Hélder de Almeida Marques

Licenciado em Engenharia Informática

Towards an Algorithmic Skeleton Framework for Programming the Intel® Xeon Phi™ Processor

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

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

Júri:

Presidente:	Prof. Doutor Miguel Pessoa Monteiro Universidade Nova de Lisboa
Arguente:	Prof. Doutor Nuno Roma Universidade de Lisboa
Vogal:	Prof. Doutor Hervé Miguel Cordeiro Paulino Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2014

Towards an Algorithmic Skeleton Framework for Programming the Intel® Xeon Phi™ Processor

Aos meus pais e amigos que tornaram esta tese possível

Acknowledgements

I would like to start by thanking so much to my adviser Hervé Paulino for all the support, availability and constant meetings throughout the course of this thesis that really helped me a lot to develop this work. I would like to thank the department of Computer Sciences of Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa, for the facilities and its constant availability to always have a friendly environment to work. Also, thanks to the projects PTDC/EIA- EIA/113613/2009 (Synergy-VM) and PTDC/EEI-CTP/1837/2012 (SwiftComp) for financing the purchase of the Intel®Xeon Phi™, without which I could not certainly have carried out this project.

I want to thank professor Pedro Medeiros for the assistance with the problems that would appear with the machine used to develop the work done. I would like to specially thank all my colleagues that shared with me many hours on the same room, questioning why the code did not work, providing fun moments to relax and re-organizing ideas and giving support to never give up, namely, Catarina Gralha, Diogo Cordeiro, Gabriel Marcondes, Gonçalo Tavares, João Claro, Sara Gonçalves, Hélder Gregório. To the others that sometime passed by to cheer up, also thank you. I would like to thank my friends from Scouts that always gave me support to keep going, thank you Patrícia Correia, Sara Silva and João Silva.

I want to give a very big thank you to Joana Coelho for the constant support and motivation, and for always being there when things did not work so well.

Finally I want to thank my parents for all the support and for always believing in my capabilities.

Muito obrigado a todos!

Abstract

The Intel® Xeon Phi™ is the first processor based on Intel's MIC (**M**any **I**ntegrated **C**ores) architecture. It is a co-processor specially tailored for data-parallel computations, whose basic architectural design is similar to the ones of GPUs (**G**raphics **P**rocessing **U**nits), leveraging the use of many integrated low computational cores to perform parallel computations. The main novelty of the MIC architecture, relatively to GPUs, is its compatibility with the Intel x86 architecture. This enables the use of many of the tools commonly available for the parallel programming of x86-based architectures, which may lead to a smaller learning curve. However, programming the Xeon Phi still entails aspects intrinsic to accelerator-based computing, in general, and to the MIC architecture, in particular.

In this thesis we advocate the use of algorithmic skeletons for programming the Xeon Phi. Algorithmic skeletons abstract the complexity inherent to parallel programming, hiding details such as resource management, parallel decomposition, inter-execution flow communication, thus removing these concerns from the programmer's mind. In this context, the goal of the thesis is to lay the foundations for the development of a simple but powerful and efficient skeleton framework for the programming of the Xeon Phi processor. For this purpose we build upon Marrow, an existing framework for the orchestration of OpenCL™ computations in multi-GPU and CPU environments. We extend Marrow to execute both OpenCL and C++ parallel computations on the Xeon Phi.

We evaluate the newly developed framework, several well-known benchmarks, like Saxpy and N-Body, will be used to compare, not only its performance to the existing framework when executing on the co-processor, but also to assess the performance on the Xeon Phi versus a multi-GPU environment.

Keywords: Many Integrated Cores architectures, Intel® Xeon Phi™, Parallel Programming, Algorithmic Skeletons

Resumo

O Intel® Xeon Phi™ é o primeiro processador baseado na arquitectura *Many Integrated Cores* da Intel. O seu propósito é ser usado como um co-processador para a unidade de processamento central para as computações de dados em paralelo. Os pricípios básicos de desenho arquitectual são semelhantes aos usados nas unidades de processamento gráfico (GPU), tirando proveito do uso de muitos núcleos de poder computacional menor para efectuar computações paralelas. A grande novidade da arquitectura MIC, relativamente aos GPUs, é a sua compatibilidade com a arquitectura x86, o que permite o uso de muitas das ferramentas mais comuns utilizadas em computação paralela, o que pode levar a uma menor curva de aprendizagem. No entanto, o uso do Xeon Phi em aplicações comuns ainda requer o conhecimento geral de programação baseada em aceleradores e, em particular, da arquitectura MIC.

Nesta tese, advogamos o uso de esqueletos algorítmicos para a programação do Xeon Phi. Esqueletos algorítmicos abstraem a complexidade inerente à programação paralela, escondendo detalhes como a gestão de recursos, decomposição paralela, ou a comunicação com a camada subjacente, removendo assim estas preocupações da mente do programador. Neste contexto, o nosso objectivo é oferecer as bases para o desenvolvimento de uma ferramenta simples mas poderosa e eficiente, para a programação do Xeon Phi. Como este propósito vamos trabalhar sobre o Marrow, uma ferramenta existente para a orquestração de computações OpenCL™ em ambientes CPU e multi-GPU. Nós adaptamos o Marrow para executar computações paralelas tanto em OpenCL como em C++.

A nova versão é avaliada, usando vários testes bem conhecidos como o Saxpy ou on N-Body, para comparar, não só o seu desempenho com a actual versão do Marrow quando executada no co-processador, mas também para medir o desempenho num ambiente com multi-GPUs e Xeon Phi.

Palavras-chave: Arquitecturas *Many-Core*, Intel® Xeon Phi™, programação paralela, esqueletos algorítmicos

Contents

List of Figures

Listings

1

Introduction

1.1 Motivation

Over the years, the magnitude of the problems addressed with the help of computational resources has consistently grown at a exponential rate. This is true for both the scientific computing and the information technology fields. An example of the former is the experiments that take place at CERN's Large Hadron Collider[1]. At peak rates, about 10 gigabytes of data are generated every second. This amount of data needs to be processed quickly to assess its importance and determine if it is worth storing and so the data must be processed in parallel in order to increase the throughput of the algorithms. On the information technology end, the generalized use of the World Wide Web as a vehicle to transmit both public and personal information. For instance, the amount of data generated by social networks grow exponentially over time. As a result, there are many companies, such as Google, that are inferring information from this unstructured data[2]. For that purpose, they need to find efficient ways to process all that information.

In this context, accelerator-based computing in general, and GPU computing in particular, is establishing itself as an consolidated field. The massive adoption of this model is evident in the list of the current top 500 supercomputers, where the two fastest ones, by June 2014, feature accelerators in their configuration[3]. Recent proposals explore the use of many-core processors as dedicated co-processors to provide a tool with a great computational power but also, power efficient. Examples are AMD's APUs [AMD] and Intel®'s Xeon Phi™ [Int]. In this dissertation we are particularly interested in the latter.

