compiler that the following class definition is, in fact, a value type and must adhere to the strict immutable principle. This implies that all fields are implicitly Java-like *final*. Still on the type system subject, Lime supports two different arrays: value arrays that extend value class proprieties to arrays rendering them immutable and composed of value types of arrays of value types.

The other array type supported is mutable arrays, with common Java semantics. While this, in itself, offers little to the already defined Java type system, there is another taxonomy on Lime's array exists: bounded and unbounded. A unbounded array is declared with double squared brackets (e.g. *int[[]]*), and is a construct that allows for an expandable container. Bounded arrays are declared with the more familiar syntax, like Java. These are a subset of unbounded arrays and are defined with either a positive integer or an *Ordinal*, another of Lime's value type that corresponds to the positive integer set. Bounded arrays have fixed, known size, an this is another propriety that facilitates the data transfer between contexts, be them either FPGA or GPU. The same propriety was required of Dandelion, that attempted to calculate the *Collection* size on runtime. To be a general use language, Lime had to offer modern mechanisms for polymorphic code while maintaining its goal for heterogeneous computation.

While in Dandelion, for instance, one defined the algorithm to be ran on the accelerator on a functor-like manner, with side-effect free semantics, transferring the responsibility of achieving efficient compilation to the back-end compiler, Lime's authors took upon

themselves the responsibility to offer a programming model that transparently translated code to the accelerators; this meant allowing templated programming in Lime meant a trade-off. While in CPU context, and more specifically in Java, generics are supported by a process of parametrization that generates one copy of compiled code, meaning that for a given parameter combination one loses on compiler optimization while gaining of memory usage, the requirement of fixed sized value-types in hardware made this approach nonviable. Generics for FPGA meant generating templated code, which in turn meant generating copied code for each parameter combination, effectively compiling code that could further suffer type specific optimization but that ultimately hurt the space usage. Lime, therefore, took a hybrid approach using parametrization as a default and switching to templated code depending on the device on where to ran.

Lime's programming model adapts very nicely to parallel programming by exposing a useful set of operators which allows task, data and pipeline parallelism. Parallelism is based on data-flow graphs which consists in nodes, that represent *task*s, edges that represent *stream*s and *port*s. The end of the graph is expressed with a *finish* node which also initiates the submission of the graph to the underlying compiler that analyzes and chooses the best concurrent execution algorithm distribution available, based on the graph and the system element matrix available.

2.3.2.1 *task* operator

The *task* primitive is used to create tasks. A task is a high level function that applies a method as long as data is presented to the input *port*. The output produced by the method is queued to an output stream. The method can be either a static or an instanced method, allowing the algorithm to be written either on a purely functional or in an stateful manner. The operator *task* binds the method with the worker itself, being that when one gives an instance, its' methods become the workers' methods by applying the task operator. The execution consists on the iterative application of the worker method as long as data is available or until *Underflow Exception* is thrown, and is started with a call to the task's *.start* method. Synchronization with a given task can be achieved with the *rendezvous()* method which is a blocking call. Taxonomy on Lime tasks is divided in two main categories: isolated and non-isolated. Isolated tasks map themselves to non-final static fields in Java, that have their own address space; an isolated method can mutate all the fields accessible through the *this* reference. Only these isolated methods may be used to create tasks, while attempting to initiate a task from a not isolated method will result in compile-time error. As already discussed, value types are immutable and as such,any of a value type's isolated method can be used in task creation. There are two special kind of tasks in Lime's model, that go by the name of *source* and *sink*; while these concepts will not be further developed, its important to state that, as tasks, these need not be isolated, and represent special cases of tasks that either perform I/O operations or side effect operations.

***split* and *join* tasks** In addition to the extendable *task* concept, Lime offers system tasks which are ready to use, and are commonly needed, like *split* and *join*. The *split* task allows for a transformation over a *tuple stream*, composed of a sequence of *tuples*, into another *tuple stream* where the latter's *tuples* are composed solely on the members of the *tuples* received on the input. The *join* task does the inverse operation, receiving as input *tuple* members and emits as output *tuples* that are formed by those inputs.

2.3.2.2 => operator

The => operator serves as a way to concatenate execution of tasks over a given data. Given two tasks, each has its own *port*, which is associated with the task parameters. The input *stream* works alongside the *port*, effectively filtering only the data for input that is of interest for a task.

2.3.2.3 @ operator

Collective operations are expressed with @ operator, which can be applied to operators, instance methods, and static methods. Lime syntax dictates that the operator must precede other operators, may them be either infix- or post-fix notation. This operator can be seen as a *map* operation, since it applies a operator, instance method or static method to every element of a collection. Lime compiler understands when, given a combination of @ and a binary operator, like + the expression on the right side can be either a valid instance of the expression's immutable value type or a collection of the same type. Double application of the @ operator results on a *reduction* operation over an array. All of Lime's *task*s can be offloaded to FPGA or GPU devices, through the OpenCL's back end, except the very first or the very last tasks on an execution graph. As already seen across other approaches, one of the requirements for a successful offload to such a backend is necessary, as already discussed, that the *task* is isolated or, in other words, does not use operations with side effects.

2.3.3 StreamIt

StreamIt is a language for streaming applications from Massachusetts Institute of Technology and its goal is to provide novel high-level representations that allow the programmer to improve on its own productivity. The programming model focus on the stream abstraction; there is an increasingly movement towards device or applications whose performance depends on dealing with an unending sequence of similar data.

This is a different approach to a linguistic from that taken in Dandelion or Lime. While Dandelion extended an existing language with primitives and a side-effect free functor-like semantics to LINQ, and Lime extended Java's Virtual Machine ecosystem with a new functional language, StreamIt's authors defined a whole new programming paradigm, with a new language and three basic concepts with which any stream computation can be expressed.

The most fundamental aspects of a streaming application is that its data-set is a virtually infinite sequence of data items. As such, a streaming application can be regarded as a composition of transformations over one or more streams. The fundamental computation unit in this programming model is a *Filter*.

A *Filter* is quintessentially a transformation over a *stream*, that operates on one, or more, data items running on a *stream*. The bulk of operation in a filter is written on the *work* method, while initialization procedures occur during the *init* call.

There are three basic constructs offered by StreamIt, with which one can express any stream computation: *Pipeline*, *SplitJoin*, and *FeedbackLoop*. These three constructs allow for the application of *Filter*s, and any StreamIt application is a *stream graph* whose nodes and edges represent instances of one of these types.

Each graph is a composition of these nodes; *Pipeline* nodes symbolize that a finite sequence of nodes will follow, or in other words, that a stream will suffer a finite set of transformations. The finite set of transformations, that the *Pipeline* symbolizes, can be a composition of any of the three constructs, where *SplitJoin* allows one to specify parallel independent streams that stem from the application of two different filters to the same input stream. The splitting and merging takes place with a *splitter* and a *joiner* that are defined in *.init*, alongside the *Filter* sub-classes to be used. The last construct, *FeedbackLoop*, allows for the composition of loops. Each instance of *FeedbackLoop* contains a body stream, a loop stream, a *splitter* and a *joiner*. All these components work together in allowing the same filters to be applied in a feedback path. These constructs allow for a viable machine language for a grid-based architecture, since StreamIt abstracts from granularity and memory layout, focusing primarily on independent processors.

On compile time, StreamIt compiler, creates a number of profiles from the *Filter*s that make the program; these profiles are used in multiple stages, to check if a given *Filter* is better suited for the GPU profiles available. Once a near to optimal configuration of *Filter* per computation device is achieved, stream flow is analyzed in order to improve the execution schedule. The StreamIt compiler tries to improve performance by, using GPU specific characteristics, combine memory transfers between GPU and host.

2.3.4 OpenACC

OpenACC is a specification for an unified manner to support accelerators. It is yet another linguistic-type approach. OpenACC adds three basic concepts to the C programming language but, differently from Dandelion that also added three primitives, using accelerators with OpenACC does not require usage of side-effect free semantics explicitly. This approach is solely based on the power of compiler hints; in OpenACC, the programmer can harness the accumulated technology advancements in both compiler technology and GPU backends by providing hints to the compiler. Using *#pragma*s like *#pragma acc kernels* indicates to the compiler that a given loop of the following code could be parallelized in other contexts. The code that can be parallelized is primarily loops. Given that the

programming model, while using OpenACC, remains the same, code written under this technology can be easily exported from NVIDIA GPU, to ARM CPU, to Intel Xeon Phi, without the need to meddle with OpenACC code. The abstraction from CPU or accelerator specific code, the usage of the directive #*pragma acc kernel*, #*pragma acc data*, and #*pragma acc parallel* requires only a short shift from the traditional memory addressing schema (i.e. translating a AoA[10] to a SoA[11] to take into account memory accessing and writing) to achieve a considerable speedup faced to non-accelerated implementations. There is an additional advantage, in contrast with other approaches, which is that a compiler which is not OpenACC aware will safely ignore the hints, and compile the valid code.

2.3.4.1 *acc kernels*

The #*pragma* macro, as already discussed, is a compiler hint. On OpenACC #*pragma acc kernels* is a directive that indicates to the compiler that the following loop should be turn into a function in a kernel, and accelerate it on a GPU. This directive gives freedom to the compiler to decide on the best way it should accelerate the code.

2.3.4.2 *acc parallel*

The #*pragma acc parallel* serves as a way to indicate to the compiler that some parallelization of the following loop is required; this construct means the creation of a number of parallel *gang*s that immediately begin to execute the body of the construct redundantly; when a *gang* reaches a work-sharing loop, that *gang* will execute a subset of the loop's interactions depending on the scheduling policy. Synchronization mechanisms, like **barriers** do not exist at the end of these work-sharing loops. In a work-sharing model, like *parallel* constructs, there's no parallelization required.

The OpenACC *gang*s execute code redundantly until they reach a work-sharing construct, then split the work of that construct across the *gang*s. This means that one needs to include a work-sharing loop in the *parallel* construct, or each gang will execute all the code within the construct redundantly. Gang-level parallelism is exploited with work-sharing, but vector parallelism is not. Vector parallelism can be exploited in any of four different ways. The programmer can add a *loop vector* directive just before a loop, telling the compiler to generate vector code for that loop or, the compiler can use classical vectorization analysis to automatically identify loops that can be compiled in vector mode. Failing that, if there is only a single parallel loop, the work-sharing loop, the compiler can split the iterations across the *gangs*, then generate vector code for the each iteration executed by each gang. The code's generation and optimization for a parallel construct is the same as for the kernels construct. A key difference is that unlike a kernels construct, the entire parallel construct becomes a single target parallel operation; the parallel construct becomes

[10] AoA stands for Array of Arrays.
[11] SoA stands for Structure of Arrays.

a single kernel.