[1] http://home.web.cern.ch/about/computing
[2] http://www.ams.org/samplings/feature-column/fcarc-pagerank
[3] http://www.top500.org/lists/2014/06/

Xeon Phi is the first product based on the MIC (**M**any **I**ntegrated **C**ore) architecture [Sei+08], being announced in November 2012. This co-processor features a large number of cores (starting at 57 cores) and is attached to the host computer via a PCIe interface. Its main advantages are the high degree of parallelism that offers and, opposing to GPUs, its compatibility with the x86 architecture, which means that existing tools for x86-based systems are supported, moving away from the low level programming required by GPUs. Additionally, it is also more power efficient [Abd+13] since only the cores actually being used consume significant amounts of power. Performance-wise, the efficiency of this product was demonstrated by the first place in the list of supercomputers offering most FLOPS (**Fl**oating-point **O**perations **P**er **S**econd) per Watt by the time of its launch[4]. The promise of the co-processor is evinced by its use on the fastest supercomputer by June 2014.

In sum, Xeon Phi embodies the new kind of general-purpose accelerators, which seems to be a first step towards a widespread adoption of accelerator-based computing.

1.2 Problem

Xeon Phi is the first commercial product based on the MIC architecture, and so, due to its tender age, there are still many challenges to address when trying to efficiently program this kind of architectures. Due to its compatibility with x86, many of the existing tools, like OpenMP [Ope13b], Cilk™ [Int13b], OpenCL™ [Khr13], MPI [For12] or Intel TBB [Int13c], can be directly used in the programming of Xeon Phi. Nevertheless, the programmer is still faced with many details intrinsic to accelerator-based computing, such as:

- The explicit management of multiple disjoint address spaces;
- The orchestration of data transfers and computation;
- The overhead of the communication between the host device and the accelerator.

Beyond that, there is the need to understand the implications that the particular nature of the Xeon Phi architecture can have on the application, specially because of its high level of parallelism, due to the 240 threads available. This understanding is important to take advantage of the full potential of the co-processor and to avoid the under utilization of the resources available [Joh+13; Sch+13]. In fact, although Xeon Phi grows from x86 cores, there is some debate concerning the ease of use of this co-processor. Some opinions are in favor of GPUs stating that, in order to fully use the co-processor to its maximum, the effort needed in programming is the same as the one needed in programming a GPU, but the results are much better on a GPU [Hem13; Nvi]. To this extent, raising the degree of abstraction in the programming of this kind of architectures is a relevant research topic.

[4] http://www.green500.org/lists/green201211

The Xeon Phi processor has become available to the general community very recently, thus only now the community is gathering efforts to address the aforementioned challenges. Examples are the porting of MapReduce [Lu+13], parallel libraries [Cla+13; Dok+13; Pot+] and compilers [PM12].

Research groups with experience on the GPGPU (**G**eneral-**P**urpose computing on **G**raphics **P**rocessing **U**nits) field are directing the knowledge gathered in the accelerator computing field to the Xeon Phi, extending existing frameworks to tackle the challenges associated with this platform [Lim+13; Men+13]. The approach of the work to be developed in the context of this dissertation takes a similar approach. We will build on previous knowledge acquired in the development of an algorithmic framework for general-purpose computing on GPUs, the Marrow framework [AMP13; AMP14; Mar+13; SAP14], and apply it in the construction of a similar framework for the programming of the Xeon Phi.

In what concern structured parallel programming, to the best of our knowledge, the adaptation of MapReduce to Xeon Phi proposed in [Lu+13] is the only work to port known algorithmic skeletons to Xeon Phi. As such, Marrow's features, namely a library of multiple skeletons that can be combined to build complex, compound, behaviors, are new to the field.

The challenges that we will face are:

- to subsume all communication with the device, and between the multiple concurrent computations that be encased in an skeleton, for instance in a pipeline;
- to transparently and efficiently manage the orchestration of the computation to be offloaded to the device, this includes understanding when to move the data, doing it efficiently over the PCIe bus and when to reuse data already residing on the device's memory;
- to efficiently execute this computation on the device. This includes paralelizing the code given by the programmer, how many threads to spawn, where to place them and how to exploit vectorization.

1.3 Proposed Solution

The Marrow framework is a library that provides algorithmic skeletons to orchestrate compound OpenCL computations in heterogeneous environments with multiple CPUs and GPUs. This framework steps away from other GPU programming frameworks by offering both task- and data-parallel skeletons and by supporting skeleton nesting in this type of environments. Currently Marrow supports the Map and MapReduce data-parallel skeletons and the Pipeline and Loop task-parallel skeletons.

The goal of this thesis is to leverage Marrow's structured programming model in the programming of the Xeon Phi processor. For that purpose, we intend to initially develop

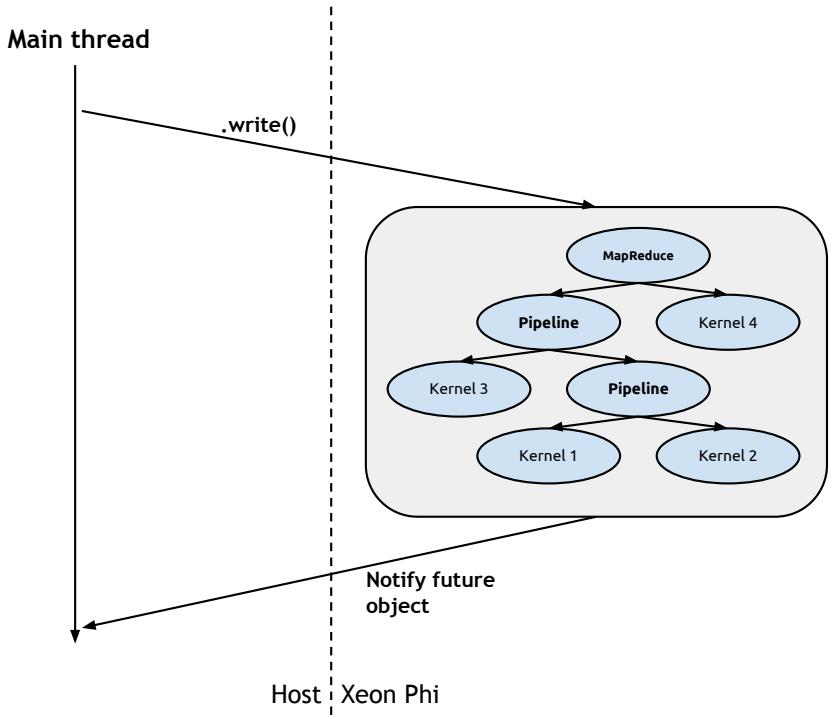


Figure 1.1: Offloading the computation tree to the Xeon Phi

a Xeon-specific OpenCL back-end, so that the framework can efficiently execute trees of nested OpenCL applications on this device, as shown in Figure 1.1.

On a second phase, we want to adapt the framework with the purpose of supporting different technologies, besides OpenCL. For that, the framework will be generalized to hide all mechanisms intrinsic to OpenCL and the runtime will be extend with a new back-end to support the execution of kernels expressed as C++ functions, expressing the sequential behavior that each thread running on the Xeon Phi must execute. The parallel execution of such code is managed by the framework.