2.3.4.3 *acc data*

The compiler has the responsibility to identify the code that is parallelizable, and act accordingly. While the compiler can detect parallelizable code, nonetheless, it is the memory usage and data transfers who remain the central players in the gain on accelerating the code. This pragma indicates that the compiler should act over the data identified previous to execute the code. Without this *pragma*, every execution will issue a data-copying operation, effectively moving any used array or variables in host's memory to the accelerator. With this *pragma*, (e.g. *#pragma acc data copy(A)*, where A stands for an integer array used within the to-be parallelized code) the data is copied to the device once, the execution is parallelized in many threads and only finally is the code copied back to the host's memory.

2.4 Polyhedral model

There are other approaches that relate to the parallelization of code, by means of a polyhedral compiler; this technique allows the compiler to detect code that can be parallelized by defining the execution of loops as a polyhedron, where the plane is defined by affine inequalities and execution is an integer set corresponding to the values that the variables may assume. This allows for the compiler to figure out data dependencies between iterations and parallelize the loop itself by unwinding these dependencies [29, 30, 31].

2.5 Expression templates

Expression templates are a mechanism of parametric code offered by C++, where the programmer writes expressions based on typenames. These expressions behave differently, accordingly with the types of arguments used. For any given expression template, C++ compilers create a different implementations for each combination of the parameters used, and this is useful in describing objects that should change internal behavior when composed of different objects. In the literature, there is some success in using expression templates for OpenCL code generation[32, 33, 34].

GPGPU code generation from expression template was studied[32] in order to minimize the complexity associated with Cuda or OpenCL devices. In the same paper the authors concluded that C++ operator overloading could be used to create a domain specific language to abstract from the intricacies of the backend used. The way that expression templates can be used, follow very closely the notion of a AST[12]. Every expression template can be a node in a AST, and may specify the dataset elements' type, a operation

[12] Abstract Syntax Tree – a tree that represents a computation. Each node represents an element of an expression.

that represents the computation that will be executed on the elements and other auxiliary information.

This compact way of representing the AST node can be converted into a OpenCL kernel, if one considers that the operation factor can be fused together into one single operation. This is already documented in [32], where it is used for loop fusion or increasing the accuracy in product expressions.

To generate OpenCL kernels from expression templates, one needs to solve three problems identified in the literature: data transfer between host and device, unique mapping between the data and kernel parameters and the generation of the kernel itself[32]. In the present work, only the last two problems need solution, since **Marrow** already handles data transfers between host and device. The AST generated from the expression template usage, can be compiled into an **Marrow**'s skeleton computation tree, and it is in this process that transformations can take place. The trigger for data transfers, for example, could potentially be the overloading of the assignment operator in the ADT's, which will correspond to a synchronization barrier in **Marrow**'s SCT[13], causing the processed data to be recovered from the GPGPU's context.

Kernels' headers can be automatically generated from observing the parameters used on the expression templates, filling in the data directly from the ADT's used in C++ code itself.

In **SkeTo**, a templated functor is used to define the second class function that will be applied to the data; given that **SkeTo** has only data parallel skeletons, this allows the framework to translate the body of the functor to OpenCL code, with both the output and input types being parametrized.

VexCL[35] is another approach, whose syntax is nearest to what is envisioned for **Marrow**. In **VexCL**, operators are overloaded in order to automatically convert standart expressions into kernels. Assuming a correct initialization of **VexCL** context, a valid kernel can be compiled from an expression that consists on vectors. Host code execution results in kernel compilation, from these expressions, which are executed when the assignment operator is used. The result is transfered back to the host using primitives like *vex::copy(std::vector<T>&, vext::vector<T>&);*.

2.6 Discussion

The presented approaches try to ease heterogeneous programming by providing simple programming models with either **OpenCL** or **CUDA** as a back-end. The programming models that resulted from the approaches discussed have some limitations in the expressiveness offered. Some approaches constraint the programmer to the constructs that the system exposes, for instance **Dandelion** while others as **SkeTo** or **Marrow** allow the programmer to use the system's constructs to develop new behaviors through the creation of **OpenCL** kernels. Some approaches also constraint the statements which are to

[13]SCT - Skeleton computational tree

be accelerated, like **SkeTo** or **Marrow**, maintaining that when all parts of the code are compiled there is a guarantee that the section that the programmer intended to run in the device will actually run in the device. Approaches like **Dandelion** or **SkelCL** lack this guarantee, since the code to be accelerated can contain expressions which cannot be executed in device context (for instance, I/O operations).

Both dimensions are equally important: a programmer expects a system to be flexible so the programmer can develop new behaviors when needed but the programmer also desires the guarantee that the code which is meant to be accelerated is.

As such, there isn't an approach which guarantees the execution of the code, in a flexible manner, that does not require the programmer to deal with the back-end in question. Examples of this are **SkeTo**, **SkelPU** and **Marrow** which allow the programmer the flexibility to develop their own behaviors through **OpenCL** hand-written kernels, thus also guaranteeing the execution of those algorithms in device context, but do very little to abstract from the complexity of **OpenCL** language.

To achieve a more abstract programming model for heterogeneous programming the back-end code generation must be detached from the general programming model of the system at hand, which in **Marrow** case, means that **OpenCL** kernels should be automatically created without programmer intervention.

CHAPTER 3

MARROW EXPRESSIONS

The work presented here aims at offering an abstraction from the back-ends involved in heterogeneous programming. The objective is to allow that operations over vectors (such as **Listing** 3.1) can be automatically offloaded to the device without any intervention from the programmer.

As introduced in **Section** 2.5, expression templates are a polymorphic mechanism for the generation of ASTs in a type-safe manner. In this work, expression templates are used in conjunction with `marrow::vector`, to generate the aforementioned trees and

from this structure generate a compilable **OpenCL** kernel and the Marrow orchestration code required to launch the execution of such kernel in the selected hardware.

The expression template mechanism developed uses **Marrow** as back-end. **Marrow** was chosen as it already handles both kernel orchestration and data transfer between host and device.

3.1 General overview

One of the central problems identified while studying state-of-the-art approaches was the lack of abstraction from the back-end system used, in most approaches. Where skeleton driven approaches are concerned, the programmer is required to know **OpenCL** to achieve new behaviors from the system by actively extending the set of functions or macros exported by the system or by running a hand-written kernel which represents the

Listing 3.1: Expected simplification of a vector addition

```
1 std::vector<int> a(10), b(10), c(10);
2 // Omitted initialization a, b and c vectors
3 c = a + b;
```

Listing 3.2: Implemented simplification

```
marrow::Vector<int> a(10), b(10);
// let an and b elements be defined.
c = a + b;
int result = c[0]; // result will be equal to a[0] + b[0]
```

computation desired.

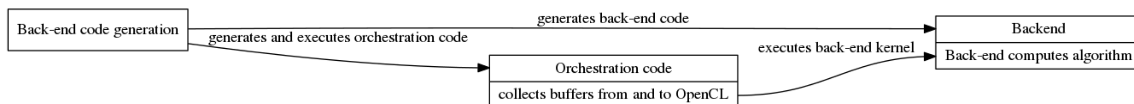

Figure 3.1: The application code generates orchestration code and back-end code. The generated orchestration code will execute the back-end kernel.

The aim of the current research herein documented was to develop a mechanism that abstracted from the back-end.

The focus was to offload embarrassingly parallel computations[1] automatically to **OpenCL**, thus taking full advantage of the system's devices capabilities.

As can be seen in **Listing** 3.1 and **Listing** 3.2 the implemented version remains close to the desired simplification. Accessing container's elements serves as a serialization mechanism. Serialization is guaranteed to occur when any element is required; if the container is yet to be synchronized, the container actively waits for the synchronization to complete before returning the element.

Since **Marrow** already deals with operationality with **OpenCL** calling the primitives when it has to, the focus was on kernel generation and kernel composition.

The code generation has two parts: one related with **Marrow** specific code, and the other related to **OpenCL** kernel generation. **Marrow** generated code will map the kernels to specified containers in order to compute the operation's result.

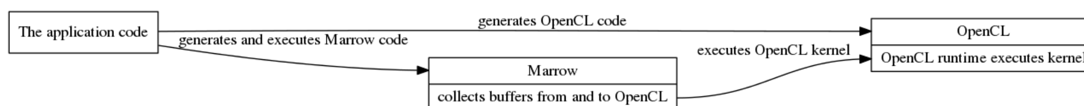

Figure 3.2: The application code, which contains the Marrow Expressions, generates **Marrow** and **OpenCL** code. The generated **Marrow** code will execute the **OpenCL** kernel.

Marrow exports an API that allows a programmer to run a kernel. One has to specify

[1]embarrassingly parallel problems are problems where little or no effort is needed to separate the problem into smaller independent problems

Listing 3.3: Marrow code for launching kernel

```
marrow::Vector a, b, c;
map(KernelWrapper<int*,int*,int*>("marrow_kernel", "evavv.cl", a, b, c));
```

Listing 3.4: Addition example kernel

```
__kernel void marrow_kernel( __global  int * a,
__global  int * b,
__global  int * c) {
        int tid = get_global_id(0);
        a[tid]=b[tid]+c[tid];
}
```

the containers on which to operate on. These serve as input and output buffers for the kernels. The execution of the kernels is also handled through the API, which the programmer must use to launch the kernels when they are needed, reuse the kernels already uploaded or even pipeline the kernel execution. Thus the implementation discussed must offer an abstraction over the API, allowing for API calls (and corresponding arguments) to be generated automatically, without specific programmer commands.

Kernel generation is the main strategy to provide a higher abstraction from the backend. Kernels are generated when the capture of operations over the containers takes place. Each kernel generated corresponds to the **OpenCL** version of the captured operation(s).

Listing 3.3 and **Listing** 3.4 are the generated code of corresponding **Listing** 3.2. When the assignment of one of the variables with the resulting addition of the other two is captured, both the **Marrow** and **OpenCL** codes are generated. **Listing** 3.3 is the **Marrow** code generated. The **Marrow** code is responsible for launching the kernel's execution. **Listing** 3.4 contains the kernel code generated which will be the algorithm ran on device context.

To allow for kernel generation to occur, each captured operation must not be calculated at the point it is called upon. Instead, the normal flow of C++ evaluation is bypassed and an expression template is created.

The newly generated expression template acts as a temporary value, within that operation evaluation, and eventually gives rise to the generation of **Marrow** and **OpenCL** code.

In **Figure** 3.3 we can see what follows the operation capture with a graphical mapping of the operation captured to its corresponding template node.