In sum, the goal of this thesis is to offer both task- and data-parallel skeletons to execute efficiently on the Intel Xeon Phi, and therefore, in heterogeneous environments with CPU, GPU and Xeon Phi.

1.4 Contributions

The presented work will provide the following contributions:

1. A comparative study about the performance of multiple technologies used for programming the Xeon Phi (OpenMP, Cilk Plus and OpenCL);

2. An OpenCL back-end to add support for the Xeon Phi to the Marrow framework and the corresponding performance evaluation against GPU and CPU environments;

3. A new runtime layer for the Marrow framework to support the execution of kernels expressed as C++ functions, leaving for the framework the task of executing them in parallel;

4. The evaluation of the performance of the new runtime layer relatively to the previous version by measuring the execution time of different benchmarks when executing on the Xeon Phi, using both the OpenCL and the chosen technologies.

Part of the work developed in the scope of this thesis was presented via a poster communication in INForum 2014, the 6th edition of the portuguese conference in computer science, that took place at Universidade do Porto.

1.5 Document Structure

This document has the following structure:

Chapter 2 starts with state of the art by the time prior to the work presented on this thesis. It starts by the architecture of the co-processor, highlighting the important features to consider when using this device, and then, it details the models used in the programming of the co-processor as well as some tools available to achieve those models.

Chapter 3 provides an overall description of the Marrow framework, the starting point for this thesis. It presents the framework's programming and execution models, as well as its architecture.

Chapter 4 presents the work developed in the scope of this thesis, namely the implementation of the Xeon Phi OpenCL Marrow back-end, the full support for the execution of C++ instantiated Marrow skeletons on the Xeon Phi, and the changes performed on the Marrow framework in order to efficiently and seamlessly support both the OpenCL and C++ versions.

Chapter 5 shows the tests performed to evaluate the performance of the newly developed framework and how it compares to the previous version. The tests were divided into two stages, the comparison between the Xeon Phi and GPUs using OpenCL and the comparison of the OpenCL and C++ versions running exclusively on the Xeon Phi.

Chapter 6 pose the conclusions derived from the work performed and discusses open issues that may be carried out in future work.

2

The Intel® Xeon Phi™ Processor and its Programming

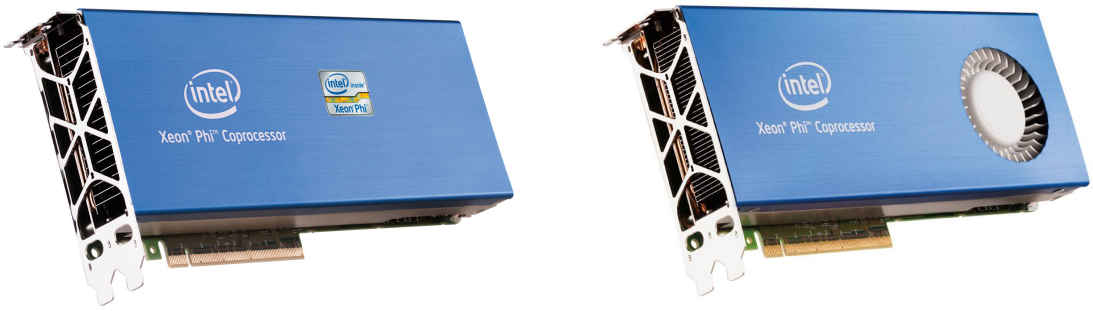
The Intel Xeon Phi co-processor is the first commercial product to use Intel MIC architecture, based on the Larrabee [Sei+08] microarchitecture. In order to introduce the reader to its key features, we present a general overview of its architecture in Section 2.1 and of programming tools offered by Intel for its programming in Section 2.2. Furthermore, we present an initial performance comparison of some of these tools in Section 2.3, to guide us in the choice of which technologies to use in our work. The research presented in this chapter was supported by the following literature: [JR13], [Col13] and [Rah13].

2.1 Intel® Xeon Phi™'s Architecture

The Xeon Phi is available in two form factors, shown in Figure 2.1, with passive or active cooling. The co-processor features multiple Intel x86 cores, up to eight memory controllers and a PCI Express system interface to communicate with the host device. Although based on the Intel ia-64 architecture, the cores featured in a Xeon Phi processor are simpler versions of the ones currently featured in the current Intel-compatible multi-core CPUs. Consequently, they are smaller in size, a fact that contributes for more of them to be packed in a single processor. Given that all these cores are identical, the Xeon Phi falls into the category of *homogeneous many-core architecture*.

We begin by giving a general overview of the Xeon Phi's architecture, detailing subsequently its main components.

The key components of a Xeon Phi chip are:



(a) Model with passive cooling (b) Model with active cooling

Figure 2.1: Available models of the Xeon Phi co-processor

Co-processor cores These are based on P54c (Intel Pentium from 1995) cores with major modifications including Intel 64 ISA, 4-way SMT, new vector instructions, and increased cache sizes.

Vector Processing Unit The VPUs are part of the core and capable of performing 512-bit vector operations on 16 single-precision or 8 double-precision floating-point arithmetic operations as well as integer operations.

Cache All cores have full coherent L2 caches.

Tag directories (TD) Components used to look up cache data distributed among the cores and keep coherency.

Ring interconnect The interconnect between the cores and the rest of the co-processor's components.

Memory controller (MC) Interface between the ring and the graphics double data rate (GDDR5) memory.

PCIe interface Used to connect with PCIe bus and the host.

The topology used to interconnect all the cores in the system is a bidirectional interconnect ring (Figure 2.2). This bus is also used by the cores to access data from the main memory through the memory controllers also connected to the ring. There are eight memory controllers with two channels, each communicating with GDDR5 memory at 5.5 GT/s (Giga transfers per second) having a theoretical memory bandwidth of 352 GB/s. The movement of data between the host's memory and the Intel Xeon Phi's memory is made through DMA (**D**irect **M**emory **A**ccess) enabling the independence from the host intervention in some cases. One key requirement to allow the usage of the co-processor is the BIOS support for memory mapped I/O address ranges above 4GB.

To manage all the hardware, one of the cores executes a micro-OS, based on Linux, that is read at boot-time from an on-board flash.