3.2 Marrow vector

Marrow has a specialized set of containers which have specific functionality in order to be used with **OpenCL**. Since **OpenCL** runtime has no mechanism for dynamic memory

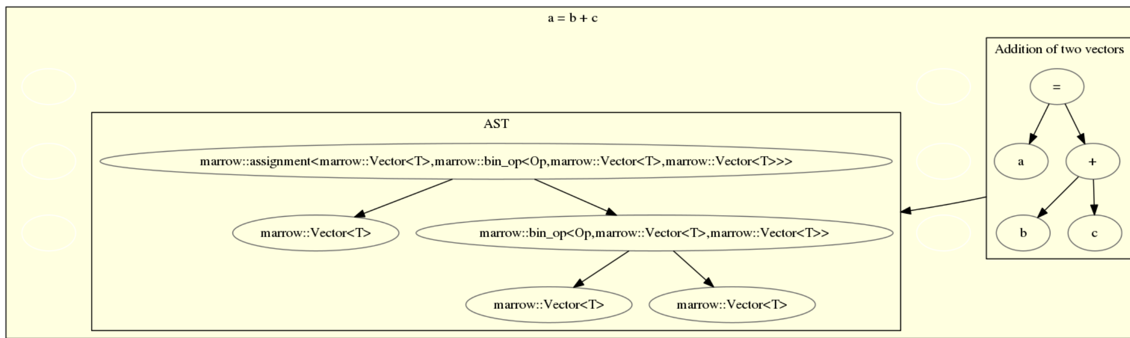

Figure 3.3: Operation capture.

allocation or management, a specialized set of ADT has to be developed in order to allow for transfering data to and from device context.

The `marrow::vector` is one of such containers, allowing memory to be an input and output buffer for **OpenCL**. A `marrow::vector` is parameterized with the data type, how the dynamic proprieties of the vector are managed, and the partitioning unit.

The data type template parameter allows the `vector` to be independent of the data type of the elements it stores. Another template parameter is used to determine how the element count is managed, if it is at all. This allows for the data to be held at a constant sized memory area or to be automatically extended, being copied to a new area when more elements are needed. The `vector` uses operator overloading to offer a simple API for the programmer. `operator[]` retrieves (or sets) the value held at the position specified. `operator=` copies another vector, position by position, to the receiving instance.

In the context of the present work, additional operators were overloaded to implement code generation. `operator+`, `operator-`, `operator*` and `operator/` to support the code generation by first creating template expressions which are also abstract syntax trees that formulate computation at hand.

3.3 Abstract syntax tree

An AST denotes a captured operation and is the base for the code generation process. The first phase of the code generation process is the creation of an AST whose morphology represents the operation captured.

The creation of the AST nodes occurs when an overloaded operator is called upon. Instead of following the normal order of execution of C++, the operations are transformed into expression templates. These expression templates are evaluated only when needed, and they represent the computations to be done.

The **Listing** 3.5 represents the capture point, which occurs at the vector's `operator/`. It is in this context that the tree is formed. Each operator supported is a template; it is ready to receive either another container or an expression template.

Listing 3.5: Example of the division

```
marrow::Vector<int> a(10), b(10), c(10);
// Initialization of vectors
c = a / b;
```

Listing 3.6: Operation implementation.

```
template <op>
    struct operation {
        static const std::string _name;
        static const std::string _text;
    }
```

The tree is composed of nodes from a well defined set of possible nodes. All nodes stem from `marrow_expr`. This is the base implementation and it follows the *curiously recurring template pattern* which means the every node that extends the `marrow_expr` base class inherits all template methods that the `marrow_expr` defines. The methods inherited are the operator overloading, already described, and other methods which wrap calls to **OpenCL** calls which are called upon the template classes instance.

The `vector`, and all other nodes, extends the `marrow_expr`. `marrow_expr` implements the concept of a sized expression: a sized expression concept represents expressions which dealt with element nodes with potentially different sizes. This is the case with `vector` expressions, since different vectors may have different sizes. This concept allows for an expression to always use the lowest size present in the expression.

This is an important feature since Marrow uses this information in order to make OpenCL calls which launch the kernel.

Unary and binary expressions are both supported. The unary expression is implemented through the `un_op<Op,Expr>` template. The first template parameter defines the operation represented by the node. The `un_op`'s second template parameter is the expression to which it is applied to.

The available operations are defined within the enum class `op` in which all operations belong to. To detail an operation, another parametric structure is used: `operation` which receives as a parameter an `op` value. This structure takes an important part in the system since it holds the operation's main information: the operation name and text representation of the operation (Listing 3.6).

The available unary operation at the time of the writing is the unary minus. When applied to an expression the unary minus will be semantically equivalent to the negation of each of the container's elements.

The `operation` name is used in the tree generation, while the `operation`'s text component is used in kernel generation.

In **Listing** 3.7 we can see the definition of the `bin_op` addition node. Binary expressions are implemented through `bin_op` template class which holds three parameters. The

Listing 3.7: The definition of the addition node operation values

```
1    template <>
2    const std::string operation<op::ADD>::_name = "add";
3
4    template <>
5    const std::string operation<op::ADD>::_text = "+";
```

parameters of the `bin_op` template are the semantically equal to the parameters of `un_op` template: the first parameter of the `bin_op` template is the operation represented by the node and the last two parameters are the node's left and right children respectively. For example, an addition is represented with the `ADD` enum. The `operation`'s name is "add" and its representation is done with a "+" string.

As seen in the `un_op`, the first parameter is the definition of the operation at hand. The parameter is a valid member of the same enumeration class. The same structure is used to hold the constant name and text data members.

Both unary and binary template expression members expect to be applied to any type of node, such as containers and fundamental types nodes.

When an operator is called upon, besides generating a valid expression template which is used as a part of the AST, it also generates the correct name for the node itself. The node's name is used in conjunction with sub-nodes to generate the actual tree node's name.

This results in all the nodes of the tree having a name which in turn results in the tree itself having a name. The tree's name is used later on to name the resulting OpenCL file to be run by Marrow.

The name generation is an early optimization that allows the system to know the topology of the tree before visitation. Since the tree's topology is known before kernel generation takes place, the system can use the tree's topology data to check for the pre-existance of the corresponding kernel version.

Kernel generation occurs, during runtime, if no adequate kernel exists on the system. It is possible for the kernel generation to take place at the client application compile phase, however the code was originally geared towards a runtime implementation which invalidated this optimization later on.

To take advantage of **OpenCL** built-in mathematical functions, another node type is used: `call`. This type of node represents the calls to such functions, and in the future, might even be used for user defined **OpenCL** functions, if he/she wishes so. Examples of such functions are the `sqrt`, `dist` or `dot` calls. These calls happen in the context of a vector or scalar. Also implemented through a template class, these functions follow the strategy of the first template parameter being substituted by the structure that holds the call text and the node name.

All nodes, being subclasses of `marrow_expr`, are able to be used in a `sqrt` for instance, allowing for more complex and useful expressions.

Listing 3.8: A division of a vector's square root values by other vector after kernel generation

```
marrow::Vector<int> a(10), b(10), c(10);
// Initialization of vectors
c = a.sqrt() / b;
```

Listing 3.9: A kernel of a division of a vector's square root values by other vector after kernel generation

```
__kernel void main( __global int *a,
 __global int *b,
 __global int *c) {
a = sqrt(b+c)
}
```

Listing 3.10: Power of 2 of a multiplication of two vectors

```
marrow::Vector<int> a(10), b(10), c(10);
// Initialization of vectors
c = (a * b).pow(2);
```

Calls can be made over vectors, mapping the application of the **OpenCL** function to all the elements of the vector.

Calls can also be made over expressions, which can involve any arbitrary number of terms.

The assignment operator has a special meaning within **Marrow**'s containers. Since it is at the tree's root level, it serves as the final operator that deals with the expression template before its submission for computation. The assignment node is the AST's entry point and its left and right side template parameter substitutions are relevant for the system as a whole. The left side of the tree's root node - the assignment node - is related to the vector involved in the left hand side of operation that the tree represents. On the other hand, the right side of the root node is related with the expression received by the assignment operation, and that will therefore change the vector on the other side of the AST.

A right parameter substitution for an assignment node is the operation that will operate on the data. It must be a `call`, `bin_op`, `un_op` or any other valid node instance and corresponds to the right hand side of the expression to be calculated.The left parameter substitution corresponds to the memory area which will be the output buffer that will be

Listing 3.11: Power of 2 of a multiplication of two vectors after kernel generation

```
__kernel void main( __global  int * a,
__global  int * b,
__global  int * c) {
        a=pow(b*c);
}
```

receiving the result of the computation.

Two available data nodes exist for the left side substitution: the vector, being that it extends from the `marrow_expr`, it is a node besides being a container, and `scalar`. `scalar` template class is a wrapper for base types in order to integrate them into the tree. The class is needed in order to have base types as output buffers for the computation. This allows for **Marrow** to see host code variables as output buffers without any specialized commands from the programmer. While needed on the left hand side, the `scalar` wrapper never occurs on the right hand side of the `assignment` node; without the need of any logic to handle the base types – since they do not act as an output buffer then access to their values must not be synchronized or serialized and can be passed by copy – then the fundamental types are used themselves on the tree for performance sake. Vector instances can occur on the left or right hand side of the assignment node. These nodes act as a output buffer when on the right hand side, and their association with the parent tree node, which invariably is a operation node, is done through a reference. Reference usage averted the overhead cost of copying the container when it is associated with a tree.

The computation of the tree occurs after the tree is submitted and dispatched to the runtime.

3.4 Runtime system

When an `operator=` is called it submits the computation, affecting the left side of the operator.

The submission is handled by the `jit::base_runtime` class whose responsibilities are to ensure that previously existing kernels are reused, or, if there are not applicable kernels already generated it guaranties that the correct one is created.

The first action taken by the `jit::base_runtime::submit` method is to initialize an empty `jit::kernel` instance. A `jit::kernel` is a class that holds an yet-to-be-compiled **OpenCL** kernel. It has the filename, the kernel's parameters and body. All kernels reside on the same directory, which can be seen as a kernel library.

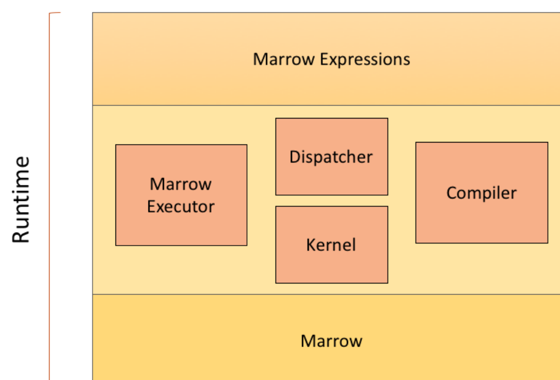

Figure 3.4: Runtime overview.

Listing 3.12: Marrow kernel function.

```
/** kernel parameter section **/
__kernel void marrow_kernel( ) {
  /** function body **/
}
```

The **OpenCL** kernel only exists after a `marrow::jit::kernel` is serialized which results on a well formed file to be created. The kernel's content is a structure which consists on the kernel parameters, if there are any, and the kernel function which is invariably called `marrow_kernel` (Listing 3.13).