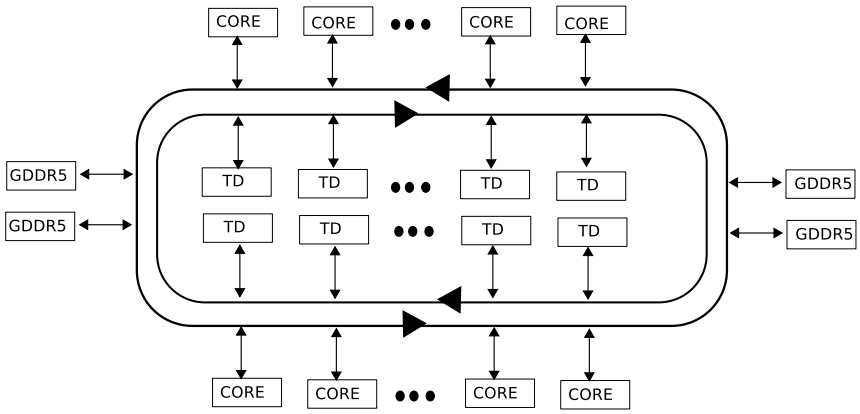


Figure 2.2: Diagram of the Xeon Phi's architecture [GH13]

With respect to reliability features, there is parity support in the L1 cache, error correction code (ECC) on L2 and memory transactions, cyclic redundancy code (CRC) and command and address parity on the memory I/O. The co-processors also implement extended machine check (MCA) features to help the software detect imminent failure.

2.1.1 Cores

Each core is an in-order processor that features four hardware threads that are used to mask the latencies inherent to in-order micro-architectures. The choice of this type of processor relates to the fact that it is much simpler and more power efficient, as opposed to the highly speculative out-of-order type, widely used on Intel Xeon processors. The multi-threading is done by using the time-multiplexing mechanism.

Inside the core, there are also the L1 and L2 caches, shared by the hardware threads, along with an L2 cache hardware prefetcher (HWP) that can detect sixteen, forward or backward, sequential data accesses to determine the direction of the stream and then issue multiple prefetch requests to keep the data flow. Figure 2.3 shows a simple overview of each core's micro-architecture.

Instruction execution within a core is structured in only 5 pipeline stages, unlike most modern processors, which makes the execution of an instruction to be much faster therefore improving the overall performance of the system. The architecture used is a *superscalar architecture*, with two different pipelines on the chip, known as the U and V pipelines, enabling that two instructions are at the execution stage at the same clock cycle. Integer operations and mask instructions are single-cycle. Most vector instructions have a four-cycle latency and single-cycle throughput, so the concurrent execution of four threads will not reveal the vector unit latency if all of them are executing vector instructions.

There is also a turbo mode that, if the available power allows, increases the frequency of the cores in order to boost performance.

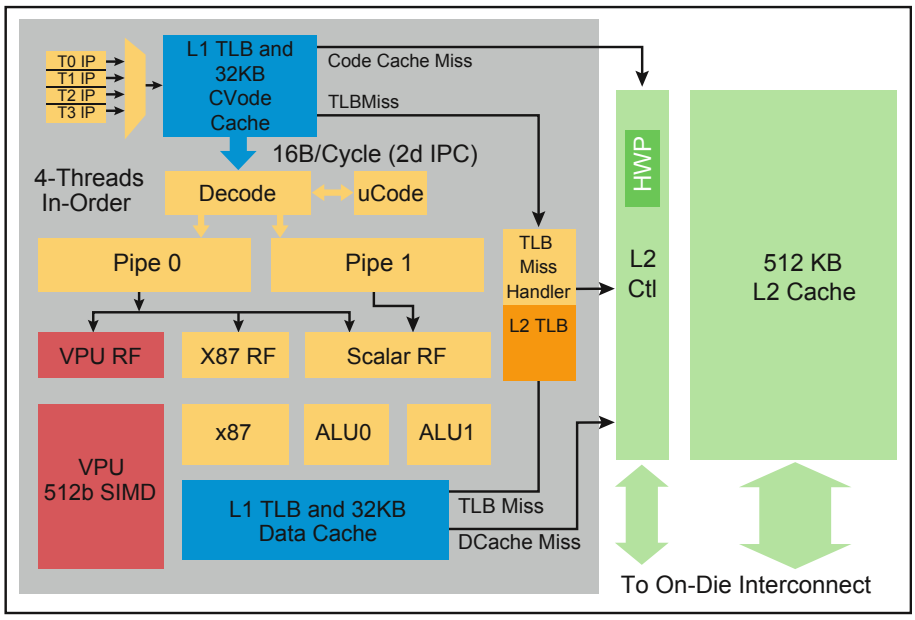


Figure 2.3: Diagram of the cores' micro-architecture [JR13]

Vector Processing Unit

The vector processing unit (VPU) used is 512 bit wide. This means that it can operate over 16 single-precision floats or 32-bit integers at a time. It implements a novel ISA (**I**nstruction **S**et **A**rchitecture), with 218 new instructions compared with those implemented in the Xeon family of SIMD instruction sets. The instructions are received from the core's ALU (**A**rithmetic **L**ogic **U**nit) and the data is received from the L1 cache by a dedicated 512-bit bus. Most of the VPU instructions are issued from the core through the U-pipe. Some of the instructions can be issued from the V-pipe and can be paired to be executed at the same time with instructions in the U-pipe. These instructions are computed using quadratic minimax polynomial approximation and use a lookup table to provide a fast approximation to the transcendental functions. The vector architecture supports a coherent memory model in which the Intel-64 instructions and the vector instructions operate on the same address space. The instructions support the following native data types:

- Packed 32-bit integers (or dword)
- Packed 32-bit single-precision FP values
- Packed 64-bit integers (or qword)
- Packed 64-bit double-precision FP values

The ISA supports the proposed standard IEEE 754-2008 floating-point instruction rounding mode requirements. Each VPU consists of eight master ALUs, each containing two SP (**S**ingle-**P**recision) and one DP (**D**ouble-**P**recision) ALU with independent pipelines, thus allowing sixteen SP and eight DP vector operations.

2.1.2 Cache

Many changes were made to the original 32-bit P54c architecture to make it into an Intel Xeon Phi 64-bit processor. The data cache was modified to non-blocking by implementing thread-specific flush, which means that, when there is a cache miss, only the pipeline of the thread that caused the miss will be flushed, preventing the other threads from also being blocked. When the data is available to the thread that had a cache miss, that thread is then woken up.

The L1 cache consists of 8-way set-associative 32 kB L1 instruction and data caches, separately, having an access time of approximately 3 cycles. The L2 cache is also 8-way set-associative and 512 kB in size. Although, unlike the L1 caches, this one is unified, in the sense that it is used to cache both data and instructions. The latency of these caches can be as small as 14-15 cycles.

The L1 cache is inclusive to the L2 cache. It can deliver 64 bytes of read data to corresponding cores every two cycles and 64 bytes of write data every cycle. Both caches use pseudo-LRU implementation as a replacement algorithm.

To maintain the coherence of all L2 caches a modified MESI [PP84] protocol using a TD-based GOLS (**G**lobally **O**wned, **L**ocally **S**hared) protocol is used.

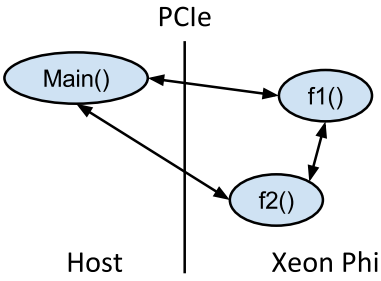
2.1.3 PCIe

The system interface of the chip supports PCI Express 2.0 interface x16 lanes with 64- to 256-byte packets, peer-to-peer read/writes. The peer-to-peer interface allows two Intel Xeon Phi cards to communicate with each other without host intervention. The clock system uses a PCI Express 100 MHz reference clock and includes on-board 100 MHz $\pm$ 50 ppm reference.