The kernel's filename generation stems from the expression template itself; the filename is the concatenation of all the node's `_name` members, which happens at the tree's construction. The filename is then used for checking the existence of the kernel. If the kernel was previously generated at any given time in the past, it is reused by having the filename being provided for **Marrow** for execution. On the other hand, if no file exists which coincides with the filename then code generation follows.

After an adequate `kernel` object is fully defined through code generation and serialized, the kernel is then dispatched for **Marrow** for execution.

3.4.1 Code generation and Marrow dispatching

Code generation is the process used to transform the expression template tree into a valid **OpenCL** kernel that can be subsequently compiled and executed. **Marrow** dispatching, on the other hand, is the collection of input and output information from the tree itself in order to feed the kernel execution.

Code generation takes place at the runtime level, when `base_runtime::submit` asserts that no kernel file exists that corresponds to the tree at hand. When no such kernel is found, a `compiler` instance is called upon the tree receiving both the tree's root and a newly created kernel object.

The `compiler` class was developed using the Visitor design pattern with template methods to deal with the various substitutions possible for the expression template tree. Each node is visit in a top-down, left-right fashion; the visited nodes are interpreted to a textual representation which is pushed on a stack.

Operation nodes, as `bin_op`, `un_op` and `call` have a textual representation embedded within their class definition through the first parameter substitution. Data nodes, as `marrow::vector` or fundamental types are handled differently. The `marrow::vector` template's type parameter is interpreted by the `type_name` static function present on the `compiler` header file which returns the textual representation of the type used; the return string is used on the function's argument definition.

Invariants are used in order to assure that the process and the generated kernel is correct, such as asserting that the processed stack is empty when the processed level

Listing 3.13: Marrow kernel function.

```
template<class T>
    std::string type_name() {
        typedef typename std::remove_reference<T>::type TR;
        std::unique_ptr<char, void (*)(void *)> own(
                nullptr,
                std::free);
        std::string r = own != nullptr ? own.get() : typeid(TR).name();
        if (std::is_const<TR>::value)
            r += " const";
        if (std::is_volatile<TR>::value)
            r += " volatile";
        if (std::is_lvalue_reference<T>::value)
            r += "&";
        else if (std::is_rvalue_reference<T>::value)
            r += "&&";
        return r;
    }
```

reaches the root node.

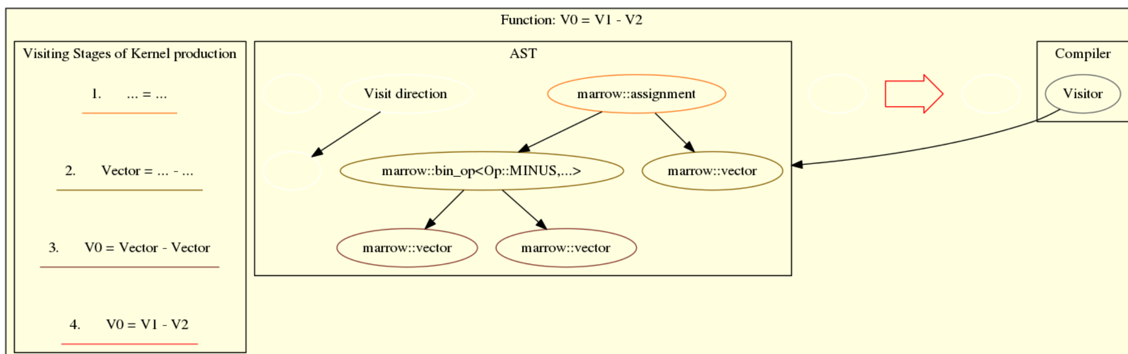

Figure 3.5: The compiler visiting a tree

Binary and unary operation nodes are processed taking into account the operation only after the left and right hand side are processed. The operation itself is processed afterwards and the text is concatenated in left, operation and right order. The call nodes are evaluated in a different order; since a call operates over the arguments it receives, the call node's `_text` is first added to the stack, having the call's arguments evaluated afterwards.

Kernel dispatching for execution happens after an adequate kernel exists. In this case, an instance of a recursive template class `marrow_executor` recursively visits the tree. Similar to the `compiler` structure, the `marrow_executor` follows the `visitor` design pattern. Unlike the `compiler`, however, this visiting is defined and happens at compile time instead of runtime. This is necessary as the prime objective of this visitor is to extract the type information held within the tree, and subsequently invoke **Marrow** with that information.

Table 3.1: Examples of Marrow calls for some host code lines.

type	value	parameter	flow
T*	vector<T>	T*	input/output
T	vector<T>	T*	input
T	T	T	input
T*	T	T*	input/output

The main focus in developing this visitor was the use of the `map` concept, mapping a kernel to every element of the containers. It is the `map` call which consumes the data this visitor collects. `marrow_executor` takes advantage of **Marrow**'s `KernelWrapper` which is a structure that represents an **OpenCL** kernel in **Marrow**, wrapping all the information that the framework needs for such a computational entity: name of the file and function and its interface.

The call made by `marrow_executor` in the case of a kernel that adds two vectors, altering a third, would then have parameterized the tree vectors on the actual call.

3.5 AST submission and kernel execution

The kernel is then finally dispatched to **Marrow** by the `map` construct. The call itself returns a `shared_future` instance which is waited on by the thread that launched it.

Since `base_runtime` had the whole operation executed within a thread running on a threadpool, this means that all operations above described are asynchronous and serialization happens only when the container is accessed. This is handled through the overloads of `operator=` and `operator[]`. `operator=` is responsible for the submission of the tree which gets compiled into a kernel and dispatched to **Marrow** at which time **Marrow** returns the `shared_future` instance described.

This `shared_future` instance returned by **Marrow** happens within the **CTPL** thread context, at which point the future is actively awaited upon. While the **CTPL** thread actively waits for the **Marrow**'s `shared_future` to have its value set, the **CTPL** thread has its own `future` which is returned to `base_runtime`. Since `base_runtime` has no interest on the result of the computation, the future is handed off to the `vector` instance which submitted the tree.

vector's `operator=` holds the `shared_future` that **Marrow** returned and whenever `operator[]` is called it checks if the instance is valid. If it is, it is waited upon. The `base_runtime` future's value is set only when **Marrow** has finished the kernel execution and corresponding data-transfers; at that point the vectors' data should be synchronized in order to allow a safe access the memory.

Chapter 4

Kernel fusion

An important dimension to measure heterogeneous programming frameworks is by the number of data transfers and code uploads done. The strategy of capturing operations over containers and converting those to expression templates, in order to have the same operations execute in a device's context, resulted on a multitude of kernels being compiled and uploaded.

Every captured operation corresponds to a single line kernel; no matter how complex the host code line is, the tree created represents one operation. Since the expression template represents only one operation, this means that all generated kernels are simple operations, equivalent to the ones that triggered the operation capture. This also means that a multitude of kernels may be generated, one for each possible expression. The computation of these kernels meant multiple kernel uploads for the GPU device; it also meant that for each operation, the same buffers were uploaded and fetched, as can also be seen on SkelCL or SkePU

In order to be minimize data and code uploads to GPU, the uploaded kernel should contain all the expressions it can so that only one execution is made.

Before a vector's element value is computed, all the operations which were captured before-hand are waited upon, guaranteeing the new elements values are correct. This also caused the serialization of the kernel execution, since a operation's terms may have pending kernels of their own.

Listing 4.1: Two expressions causing the execution of two kernels

```
1 std::vector<int> a(10), b(10), c(10);
2 // initialize a b and c
3 c = a + b;  \\ causes execution of one kernel
4 c = c*2;  \\ causes execution of another kernel
```

Listing 4.2: Two expressions causing the execution of two kernels

```
__kernel void main( __global  int * c,
__global  int * a,
__global  int * b,
const int d) {
        int tid = get_global_id(0);
        c[tid]=a[tid]+b[tid];
        c[tid]=c[tid]*2;
}
```

Listing 4.3: Valid values for computation state

```
enum computation_state {
  NOCOMPUTATION,
    UNSUBMITTED,
    SUBMITTED,
    DISPATCHED,
    EXECUTED,
    CANCELLED
};
```

Listing 4.4: Automatic fusion

```
v1 = v2 + v3
v4 = v1 + v5
```

Kernel fusion was chosen as the strategy to mitigate this problem, allowing a kernel to aggregate all instructions needed to correctly compute its value. It would allow for computations to be aggregated when they are yet to be computed. As such, a fusion mechanism was developed.

The first addition to the system was the addition of a `_computation_state` enumeration member. This member allowed a tracking of the computation state, so that the fusion would only occur between expressions that are yet to be computed.

The status of any tree is set to unsubmitted until it is submitted; once the tree is handled by the `base_runtime::submit` method, its status is changed to submitted. Once the kernel is compiled and right before the **Marrow** `map` is called the status is changed for dispatched, and after the calling `future::get` the `operator[]` changes status to executed. Two other states were also introduced which are `nocomputation` and `cancelled`. `nocomputation` state is related to trees that do not need to be computed. Examples of such trees are single nodes (i.e. a vector). `cancelled` node represents a expression that has been fused with, and whose computation will be delivered by the fusion resulting tree.

A fusion occurs when a tree is not yet submitted; fusion happens both automatically or forced through a specific fusion operator. Automatic fusion occurs whenever a assignment operator is called and is given a expression whose terms have pending computations yet to be submitted.

Listing 4.5: Automatic fusion

```
1 v1 << v2 + v3
2 v4 = v1 + v5
```

Whenever a instance's `operator=` is called the tree is submitted for execution with fusion information. If pending operations exist, all the operations get fused together on a single kernel.

When a vector's `operator«` is called upon, it receives a valid expression template as its argument. It calls the `compiler` instance, and executes a compilation of the tree. The resulting kernel is put aside on a stack structure within the vector's instance, which will be used to generate the final kernel.

The same operation can be seen in **Listing** 4.4 and **Listing** 4.5, the difference being that in **Listing** 4.4 fusion may happen, while in **Listing** 4.5 is guaranteed a fusion. The difference stems from the semantics involved with `operator=` and `operator«`. `operator=` creates an assignment node, representing the assignment of the vector's instance with the resulting computation value. This new composition is immediately submitted to the run-time for computation. `operator«` behaves differently, creating the same new AST node as `operator=`, with the same meaning but refrains from issuing the tree to the run-time for computation. Instead, `operator«` uses a `compiler` instance to transform the expression into a `kernel` object and saving the object onto a file in persistent memory.

A new expression template node was created, the `variable` node. The `variable` node is an abstraction over `vector` and `scalar` alike and is a direct descendant from `marrow_expr`.

`variable` holds another structure that helps the fusion mechanism, the `fusion_info` structure. Since **Marrow expressions** focused on converting captured operations to template expressions, issuing them to compute when a `operator=` is called upon, a mechanism to save the memory positions involved in the fusion was developed.

The `fusion_info` structure is used by both `marrow_executor` and `compiler`, because it holds the input and output information involved in the resulting fused kernel.

`fusion_info` hold all the type information related to the buffers that will be feed to the final kernel for computation; the final kernel is fed all the data present on the `_fusion_info` member by the `marrow_executor` when the execution is done.

Listing 4.6 shows the resulting fused kernel of **Listing** 4.5. As can be observed the resulting kernel has six parameters corresponding to the vectors occurring in the original operations in **Listing** 4.5. However the number of kernel parameters is greater than the number of vectors used in the operations on the host code.

This is explained by the compiler attributing an incremental id to each vector it encounters on all the trees processed. Each occurrence of the same vector in one or more expressions will be translated into different ids, as each occurrence is a distinct vector. Marrow will also interpret each occurrence as a distinct vector, thus copying the vector

Listing 4.6: Addition example kernel

```
__kernel void marrow_kernel( __global  int * v1,
  __global  int * v2,
  __global  int * v3,
  __global  int * v4,
  __global  int * v5,
  __global  int * v6) {

  int tid = get_global_id(0);
  v1 = (v2 + v3);
  v4 = (v5 + v6);
}
```

to the device multiple times. In the case presented in **Listing** 4.6, the *a* vector will be mapped to both $v1$ and $v5$, which in turn will cause the second operation to erroneously be calculated with the original value of *a*, instead of the updated one. This is a limitation being solved, whose solution will involves the systematic elimination of argument duplication on the `compiler` tree processing algorithm and the elimination of the duplication of vector uploads done by `marrow_executor`.

CHAPTER 5

EVALUATION

In this chapter the present work will be evaluated in order to assess the feasibility of the approach to heterogeneous programming. Since the focal point of the approach is to provide an abstraction from the framework's backend, the abstraction level must be itself evaluated.

5.1 Case studies

To measure the abstraction gained and the corresponding performance, it is important to compare the system's results with the same algorithms in Marrow's previous version implementation. To do so, two tests were implemented: Saxpy and the computation of trees with three-, four- and five-levels. The description of the tests is made in detail in the following lines:

Saxpy This case study is a matrix operation which is part of BLAS (Basic Linear Algebra Subroutines). It performs the multiplication of a matrix with a scalar value, and sums the result with a second matrix (y[i] = s*x[i] + y[i]). Both these operations are data-parallel as they only require the elements at the current position in both matrices in its computation. In this case study the mapping of arithmetic operations to the backend through Marrow expressions is compared against other mechanisms.

Three-, four- and five-level tree are three case studies designed to measure the overhead of the code generation process compared to direct Marrow API usage. This case study was divided into three tests, one with a three-, one with a four-, and one with a five-level tree to understand if the complexity of the trees influence the executions. All three tests were implemented in both Marrow, with OpenCL, and in a standalone C++ program that

took advantage of Marrow Expression which generated and ran an adequate set of kernels which computed the result.

5.2 Metrics

To evaluate Marrow Expressions the metrics to be used are the abstraction gained using the new interface and the performance achieved with different vector sizes. All the following tests are done with both the Marrow Expressions and the Marrow counterpart, except for the abstraction study, where the comparison is done with the final OpenCL code that Marrow uses. The evaluation starts with the abstraction analysis, which is approached in **Sec** 5.4. The three-, four- and five-level tests overheads are presented in **Sec** 5.5, from running both implementations in order to measure the overhead of the chosen technique.

5.3 Methodology

The case studies presented in **Sec** 5.1 are submitted to 1000 runs per test. One test represents the 1000 runs with a certain number of elements for each vector in the case study computation. The data-sets chosen are vectors with 10^4, 10^5 and 10^6 elements. This values have an increase of a factor of ten concerning the element size involved between them, in order to guarantee that there is a sufficient increase of elements from test to test. For each test we calculate the minimum, the maximum, the average execution times and the standard deviation. The execution times involve the entire run of the case study, from the point were the system to be evaluated is called to the point of its execution end. The case studies are implemented in both systems in the most direct and simple way in order to have the most reliable comparisons.

5.4 Abstraction

The abstraction is evaluated by the number of lines of code needed for the implementation of the case studies. As aforementioned, its implementations are the most straightforward - with the minimum lines of code needed - turning the comparisons between Marrow's code and Marrow's OpenCL code. Since kernel fusion is still under work, the tests chosen are simple operations that grow in complexity to show compilation mechanism at work and the abstraction from Marrow itself. In the next paragraphs the evaluation of the abstraction of each approach is described for the examples created.

Every code written for the abstraction evaluation is on **Appendix** 7. The code reduction is primarily in the OpenCL code, since the generated kernel substitutes the Marrow's kernel counterpart. There is a significant abstraction from the OpenCL since no OpenCL code was actually written by the programmer for the Marrow expressions, while the Marrow counterpart had its device code hand-written. The host code captured operation also generates Marrow calls whose responsibilities are to launch the kernels themselves.

On the other hand, the Marrow implementations have Marrow calls to launch the kernel which were implemented by the programmer. We also have some abstraction from Marrow at this level.

The joint abstraction from both OpenCL code and Marrow result on an abstraction from the **SIMD** architecture; with the presented programming model, the programmer sees the programming paradigm as a sequential, single operation machine.

5.5 Overhead

Overhead was measured in order to extract the loss of performance associated with our implementation. The overhead was calculated using Marrow's performance as a basis for comparison, since Marrow expressions uses Marrow as a framework from which to work on.

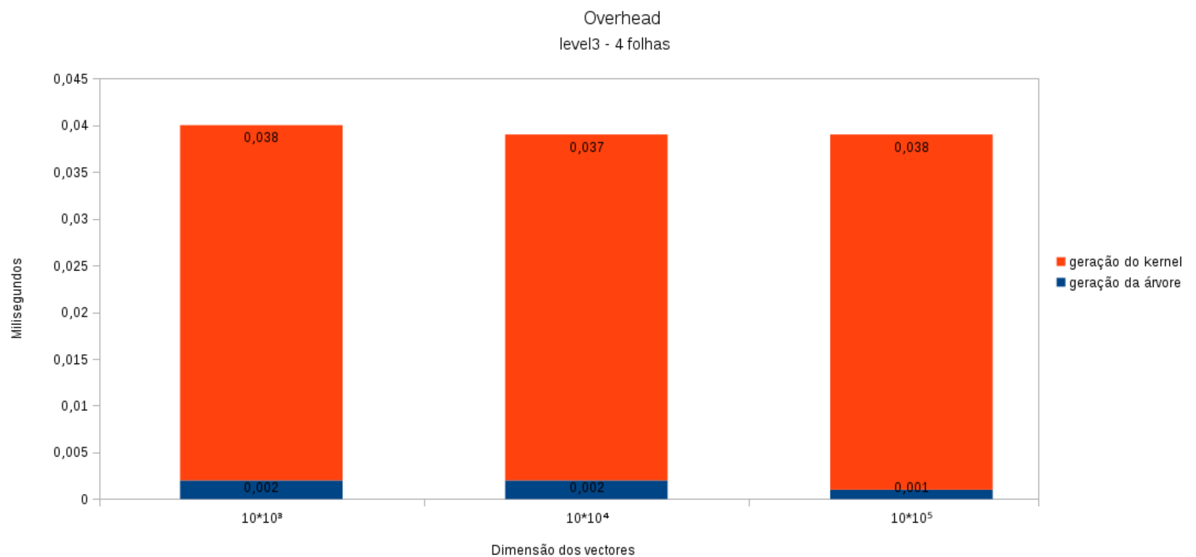

Figure 5.1: Overhead of level-three.

In this section the case studies are compared in terms of their runtime average values. The algorithms studied reflect simple operations that should not impact the overall time spent in OpenCL.

From analyzing the results seen in **Fig** 5.1, **Fig** 5.2 and **Fig** 5.3, we see that Marrow expressions has a overall performance penalty, since all expression template must be evaluated in order to decide which kernel to use. This performance penalty is constant and, as such, as the time spent computing the kernel increases, the percentage of time spent on expression template evaluation tends to 0.

Significant penalty exists on the kernel generation; subsequent calls to the same kernel seem to behave differently with different vector sizes As can be seen across **Fig** 5.1, **Fig** 5.2 and **Fig** 5.3 concerning 10^4 and 10^5 and 10^6, the results point that only after a significant

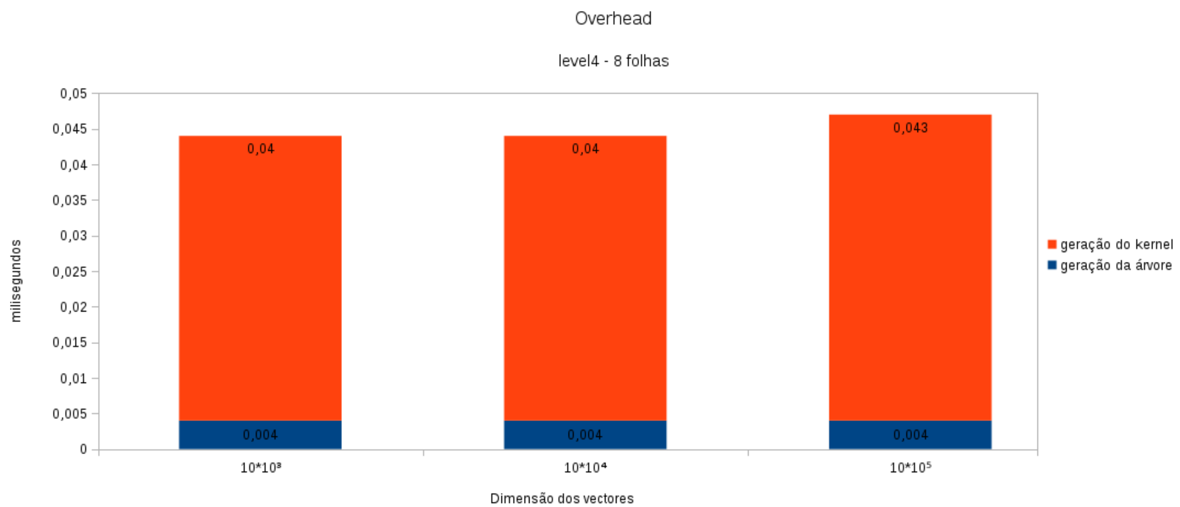

Figure 5.2: Overhead of level-four.