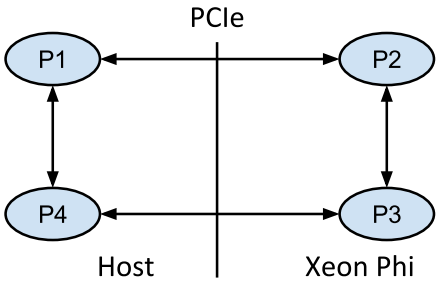
2.2 Programming the Intel® Xeon Phi™

Due to its high number of logical cores, Xeon Phi is specially tailored for HPC (**H**igh **P**erformance **C**omputing) applications. It requires the installation, on the host, of the drivers, also known as MPSS (**M**any-Core **P**latform **S**oftware **S**tack package), making it visible to the host as a co-processor connected via a network interface. The usage of the co-processor may be summarized in the three execution models depicted in Figure 2.4: the *native model*, the *offload model* and the *symmetric model*.

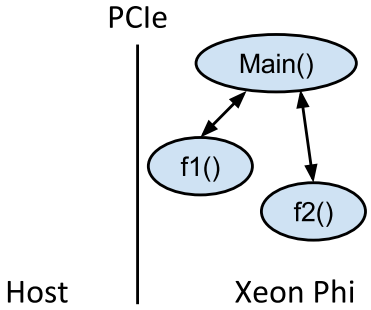
The native model consists in running the main application directly on the co-processor. This is possible due to the micro OS running on the co-processor, allowing it to be considered as a node in a cluster, therefore, being able to run applications independently. In this case, first, the code that is to be executed by the co-processor needs to be compiled for the co-processor's architecture and then, the generated binary must be sent to the co-processor using some network tools such as *secure copy* (scp). The application can then



(a) Offload model



(b) Symmetric model



(c) Native model

Figure 2.4: Xeon Phi's different execution models

be subsequently executed directly on the processor, requiring only some kind of remote shell, typically *secure shell* (ssh).

The symmetric model leverages the fact that the co-processor features a TCP/IP stack and thus may be viewed as a network node. As such, the application may be composed of multiple, independent, modules deployed at each end of the platform composed by the host and the Xeon Phi. The coordination of these modules is performed through some message-passing interface. In this model there is also the need to generate a binary specific for the co-processor and execute it like the native model.

Finally, in the offload execution model, the main application is executed on the hosting machine, offloading computations to the co-processor. These computations can be executed both synchronous or asynchronously with respect to the issuing thread. With this model, the application just has to annotate the regions that are to be offloaded, so that the compiler knows exactly which code to compile for the co-processor. After it has been compiled, the application is executed on the host, disregarding any additional effort. The parallel execution model associated with this execution model is the Fork-Join. This model of parallelism creates a new task (forks) when there is the need to do some new computation in parallel with the main thread and terminates it (joins) when the computation is done.

Because of the compatibility with the x86-architecture, some of the tools already used

for programming multi-core systems can be used to exploit the parallelism of the co-processor. Regarding the native mode, all parallel programming tools that have been, or may be, compiled to the Xeon phi architecture can be used, such as Intel Cilk™ Plus [Int13b; Lei09], OpenMP (Version 3.1) [Ope11] or Intel Threading Building Blocks [Int13c] libraries. These tools can also be used on the remaining models to program the code sections destined to run on the co-processor. As for the offload model, the tools available to send a computation to the co-processor are the language extensions provided by the Intel compiler, the OpenMP 4.0 [Ope13b] and OpenCL™ [Khr13] libraries. As previously mentioned, in order for the modules executing both on the host and the co-processor to communicate between each other, a library like MPI [For12] can be used.

The next sections will detail some of the tools aforementioned to understand the effort required to use them, and, for each one, some historical context is given as well as its basics and some of the most usual patterns of how they are used are explained. Some examples on how they can be used are presented in Appendix A.

2.2.1 Native model

Given that the tools used in the native model can also be used for the remaining models, this section will provide a general overview of state of the art of programming tools for the Xeon Phi.

2.2.1.1 OpenMP

OpenMP (**Open M**ulti-**P**rocessing) is an API for the development of parallel applications in a wide range of processors architectures and operating systems. It was developed by the nonprofit technology consortium OpenMP Architecture Review Board (OpenMP ARB) in the year of 1997 for Fortran and in 1998 for $C/C++$. The programming model is based on a set of compiler directives (*pragmas*) and a library of routines available to the programmer to query and tune some aspects of the execution environment.

The specification is currently in its 4.0 version. However, only version 3.1 is widely supported by GNU's, Oracle's and LLVM's compilers. The compiler provided by Intel for generating code for the Xeon Phi architecture is one of tools that already include support for some of the OpenMP 4.0 features.

OpenMP provides constructs for parallel sections and loops, tune the scheduling mechanism, task creation, nested parallelism, operations to avoid race conditions and reducing the results of multiple threads to a single one.

The execution model used by OpenMP is based on the Fork-Join pattern. The sequential parts of the program are executed by a single thread whilst the parallel parts are executed by multiple threads. The upper limit for the number of threads can be tuned by changing the environment variable `OMP_NUM_THREADS` or by using the routine `omp_set_num_threads()`. As for the memory model, there are two types of memory: the thread private and the one shared by all threads.

OpenMP helps the implementation of two of the most common models of parallelism, the Loop-centric and the Fork-Join models [Col13].

For-loop parallelism in OpenMP A very large number of parallel applications are based on for-loops in which the same code is executed for the multiple pieces of data available. Such loops can be easily parallelized with OpenMP through the `#pragma omp parallel for`.

The `parallel for` directive may be complemented with several clauses, with the purpose of fine tuning parallelization parameters or strategies, such as scheduling model, variable privacy among threads or reduction of the intermediate results of each iteration.

Fork-Join parallelism in OpenMP Recursive problems that cannot be trivially transformed into for-loops may be coded as Fork-Join computations. These computations are based in tasks, that are assigned to the available threads, to be executed in parallel. The expressing of such tasks in OpenMP requires the use of `#pragma omp task` to denote the task's code: a function call or a block comprising multiple function calls.

The clauses associated to this pragma can be used to specify, like in the for-loop mechanism, the privacy of the variables but also the stop condition to prevent the runtime from spawning too much threads.

A second directive that can be used to express task parallelism is the `#pragma omp section`. All of this tasks must be inside a `#pragma omp sections` block so that the code that is after this block is only executed after all units of work complete.

A common practice in fork-join parallelism is to allow the main thread to also contribute in the execution of the spawned tasks and, with that, the main thread is not blocked waiting for all spawned functions but also is one less thread that needs to be spawned.