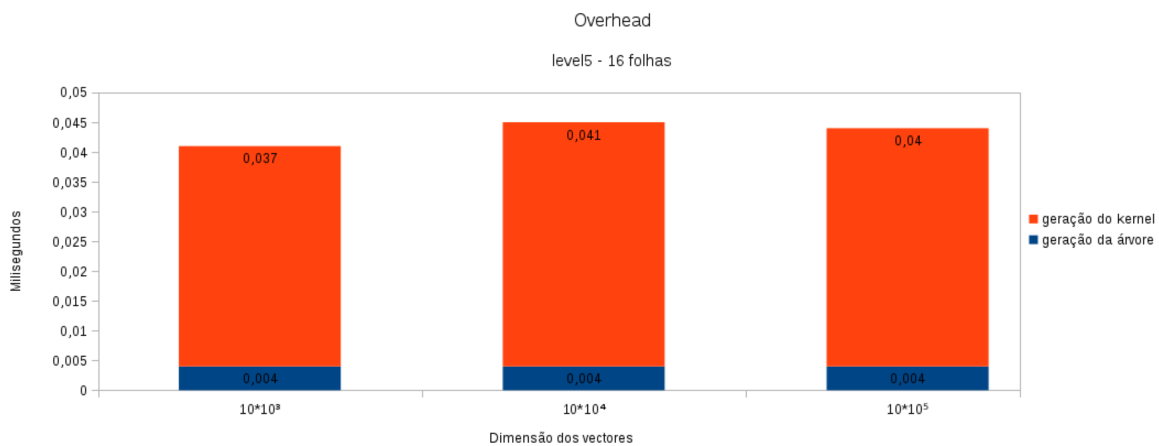

Figure 5.3: Overhead of level-five.

vector size increase does the overall performance be defined by the data-transfer instead of the expression template penalty, however as the vector size increases, the impact done by the expression template itself is less important.

5.6 Final remarks

While there is some performance penalty with the usage of Marrow Expressions, it becomes clear that the major performance impact comes from the first execution of a given expression. The first execution of an expression generates an adequate kernel, which is written to disk and, after subsequent dispatch to Marrow, the kernel is compiled by OpenCL to be executed. This whole process is cut short after the first execution since the kernel already exists and the compilation of it already took place. Even though the costly

generation and compilation takes place only at the first execution, the expression template mechanism itself has an associated cost in execution time, since this portion of the logic is always executed independently of the iteration considered.

As vector size increases, the time spent in expression template generation is comparably less than what is spent in Marrow runtime with transferring of data to and from the device.

CHAPTER 6

CONCLUSION

This final chapter presents a summarized view of the described work. Starting by highlighting some core aspects about the work (section 6.1), such as what are the objectives, what came out of our work, how the work was evaluated and what were the evaluation results. On the last section some thoughts concerning future research topics are presented.

6.1 Objectives and results

The present work identified some shortcomings in the present approaches for the development of complex software using general purpose GPU computing. All of the discussed approaches require learning either new languages or complex concepts associated with the specific approach.

Structured approaches, like the skeleton-driven approaches presented addressed on **Chapter** 2, require the user to know **OpenCL** in order to extend the functionality already present on the approach at hand. On the other hand, unstructured approaches, like linguistic-based approaches, require the user to learn a new language (**StreamIt**, **Dandelion**), or constructs (**OpenACC**). For this purpose, an objective of the present work was to offer a programming model that abstracts from the back end used, which in **Marrow**'s case is **OpenCL**.

The work herein documented resulted on a functional prototype, a C++ abstraction layer that incorporates all of the previous objectives, namely: introducing new functionality to the **Marrow** framework, providing transparent programming model via expression templates, enabling both kernel fusion and a **OpenCL** abstraction.

The proposal's evaluation was split into two sub-domains: performance and abstraction. The first sub-domain was concerned with the performance losses that could be involved with the automatic **OpenCL** kernel generation while the second sub-domain directly compares the complexity of the template expression approach using lines of code.

Listing 6.1: An assignment tree

```
marrow::vector<int> a(10), b(10), c(10);
// initialize all elements from a and b
a = a + b;
c = invert(a)
```

The abstraction comparison was done with regard to **Marrow**, on which the present work build upon, and **OpenCL** which is used by **Marrow** as a back-end.

The evaluation attested the effectiveness of the present proposal. Compared to hand-tuned **OpenCL** applications, the expression template usage promoted an abstraction of the examples presented with no direct usage of **Marrow** or **OpenCL** primitives by the programmer with some performance penalties.

While the fusion mechanism remained a work in progress at the time of writing, significant progress was made in order to allow for the kernel fusion to take place, thus minimizing the number of uploads to the device.

6.2 Future work

Future work will focus on two major aspects: increase the domain of expressible functionality and optimization of execution.

The completion of the kernel fusion mechanism is a short-term goal as it has major impact in the behavior of the system, in terms of transfers done.

The addition of more expressiveness lies with the automatic translation of utility functions, like mathematical ones, to the **OpenCL** counterparts; implementation of these would allow for a greater abstraction from the back-end used. Greater integration with **Marrow**'s Skeleton Computational Trees could be used in order to achieve a greater expressiveness. Also, some algorithms cannot be expressed using Marrow expressions, since no notion of barrier exists on Marrow Expressions. To allow for algorithms, like multiplying all elements of a matrix with a scalar and inverting the resulting matrix, to be written Marrow Expressions must be able to express a synchronization barrier in device context, to allow for the first logical part to conclude before advancing to the second part,

Automatic identification and corresponding handling of cases where some form of barrier is needed is an important requirement to promote a very abstract program model. This is a necessary step for fused kernels which comprise of algorithms where one part of the algorithm depends on all the elements of the memory to be synchronized.

Using expression templates and automatic kernel generation this hypothetical example could not have the two corresponding trees fused together. To execute any form of inversion over a, the computation that defines it must be concluded before-hand. As such, this would require a automatic detection of such usages of expression templates and behave accordingly, which in this case is to wait for the computation of a to conclude.

Futhermore, additional back-end may be added, since the back-end independent nature of Marrow expression allows for the compiler to be adapted to develop other back-end code like OpenMP.

Bibliography

[1] N. Goswami, R. Shankar, M. Joshi, and T. Li. "Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications". In: *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 1–10.

[2] *CUDA Toolkit Documentation*. URL: http://docs.nvidia.com/cuda (visited on 01/22/2015).

[3] B. R. Gaster and L. Howes. "OpenCL C+". In: (2013). URL: http://benedictgaster.org/wp-content/uploads/2013/01/CLHPPGPGPU6PrePrintAsSubmitted-1.pdf (visited on 01/11/2015).

[4] K. Matsuzaki and K. Emoto. "Lessons from implementing the biCGStab method with SkeTo library". In: *Proceedings of the fourth international workshop on High-level parallel programming and applications*. ACM, 2010, pp. 15–24. URL: http://dl.acm.org/citation.cfm?id=1863488 (visited on 01/11/2015).

[5] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. "A fusion-embedded skeleton library". In: *Euro-Par 2004 Parallel Processing*. Springer, 2004, pp. 644–653.

[6] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. "A library of constructive skeletons for sequential style of parallel programming". In: *Proceedings of the 1st international conference on Scalable information systems*. ACM, 2006, p. 13.

[7] H. Iwasaki and Z. Hu. "A new parallel skeleton for general accumulative computations". In: *International Journal of Parallel Programming* 32.5 (2004), pp. 389–414.

[8] M. Steuwer, P. Kegel, and S. Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW). 2011, pp. 1176–1182. DOI: 10.1109/IPDPS.2011.269.

[9] J. Enmyren and C. W. Kessler. "SkePU: a multi-backend skeleton programming library for multi-GPU systems". In: ACM Press, 2010, p. 5. ISBN: 9781450302548. DOI: 10.1145/1863482.1863487. URL: http://portal.acm.org/citation.cfm?doid=1863482.1863487 (visited on 01/11/2015).

[10] U. Dastgeer, J. Enmyren, and C. W. Kessler. “Auto-tuning skepu: a multi-backend skeleton programming framework for multi-gpu systems”. In: *Proceedings of the 4th International Workshop on Multicore Software Engineering*. ACM, 2011, pp. 25–32. URL: http://dl.acm.org/citation.cfm?id=1984697 (visited on 01/11/2015).

[11] M. Majeed, U. Dastgeer, and C. Kessler. “Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters”. In: *Proc. Int. Conf. on Parallel and Distr. Processing Techniques and Applications (PDPTA)*. 2013. URL: http://www.ida.liu.se/~usmda/skepu/publications/ClusterSkepUPDPTA.pdf (visited on 01/11/2015).

[12] R. Marques, H. Paulino, F. Alexandre, and P. D. Medeiros. “Algorithmic skeleton framework for the orchestration of GPU computations”. In: *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 874–885.

[13] F. Soldado, F. Alexandre, and H. Paulino. “Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments”. In: *Concurrency and Computation: Practice and Experience* 28.3 (2016), pp. 768–787. DOI: 10.1002/cpe.3612.

[14] F. Alexandre, R. Marques, and H. Paulino. “On the support of task-parallel algorithmic skeletons for multi-GPU computing”. In: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. ACM, 2014, pp. 880–885. DOI: 10.1145/2554850.2555018.

[15] *Thrust*. URL: http://docs.nvidia.com/cuda/thrust/#axzz44DFORMjG (visited on 03/28/2016).

[16] *Bolt c++ template library*. URL: http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library/ (visited on 03/28/2016).

[17] Association for computing machinery (Etats Unis). and Special interest group on programming languages. *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation: PLDI’12*. New York N.Y.: Association for Computing Machinery, 2012. ISBN: 9781450312059 1450312055.

[18] *The Liquid Metal Programming Language - Language Reference Manual*. URL: http://domino.research.ibm.com/library/cyberdig.nsf/papers/43E23EC91521C1E3852577BC \$File/rc25004.pdf (visited on 03/28/2016).

[19] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly. “Dandelion: a Compiler and Runtime for Heterogeneous Systems”. In: SOSP’13: The 24th ACM Symposium on Operating Systems Principles, Nov. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=201110.

[20] W. Thies, M. Karczmarek, and S. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *International Conference on Compiler Construction*. Grenoble, France, 2002. URL: http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf.

[21] *OpenAcc Programming guide*. URL: http://www.openacc.org/sites/default/files/OpenACC_Programming_Guide_0.pdf (visited on 03/28/2016).

[22] M. Cole. *Algorithmic skeletons*. Springer, 1999.

[23] K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi. "A Compositional Framework for Developing Parallel Programs on Two-Dimensional Arrays". In: *International Journal of Parallel Programming* 35.6 (Sept. 28, 2007), pp. 615–658. ISSN: 0885-7458, 1573-7640. DOI: 10.1007/s10766-007-0043-4. URL: http://link.springer.com/10.1007/s10766-007-0043-4 (visited on 07/20/2014).

[24] U. Dastgeer, Linkopings universitet., and Institutionen for datavetenskap. "Skeleton programming for heterogeneous GPU-based systems". PhD thesis. Linkoping: Department of Computer and Information Science, Linkoping University, 2011.

[25] U. Dastgeer and C. Kessler. "A performance-portable generic component for 2d convolution computations on GPU-based systems". In: *Proc. MULTIPROG-2012 Workshop at HiPEAC-2012, Paris*. 2012, pp. 1–12. URL: http://www.ida.liu.se/~usmda/papers/multiprog_2012_convol.pdf (visited on 01/11/2015).