2.2.1.2 Intel® Cilk™ Plus

Cilk Plus is an open specification released by Intel that extends the linguistic and runtime technology for algorithmic multi-threaded programming named Cilk [Blu+95] with features for vector parallelism. Cilk was developed at MIT in 1994, based on ANSI-C with the addition of Cilk specific keywords. After acquiring the Cilk trademark in 2009, Intel released the commercial version, Cilk Plus, in 2010. The current version of this framework is 1.2.

This specification aims at addressing some shortcomings of template libraries, such as support for the C language, and not only *C++*, as well as for vector parallelism and finally that, because it is implemented at the compiler level, there are more opportunities for optimization. One important feature of this specification is that it guarantees that the serialized code, i.e. removing/replacing the Cilk keywords from the code, has the same results as the parallel code.

The model of execution is based only on work-stealing scheduling [BL99], which gives a good, out of the box, performance to Cilk programs but provides less tuning capabilities to the programmer.

For-loop parallelism in Cilk™ The ability to have parallel for-loops is also available with Cilk Plus. These for-loops are constructed by simply changing the keyword `for` with `cilk_for` but have some restrictions like the impossibility to use `break/return` inside the loop, the independence between loop iterations and, to make the calculation of the number of iterations possible, the creation of only one control variable as well as the inability to change it or the condition expression inside the loop. The use of `cilk_for` does not mean necessarily that the loop will be executed by multiple threads in parallel. Since the creation of threads has some overhead, the runtime system decides if the body of the loop pays off the creation of a new thread, if not, is the main thread that executes all iterations. It is also possible to do some basic computations with arrays, without the need to use for-loops, by using the array annotations and the reducers available.

The array annotations are used to perform simple operations over multiple arrays with the same range. The operations can also include scalars which are converted to arrays of replicated copies of the scalar with the same length as the range specified with the annotation. The full version of specifying a section of an array is by indicating the index at which the section begins, the length of the section and the step between consecutive indexes (`[`*`first`*`:`*`length`*`:`*`step`*`]`). If the step is omitted then the default step considered is 1. There is a short version to specify that the section is composed of all the elements of the array, omitting all arguments (`[:]`). Most scalar operations can be used in array sections, including the assign operator. The operation is applied element-wise to all elements of the sections and the result has the same length as the involved sections, which must all have the same length.

To use a reducer function there is only the need to call it with a section as argument and the return is the corresponding reduction. As for the reducer variables, the variable is defined using the constructor of the reducer operation that is needed, specifying the type of the variable and, optionally, indicating the initial value. The variable can only be affected inside a `cilk_for` loop and using the corresponding operations of the reducer. Outside the loop the variable can only be modified or read using the `set_value` and `get_value` functions, respectively. The various reducers include addiction/subtraction, minimum/maximum, binary operations and string concatenation.

Regarding parallel for-loops there are still the `#pragma simd` and elemental functions. Like `cilk_for`, the use of `#pragma simd` acts only has a hint to the compiler to execute the loop with vectorization. The vectorization is performed in small chunks of data whose size will depend on the vector width of the machine making the code portable as opposed to manual vectorization.

The elemental functions are functions that are annotated so that the compiler can compile multiple versions of the function optimized to evaluate multiple iterations in parallel. The annotation is `__declspec(vector)` with some optional clauses, `uniform(`*a*`)` and `linear(`*a*`:`*k*`)`. The `uniform` clause is used when an argument is constant between multiple iterations and the `linear` clause states that the argument *a* will step by *k* in consecutive calls to the function. If *k* is omitted then the default step is 1. A function can be marked as elemental even if it is defined in a different module. This mechanism of annotating functions is very important in situations where the functions are complex and the compiler cannot analyze them properly in order to make optimization.

Fork-Join parallelism in Cilk™ The original programming model for which Cilk was developed is the Fork-Join parallelism since the first specification only featured the definition of the `cilk_spawn` and `cilk_sync` keywords. The execution model is based on work-stealing schedule so the loop parallelism is based on spawning multiple threads and then create a queue from where the threads steal work to do.

This model is very simple to use in Cilk, to execute a function by a different thread in parallel, simply precede the function call with the keyword `cilk_spawn`. This can be used not only in functions that return `void` but also on an assignment to some variable. The assignment occurs when the function returns so, after spawning the function, if the caller uses the variable, it implicitly waits for the result to be available. The `cilk_sync` keyword is used to explicitly make the thread wait for all spawned functions inside the current block (function, `try/catch` or `cilk_for`). An implicit sync is made at the end of each spawning block, only waiting for functions spawned within the block.

2.2.1.3 Intel® TBB

Intel® TBB (**T**hreading **B**uilding **B**locks) was developed by Intel in 2006, as a proprietary library but becoming open-source in version 2.0 including a GPL v2 a year later, and it is a library that provides a set of solution for programming with tasks in *C++*.

The approach of TBB is to deal with tasks instead of threads and let the runtime system assign the tasks to the available threads. The library consists in a collection of generic parallel algorithms (`parallel_for` or `parallel_pipeline`), synchronization primitives (atomic operations and mutual exclusion), concurrent containers (`concurrent_hash_map` or `concurrent_queue`), scalable memory allocation (`scalable_malloc` and `scalable_free`) and a task scheduler to manage the creation and activation of tasks. The underlying scheduling system is, like Cilk Plus, based on a work-stealing strategy guaranteeing that the load is fairly distributed among all threads.

The use of an atomic variable to store the result guarantees that there are no data races when multiple threads try to update the variable. Beyond the usual operators to assign a variable, atomic variables also provide the `fetch_and_add`, `fetch_and_store` and `compare_and_swap` methods to read and update, atomically, the value of the variable.

TBB 4.0

The 4.0 version of this library added the flow graph feature. This feature enables the creation of a dependency graph to specify the computation and the flow of messages between the nodes.

The mostly common used nodes are the `broadcast_node`, the `function_node` and the `join_node`. A `broadcast_node` simply forwards every incoming message to all its successors. The `function_node` is used when a function must be applied to the incoming messages so the type of the incoming and outgoing messages must match the type of the function. In addition to the function, the level of concurrency must be specified to state the maximum number of parallel calls that can be made. In case the function must be called serially the special value `serial` can be used and, on the other hand, if the function can be called concurrently as many times as wanted the value `unlimited` can be used. When a node receives multiple values at once from its predecessors a `join_node` must be used to combine the values from all of its inputs and broadcast to all successors a tuple with the values received. The buffering policy is `queueing` so that a tuple is only emitted when there are values available in every incoming port.

After creating the nodes, the edges between them are defined using the `make_edge` routine. To start the computation and feed the nodes with values the method `try_put` is invoked and then the method `wait_for_all` of the graph is called to wait for all nodes to finish their computations.

Summary

Regarding the ability to program the devices, the existing technologies for shared memory programming on multi-cores may be directly used, namely OpenMP, Cilk Plus and Intel TBB. One advantage of using these technologies is that, because the co-processor is composed by x86 cores, there are no special concerns when using them, unless some fine tuning is desired, then requiring the understanding of the underlying architecture.

As mentioned before, since version 3.1 is the most widely supported version, the native and symmetric models can be used to develop, more easily, parallel applications but there is the overhead to compile the program, send it to the co-processor and then execute it there which does not give the transparency or usability of the offload model.

The Cilk framework has no construct available to use the Xeon Phi as an extra executing unit, disregarding the offload execution model, and taking advantage of the vector units in each core. To use the co-processor features, other tools must be used to offload some computation that has Cilk constructs, therefore taking advantage of the high number of cores and the vector units available in the co-processor. Since the computation is offloaded to the co-processor all data needed must be transferred using the available directives and clauses. Due to the simplicity of this framework, and its goal to parallelize a program with the least changes possible, the constructors available to spawn threads or to parallelize for-loops do not have any clauses. This prevents the possibility to perform

any optimization, making this framework the one with the least chance of tuning but providing good performance out of the box.

The Cilk programming model is tailored for task parallelism, while OpenMP is specially directed for data parallelism. Hence, to express fork-join computations in Cilk is much more simpler than in OpenMP.

Like some of the previous frameworks, TBB does not have any mechanism to use the co-processor as an auxiliary device. Because of this, the application must run in native mode on the co-processor or an offload construct from other framework must be used to send some computation that uses the TBB library to the co-processor.

2.2.2 Offload model

The previous section focused on tools that can be used to program the Xeon Phi just like a multi-core processor. This section will now focus on how both data and computation can be transferred between the host and the co-processor.

2.2.2.1 Intel® Compiler

The Intel compiler generates both offload and cross-compiled code for the Intel Xeon Phi. As expected, the offload mode is used to compile programs that offload computation to the co-processor, whilst the cross compile mode is for generating code for applications to be executed on native mode. This compiler is an augmented version of the traditional Intel compiler featuring new additions to enable the code offloading and compiler options to generate MIC-specific code, with the *-mmic* flag, and to perform code optimization.

Compiler Pragmas for offloading computation

The pragmas introduced to support the offload of computation were the `#pragma offload`, `#pragma offload_attribute`, `#pragma offload_transfer` and `#pragma offload_wait`.

The `#pragma offload` is used to offload a given computation to the co-processor. The code delimited by an offload block is compiled for both the host CPU and the Xeon Phi, so that the former can execute the block when there are no co-processors available.

This pragma is always followed by the clause `target` to specify the co-processor to which the computation is to be offloaded to in the form of `target(mic[:target_number])` where `target_number`, if $>= 0$, is used to select the n^{th} co-processor when multiple are available, being $n = target\_number$ **mod** $nr\_co\_processors$. If value -1 is assigned, then the selection process is delegated onto the runtime system, which will fail the execution of the block if no co-processor is available. When the target number is not specified, the runtime proceeds as if the value was -1 but, instead of the application failing, the computation is executed by the host processor in the absence of available co-processors.

When the code is compiled, there is the need to know which sections need to be specifically compiled for the co-processor so all global variables as well as user-defined functions that are used inside an offload block must be prefixed by `__attribute__(target(mic))`. To prevent the need to write the attribute for every variable and function, the `#pragma offload_attribute` directive can be used. There are some arguments for this directive, two to start the declaration block (`#pragma offload_attribute(push, target(mic))`) and one to end the block (`#pragma offload_attribute(pop)`). An `#include` can even be used between these pragmas to compile a whole module for the co-processor. It should be noted that only bitwise copyable variables flow between the host and the co-processor making C++ classes impossible to be sent to the co-processor. However, if a class is defined inside the pragmas, an instance of such class can be created and used inside the offload region.

When an computation is offloaded, all local and non pointer-based variables in the lexical scope of the block are sent to the co-processor before the beginning of the computation, and retrieved back to the host when the computation ends. The remainder variables must be explicitly copied through the four existing clauses, namely `in`, `out`, `inout` and `nocopy`. The first three state that the variables specified should be copied between the host and co-processor (to, from or both). The last one is used to allow the reuse of a previously copied variable without being sent back to/from the co-processor. By default, the space needed for the variables is allocated before the computation and is freed after the computation has ended. To enable the reuse of allocated space, the clauses to copy data have two arguments, the `alloc_if` and the `free_if`, to which is passed a boolean expression to state if the space is to be allocated/freed or not.

The `#pragma offload_transfer` directive was designed to permit the transfer of data between the host and the co-processor even when there is no computation to be executed. This directive also accepts the `target` and the data transfer clauses available to the `offload` directive.

The last directive, `#pragma offload_wait`, is used to wait for the completion of some computation or data transfer previously started. The asynchronous computation and/or data transfer occurs by using the `signal` clause in the `offload` and `offload_transfer` directives.

MYO (Virtual-Shared) Memory Model

Some of the drawbacks of using the offload model is the need to explicitly transfer data between the host and the co-processor, as well as the impossibility of transferring complex structures that are not bitwise copyable. To resolve these issues there is an alternative to the offload model, based in virtual shared memory, the MYO (**M**ine **Y**ours **O**urs) model. The idea behind this model is to delegate on the runtime system the job of keeping all variables shared between the host and the co-processor coherent. The synchronization happens at the beginning and at the end of offload statements and only

affects the changed variables. The keyword extensions available to use this model are: `_Cilk_shared` and `_Cilk_offload`.

`_Cilk_shared` may be applied to both variables and functions. Shared variables are assigned the same virtual address on the host as on the co-processor. They are allocated dynamically in the virtual shared space by the host, which means that, when the program is compiled, the code for the allocation of the shared variables is generated in the host's compiled code only. Because of that, if shared variables are defined only on the co-processor's code, by using an `#ifdef __MIC__` directive, their allocation code will not be generated, causing an runtime error when accessed. `_Cilk_shared` annotated functions are defined both on the host and the co-processor.

The allocation of shared memory must be made using the functions `_Offload_shared_malloc` and `_Offload_shared_aligned_malloc` and the respective de-allocation with the `_Offload_shared_free` and `_Offload_shared_aligned_free` functions. As for shared classes the placement version of the operator `new` must be used along with the `_Offload_shared_malloc` function.

Shared pointers to shared data, private pointers to shared data and shared pointers to private data are allowed with this memory model. This last case is used to have persistent data between offloads calls that do not need to be shared with the host. The allocation of the memory is made using the traditional `malloc` function and the address is stored in the shared variable so that distinct offload regions have access to the allocated space. This type of pointer must be used carefully because, since the memory was not allocated in the virtual shared memory an invalid de-referenciation can occur.

OpenMP 4.0

Up to version 3.1, the threads collaboratively executing an OpenMP computation had to share a common addressing space. This is guaranteed in machines with shared memory and can be emulated by software in a distributed environment, although the approaches proposed along the years never were widely adopted due to the overhead introduced being too high [BME07]. Beyond that, there was no support to express computations that should be executed by external devices, such as co-processors. Given the incremental use of co-processors in general purpose computing, the 4.0 version of the OpenMP specification includes built-in support, namely the `#pragma omp target`, that allows the computation to be offloaded to a device instead of being executed by the main processor. By the date of writing of this document, there is not support for GPUs yet. These external devices are often referred to as accelerators because of their optimized architecture to do some type of computation which can offer some improvement to the performance of a program. Since the accelerator is a device independent from the host, possibly having a different instruction set or even be programmable in a different language, and by having its own memory it is considered an heterogeneous device in relation to the host.