[26] U. Dastgeer, L. Li, and C. Kessler. "Adaptive implementation selection in the SkePU skeleton programming library". In: *Advanced Parallel Processing Technologies*. Springer, 2013, pp. 170–183. URL: http://link.springer.com/chapter/10.1007/978-3-642-45293-2_13 (visited on 01/11/2015).

[27] K. Emoto, K. Matsuzaki, Z. Hu, and M. Takeichi. "Domain-specific optimization strategy for skeleton programs". In: *Euro-Par 2007 Parallel Processing*. Springer, 2007, pp. 705–714. URL: http://link.springer.com/chapter/10.1007/978-3-540-74466-5_74 (visited on 01/11/2015).

[28] K. O. W. Group et al. "The opencl specification". In: *version* 1.29 (2008), p. 8.

[29] J.-F. Dollinger and V. Loechner. "Adaptive runtime selection for GPU". In: *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE. 2013, pp. 70–79.

[30] D. Demidov, K. Ahnert, K. Rupp, and P. Gottschling. "Programming CUDA and OpenCL: a case study using modern C++ libraries". In: *SIAM Journal on Scientific Computing* 35.5 (2013), pp. C453–C472.

[31] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. "Polyhedral parallel code generation for CUDA". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), p. 54.

[32] U. Bawidamann and M. Nehmeier. "Expression templates and OpenCL". In: *Parallel Processing and Applied Mathematics* (2012), pp. 71–80.

[33] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula. "Generating efficient data movement code for heterogeneous architectures with distributed-memory". In: *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. IEEE. 2013, pp. 375–386.

[34] K. Matsuzaki and K. Emoto. “Implementing fusion-equipped parallel skeletons by expression templates”. In: *Implementation and Application of Functional Languages*. Springer, 2010, pp. 72–89.

[35] D. Demidov. *VexCL: Vector expression template library for OpenCL*. 2014.

Chapter 7

Appendix

7.1 Marrow saxpy implementation

Listing 7.1: Listings/marrow/saxpy.cl

```
/**
 Created by Ricardo Marques
**/

__kernel void saxpy(__global float *X, __global float *Y, const float a,
    __global float *out)
{
  int pos = get_global_id(0);
  out[pos] = a * X[pos] + Y[pos];
}
```

Listing 7.2: Listings/marrow/saxpy.cpp

```
#include <marrow.h>
#include "marrow/utils/Benchmark.hpp"

using namespace std;
using namespace marrow;

const string kernelFile = "saxpy.cl";
const string saxpyKernel = "saxpy";
const float inputValue = 10.0f;
const float a = 10.0f;

class Saxpy: public marrow::Benchmark {
private:
```

```
14     unsigned int numberElements;
15     Vector<float> inVector1;
16       Vector<float> inVector2;
17       Vector<float> outVector;
18     std::unique_ptr<
19         Map<KernelWrapper<const float, const float, const float, float*>,
20             Vector<float>&, Vector<float>&, const float&, Vector<float>&>>
                 skeleton_tree;
21   public:
22     Saxpy(int nrElements, int argc, char* argv[]) :
23         marrow::Benchmark("marrow_saxpy", argc-1, &argv[1]),
24         numberElements (nrElements),
25         inVector1 (numberElements),
26         inVector2 (numberElements),
27         outVector (numberElements) {
28       inVector1.fill([](int) { return inputValue; });
29       inVector2.fill([](int) { return inputValue; });
30     }
31
32     static string usage(string program) {
33       return program + " number_elements " + Benchmark::usage();
34     }
35   protected:
36     void init() {
37       skeleton_tree = make_map(
38           kernel<const float, const float, const float, float*> (
39           resolveKernel(kernelFile), saxpyKernel),
40           Args<Vector<float>&, Vector<float>&, const float&, Vector<float>&>
               {} );
41     }
42
43     void run() {
44       marrow::future future = skeleton_tree->run(inVector1, inVector2, a,
             outVector);
45       future.get();
46     }
47
48     bool validate() {
49       const float referenceValue = inputValue*a + inputValue;
50       for (unsigned int i = 0; i < numberElements; i++)
51         if (outVector[i] !=  referenceValue)
52           return false;
53       return true;
54     }
55   };
56
57   int main(int argc, char* argv[]) {
58     if (argc < 2) {
59       cerr << Saxpy::usage(string (argv[0])) << endl;
60       return EXIT_FAILURE;
```

```
61    }
62
63    int n = atoi(argv[1]);
64    if (n <= 0) {
65      cerr << "Wrong value for \"number_elements\"." << endl <<
66      Saxpy::usage(string (argv[0])) << endl;
67      return EXIT_FAILURE;
68    }
69
70    try {
71      Saxpy s(n, argc, argv);
72      s.timedRuns();
73    } catch (std::runtime_error& e) {
74      cerr << "Error : " << e.what() << std::endl << std::endl;
75      return EXIT_FAILURE;
76    }
77
78    return 0;
79  }
```

7.2 Marrow expression saxpy implementation

Listing 7.3: Listings/marrow-expressions/saxpy.cpp

```
1   #include <marrow.h>
2   #include "marrow/utils/Benchmark.hpp"
3
4   using namespace std;
5   using namespace marrow;
6   static const int inputValue = 10;
7   static  int a = 10;
8
9   class Saxpy: public marrow::Benchmark {
10      unsigned int numberElements;
11      Vector<int> inVector1, inVector2, outVector;
12  public:
13      Saxpy(int nrElements, int argc, char* argv[]) :
14              marrow::Benchmark("jit_saxpy", argc-1, &argv[1]),
15              numberElements (nrElements),
16              inVector1 (numberElements),
17              inVector2 (numberElements) {
18          inVector1.fill([](int) { return inputValue; });
19          inVector2.fill([](int) { return inputValue; });
20      }
21
22      void init() {}
23      void run() {
24          outVector = inVector1*a + inVector2;
25          auto a = outVector[0];
26      }
```

```
27
28      bool validate() {
29          const int referenceValue = inputValue*a + inputValue;
30          for (unsigned int i = 0; i < numberElements; i++)
31              if (outVector[i] !=  referenceValue)
32                  return false;
33
34          return true;
35      }
36  };
37
38  int main(int argc, char* argv[]) {
39      if (argc < 2) {
40          cerr << argv[0] << " <element_count>" << endl;
41          return EXIT_FAILURE;
42      }
43
44      int n = atoi(argv[1]);
45      if (n <= 0) {
46          cerr << "Invalid vector size." << endl
47          << argv[0] << " <element_count>" << endl;
48          return EXIT_FAILURE;
49      }
50
51      try {
52          Saxpy s(n, argc, argv);
53          s.timedRuns();
54      } catch (std::runtime_error& e) {
55          cerr << "Error : " << e.what() << std::endl << std::endl;
56          return EXIT_FAILURE;
57      }
58      return 0;
59  }
```

7.3 Marrow sqrt implementation

Listing 7.4: Listings/marrow/sqrt.cl

```
1  /**
2   Created by Ricardo Marques
3  **/
4
5  __kernel void kernel_f(__global float *X, __global float *Y, __global float
       *out)
6  {
7    int pos = get_global_id(0);
8    out[pos] = sqrt(X[pos] + Y[pos]);
9  }
```

Listing 7.5: Listings/marrow/sqrt.cpp

```

#include "marrow.h"
#include "marrow/utils/Benchmark.hpp"

using namespace marrow;
using namespace std;

const string kernelFile = "sqrt.cl";
const string saxpyKernel = "kernel_f";

// Problem related constants
const float inputValue = 10.0f;
const float a = 10.0f;

class Sqrt: public marrow::Benchmark {
private:
  unsigned int numberElements;

  Vector<float> inVector1;
    Vector<float> inVector2;
    Vector<float> outVector;

  std::unique_ptr<
      Map<KernelWrapper<const float, const float, float*>,
          Vector<float>&, Vector<float>&, Vector<float>&>> skeleton_tree;

public:
  Sqrt(int nrElements, int argc, char* argv[]) :
      marrow::Benchmark("marrow_sqrt", argc-1, &argv[1]),
      numberElements (nrElements),
      inVector1 (numberElements),
      inVector2 (numberElements),
      outVector (numberElements)
  {

    for (unsigned int i = 0; i < numberElements; i++) {
      inVector1[i] = inputValue;
      inVector2[i] = inputValue;
    }
  }

  static string usage(string program) {
    return program + " number_elements " + Benchmark::usage();
  }

protected:
//  void setup_skeleton_tree() {
void init() {
```

```
49
50      skeleton_tree = make_map(
51          kernel<const float, const float, float*> (
52          resolveKernel(kernelFile), saxpyKernel),
53          Args<Vector<float>&, Vector<float>&, Vector<float>&> {} );
54
55    }
56
57    void run() {
58      marrow::future future = skeleton_tree->run(inVector1, inVector2,
          outVector);
59      future.get();
60    }
61
62    bool validate() {
63      const float referenceValue = std::sqrt(inputValue + inputValue);
64
65      for (unsigned int i = 0; i < numberElements; i++)
66        if (outVector[i] !=  referenceValue)
67          return false;
68
69      return true;
70    }
71  };
72
73  int main(int argc, char* argv[]) {
74    if (argc < 2) {
75      cerr << Sqrt::usage(string (argv[0])) << endl;
76      return EXIT_FAILURE;
77    }
78
79    int n = atoi(argv[1]);
80    if (n <= 0) {
81      cerr << "Wrong value for \"number_elements\"." << endl <<
82          Sqrt::usage(string (argv[0])) << endl;
83      return EXIT_FAILURE;
84    }
85
86    try {
87      Sqrt s(n, argc, argv);
88      s.timedRuns();
89    } catch (std::runtime_error& e) {
90      cerr << "Error : " << e.what() << std::endl << std::endl;
91      return EXIT_FAILURE;
92    }
93
94    return 0;
95  }
```

7.4 Marrow expression sqrt implementation

Listing 7.6: Listings/marrow-expressions/sqrt.cpp

```
#include <math.h>
#include <marrow.h>
#include "marrow/utils/Benchmark.hpp"

using namespace std;
using namespace marrow;
static const int inputValue = 10;

class Sqrt: public marrow::Benchmark {
    unsigned int numberElements;
    Vector<float> inVector1;
    Vector<float> outVector;
public:
    Sqrt(int nrElements, int argc, char* argv[]) :
            marrow::Benchmark("jit_sqrt", argc-1, &argv[1]),
            numberElements (nrElements),
            inVector1 (numberElements),
            outVector (numberElements) {
        inVector1.fill([](int) { return inputValue; });
    }

    void init() {}
    void run() {
        outVector = inVector1.sqrt();
        auto o = outVector[0];
    }

    bool validate() {
        const float referenceValue = std::sqrt(inputValue + inputValue);

        for (unsigned int i = 0; i < numberElements; i++)
            if (outVector[i] !=  referenceValue)
                return false;

        return true;
    }
};

int main(int argc, char* argv[]) {
    if (argc < 2) {
        cerr << argv[0] << " <element_count>" << endl;
        return EXIT_FAILURE;
    }

    int n = atoi(argv[1]);
```

```
46      if (n <= 0) {
47          cerr << "Invalid vector size." << endl
48          << argv[0] << " <element_count>" << endl;
49          return EXIT_FAILURE;
50      }
51
52      try {
53          Sqrt s(n, argc, argv);
54          s.timedRuns();
55      } catch (std::runtime_error& e) {
56          cerr << "Error : " << e.what() << std::endl << std::endl;
57          return EXIT_FAILURE;
58      }
59
60      return 0;
61  }
```

7.5 Marrow twolevels implementation

Listing 7.7: Listings/marrow/twolevels.cl

```
1  __kernel void two_levels(
2      __global float* o,
3      __global float* a,
4      __global float* b) {
5      int i = get_global_id(0);
6      o[i] = (a[i]+b[i]);
7  }
```

Listing 7.8: Listings/marrow/twolevels.cpp

```
1  #include "marrow.h"
2  #include "marrow/utils/Benchmark.hpp"
3
4  using namespace marrow;
5  using namespace std;
6
7  const string kernelFile = "two_levels.cl";
8  const string kernelFunction = "two_levels";
9
10 class ThreeLevels: public marrow::Benchmark {
11 private:
12   unsigned int numberElements;
13   Vector<int> O;
14   Vector<int> A, B;
15
16   std::unique_ptr<
17         Map<KernelWrapper<int*, const int, const int>,
18             Vector<int>&, Vector<int>&, Vector<int>&>> skeleton_tree;
```

```
19
20 public:
21   ThreeLevels(int nrElements, int argc, char* argv[]) :
22       marrow::Benchmark("marrow_two", argc-1, &argv[1]),
23       numberElements (nrElements),
24       O (numberElements),
25       A (numberElements), B (numberElements)
26   {
27
28     for (unsigned int i = 0; i < numberElements; i++) {
29       O[i] = 0.0;
30       A[i] = B[i] = 25;
31     }
32   }
33
34   static string usage(string program) {
35     return program + " number_elements " + Benchmark::usage();
36   }
37
38 protected:
39 //  void setup_skeleton_tree() {
40 void init() {
41     skeleton_tree = make_map(
42         kernel<int*, const int, const int> (
43         resolveKernel(kernelFile), kernelFunction),
44         Args<Vector<int>&, Vector<int>&, Vector<int>& > {} );
45
46   }
47
48   void run() {
49     marrow::future future = skeleton_tree->run(O, A, B);
50     future.get();
51   }
52
53   bool validate() {
54     auto reference = (A[0]+B[0]);
55     for (unsigned int i = 0; i < numberElements; i++)
56       if (O[i] != reference)
57         return false;
58     return true;
59   }
60 };
61
62 int main(int argc, char* argv[]) {
63   if (argc < 2) {
64     cerr << ThreeLevels::usage(string (argv[0])) << endl;
65     return EXIT_FAILURE;
66   }
67
68   int n = atoi(argv[1]);
```

```
69   if (n <= 0) {
70     cerr << "Wrong value for \"number_elements\"." << endl <<
71         ThreeLevels::usage(string (argv[0])) << endl;
72     return EXIT_FAILURE;
73   }
74
75   try {
76     ThreeLevels s(n, argc, argv);
77     s.timedRuns();
78   } catch (std::runtime_error& e) {
79     cerr << "Error : " << e.what() << std::endl << std::endl;
80     return EXIT_FAILURE;
81   }
82
83   return 0;
84 }
```

7.6 Marrow expression twolevels implementation

Listing 7.9: Listings/marrow-expressions/twolevels.cpp

```
1  //
2  // Created by fmmarques on 21/03/16.
3  //
4
5  #include <iostream>
6
7  #include <marrow.h>
8  #include "marrow/utils/Benchmark.hpp"
9
10
11 using namespace std;
12 using namespace marrow;
13
14 static const int inputValue = 10;
15
16
17 class three_levels: public marrow::Benchmark {
18     unsigned int numberElements;
19
20     Vector<float> o, a, b;
21
22 public:
23     three_levels(int nrElements, int argc, char* argv[]) :
24             marrow::Benchmark("jit_threelevels", argc-1, &argv[1]),
25             numberElements (nrElements),
26             o (numberElements),
27             a (numberElements),
28             b (numberElements) {
29         a.fill([](int) { return inputValue; });
```

```
30            b.fill([](int) { return inputValue; });
31        }
32
33
34        void init() {}
35        void run() {
36            o = a+b ;
37            auto first = o[0];
38        }
39
40        bool validate() {
41            const float referenceValue = a[0]+b[0];
42
43            for (unsigned int i = 0; i < numberElements; i++)
44                if (o[i] !=  referenceValue)
45                    return false;
46
47            return true;
48        }
49    };
50
51    int main(int argc, char* argv[]) {
52        if (argc < 2) {
53            cerr << argv[0] << " <element_count>" << endl;
54            return EXIT_FAILURE;
55        }
56
57        int n = atoi(argv[1]);
58        if (n <= 0) {
59            cerr << "Invalid vector size." << endl
60            << argv[0] << " <element_count>" << endl;
61            return EXIT_FAILURE;
62        }
63
64
65        try {
66            three_levels s(n, argc, argv);
67            s.timedRuns();
68        } catch (std::runtime_error& e) {
69            cerr << "Error : " << e.what() << std::endl << std::endl;
70            return EXIT_FAILURE;
71        }
72
73        return 0;
74    }
```

7.7 Marrow threelevels implementation

Listing 7.10: Listings/marrow/threelevels.cl

```
__kernel void three_levels(
   __global float* o,
   __global float* a,
   __global float* b,
   __global float* c,
   __global float* d ) {
   int i = get_global_id(0);
   o[i] = (a[i]+b[i])+(c[i]+d[i]);
}
```

Listing 7.11: Listings/marrow/threelevels.cpp

```
#include "marrow.h"
#include "marrow/utils/Benchmark.hpp"

using namespace marrow;
using namespace std;

const string kernelFile = "three_levels.cl";
const string kernelFunction = "three_levels";

class ThreeLevels: public marrow::Benchmark {
private:
  unsigned int numberElements;
  Vector<int> O;
  Vector<int> A, B;
  Vector<int> C, D;

  std::unique_ptr<
        Map<KernelWrapper<int*, const int, const int, const int, const int
            >,
            Vector<int>&, Vector<int>&, Vector<int>&, Vector<int>&, Vector<
                int>&>> skeleton_tree;

public:
  ThreeLevels(int nrElements, int argc, char* argv[]) :
      marrow::Benchmark("marrow_four", argc-1, &argv[1]),
      numberElements (nrElements),
      O (numberElements),
      A (numberElements), B (numberElements),
      C (numberElements), D (numberElements)
  {

    for (unsigned int i = 0; i < numberElements; i++) {
      O[i] = 0.0;
      A[i] = B[i] =
      C[i] = D[i] = 25;
    }
```

```
35    }
36
37    static string usage(string program) {
38      return program + " number_elements " + Benchmark::usage();
39    }
40
41  protected:
42  //  void setup_skeleton_tree() {
43  void init() {
44      skeleton_tree = make_map(
45          kernel<int*, const int, const int, const int, const int> (
46          resolveKernel(kernelFile), kernelFunction),
47          Args<Vector<int>&, Vector<int>&, Vector<int>&, Vector<int>&, Vector
              <int>& > {} );
48
49    }
50
51    void run() {
52      marrow::future future = skeleton_tree->run(O, A, B, C, D);
53      future.get();
54    }
55
56    bool validate() {
57      auto reference = (A[0]+B[0])+(C[0]+D[0]);
58      for (unsigned int i = 0; i < numberElements; i++)
59        if (O[i] != reference)
60          return false;
61      return true;
62    }
63  };
64
65  int main(int argc, char* argv[]) {
66    if (argc < 2) {
67      cerr << ThreeLevels::usage(string (argv[0])) << endl;
68      return EXIT_FAILURE;
69    }
70
71    int n = atoi(argv[1]);
72    if (n <= 0) {
73      cerr << "Wrong value for \"number_elements\"." << endl <<
74          ThreeLevels::usage(string (argv[0])) << endl;
75      return EXIT_FAILURE;
76    }
77
78    try {
79      ThreeLevels s(n, argc, argv);
80      s.timedRuns();
81    } catch (std::runtime_error& e) {
82      cerr << "Error : " << e.what() << std::endl << std::endl;
83      return EXIT_FAILURE;
```

```
84     }
85
86     return 0;
87 }
```

7.8 Marrow expression threelevels implementation

Listing 7.12: Listings/marrow-expressions/threelevels.cpp

```
1  //
2  // Created by fmmarques on 21/03/16.
3  //
4
5  #include <iostream>
6
7  #include <marrow.h>
8  #include "marrow/utils/Benchmark.hpp"
9
10
11 using namespace std;
12 using namespace marrow;
13
14 static const int inputValue = 10;
15
16
17 class three_levels: public marrow::Benchmark {
18     unsigned int numberElements;
19
20     Vector<float> o, a, b, c, d;
21
22 public:
23     three_levels(int nrElements, int argc, char* argv[]) :
24             marrow::Benchmark("jit_threelevels", argc-1, &argv[1]),
25             numberElements (nrElements),
26             a (numberElements),
27             b (numberElements),
28             c (numberElements),
29             d (numberElements) {
30         a.fill([](int) { return inputValue; });
31         b.fill([](int) { return inputValue; });
32         c.fill([](int) { return inputValue; });
33         d.fill([](int) { return inputValue; });
34     }
35
36
37     void init() {}
38     void run() {
39         o = a+b+c+d ;
40         auto first = o[0];
41     }
```

```
42
43      bool validate() {
44          const float referenceValue = a[0]+b[0]+c[0]+d[0];
45
46          for (unsigned int i = 0; i < numberElements; i++)
47              if (o[i] !=  referenceValue)
48                  return false;
49
50          return true;
51      }
52  };
53
54  int main(int argc, char* argv[]) {
55      if (argc < 2) {
56          cerr << argv[0] << " <element_count>" << endl;
57          return EXIT_FAILURE;
58      }
59
60      int n = atoi(argv[1]);
61      if (n <= 0) {
62          cerr << "Invalid vector size." << endl
63          << argv[0] << " <element_count>" << endl;
64          return EXIT_FAILURE;
65      }
66
67
68      try {
69          three_levels s(n, argc, argv);
70          s.timedRuns();
71      } catch (std::runtime_error& e) {
72          cerr << "Error : " << e.what() << std::endl << std::endl;
73          return EXIT_FAILURE;
74      }
75
76      return 0;
77  }
```

2016 | Frederico Mariano Marques | High-level Programming of Many-Core Architectures