The changes made for this version were based on other API created to help develop parallel programs that offload computation to an accelerator, the OpenACC API [Ope13a; Wie+12]. The design of the OpenACC directive set always took into consideration a future integration in OpenMP. In fact, some of the people that worked on OpenACC's specification were members of the OpenMP ARB.

With the concept of accelerator, the notion of teams was added to aggregate multiple threads, with the `#pragma omp teams` directive. The approach enables a more complex distribution of work among all teams, and the threads that compose them, with the `#pragma omp distribute` directive. Also the memory model had to be changed to also take into account the accelerator's memory. To ease the mapping from the host's memory and the accelerator's memory, new clauses were added to the `#pragma omp target` directive: the `map` clauses. These clauses enable the programmer to specify if a host's variable is to be copied from/to the accelerator or just to allocate space in the accelerator for some variable.

In addition to the support for accelerators, the last version of OpenMP adds *SIMD* constructs, task enhancements, thread affinity (via environment variables), and user-defined reductions, through the `#pragma omp declare reduction`. The *SIMD* constructs added are the `#pragma omp simd` directive, that annotates a loop to be automatically vectorized, so that its execution makes use of *SIMD* lanes, the `#pragma omp declare simd` directive, to explicitly state that the target function can be called from a *SIMD* loop, and the `#pragma omp parallel for simd` directive, that marks a loop be executed by multiple threads, like the `#pragma omp parallel for`, but also to make use of the *SIMD* lanes.

The task enhancements made were the ability to stop an ongoing task or a group of tasks and possibility to have task dependencies between tasks.

2.2.2.2 OpenCL™

OpenCL (**Open C**omputing **L**anguage) is a specification developed in 2008, initially by Apple® Inc. and then taken by the Khronos Group, to be used in writing of parallel applications on an heterogeneous system consisting of CPUs, GPUs, FPGAs and other processors. The latest version of the specification is 2.1 but the version supported by Intel®'s compiler is 1.2. This framework is very focused in data parallelism since it was initially designed to work with the GPU architecture in mind, which allow the execution of the same instructions over multiple data at once.

The platform model in OpenCL consists of a host connected to multiples compute devices, or OpenCL devices, with their own memory, which in turn are composed of compute units that will execute processing elements. The processing elements are instances of kernels, defined in OpenCL C, that are applied to some portion of data. Each instance of a kernel is also known as work item and can be grouped into work groups. The set of all work items is known as NDRange (**N-D**imensional **Range**) and can have

up to three dimensions. An example of an one dimensional range is the vector addiction where work item i has to compute the sum of the elements in position i of the input arrays. A two dimensional range can be used in matrix multiplication where work item (i, j) computes the position (i, j) of the result. The value of the indexes can be obtained with the `get_global_id(`x`)` where x is the order of the index.

The memory objects available in OpenCL are only buffer and image objects. The buffers are one dimensional collections of scalars or user-defined structures and are accessed in kernels via pointers. The image objects are 2D or 3D textures, frame-buffers or images and can only be accessed through built-in functions. The kernels are implemented considering a local view of a thread, as opposed to the other frameworks that are based on a global view.

In addition to the memory objects, there are other entities that need to be created when developing an OpenCL application. The first one is a `cl_context`. This entity is used by the OpenCL runtime all the remaining entities. Next, to compile the program for the devices specified in the `cl_context`, the `cl_program` that contains the kernels to be used in the application must be created, providing its source code or the file containing it. After the `cl_program` is created, an entity for each kernel must be created, a `cl_kernel`, indicating the `cl_program` and the name of the kernel. Finally, all the commands for creating, writing and reading buffers or for executing a kernel must be issued to a `cl_command_queue`. When a command to execute a kernel is issued, it executes asynchronously with the host, not guaranteeing that it will start immediately.

2.2.2.3 Summary

The directives supported by the Intel compiler, OpenMP 4.0 and OpenCL, all feature constructs to offload computations to the co-processor. However, to make use of Xeon Phi's offload model, the host side orchestration must be explicitly managed by the programmer. This is particularly noticeable when in the presence of compound computations, such as pipelines. An alternative is MPI, which allows for message-passing communication between threads running in the host CPU and in the co-processor, being important for the symmetric model. Nonetheless, MPI's API is known to be low-level and cumbersome. Thus, the programmer is still burden with low-level details.

Several of the features incorporated in OpenMP 4.0, such as task affinity, multiple teams and the *SIMD* constructs, complemented by the clauses to specify byte alignments or work distribution among teams, provide a tuning framework for exploiting the advantages of the multiple cores and the *SIMD* lanes present on the Intel Xeon Phi.

The support of these technologies for the Xeon Phi appears to still be very primitive, but we personally assessed it doing some experimental tests.

2.3 Preliminary Results

After the initial research for the tools available to program the Xeon Phi, a comparison between their performance was conducted to choose the technology that would be used in the scope of this thesis. The results of that comparison are presented in this section.

There were two important aspects that we wanted to measure, the performance of memory transfers, and the performance of the execution itself disregarding the communication overhead.

The first test consisted in allocating an array of `char` elements, fill it with randomly generated values and then measure the time required to transfer the data in and out of the co-processor. For each technology there were executed 20 tests, from 100 Mbytes to 2 Gbytes of data with a step of 100 MBytes. In Figure 2.5 are depicted the results obtained and it can be observed that with both OpenMP and Cilk the data transfer is faster, 52% and 61%, respectively.

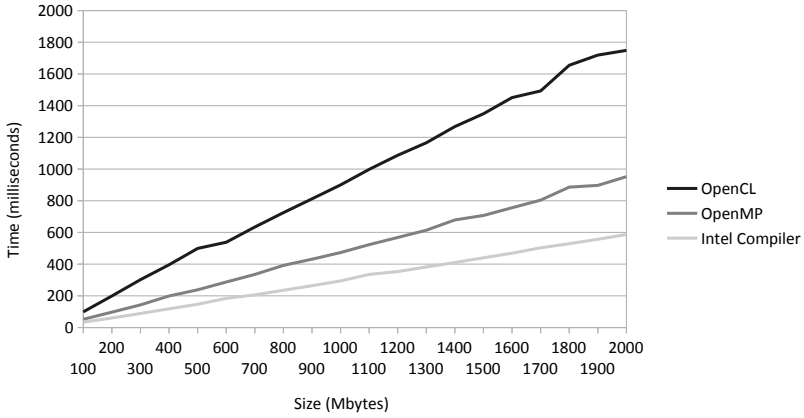


Figure 2.5: Transfer times with different technologies

After the evaluation of the performance of data transfers, the performance of common applications was measured. For that, the first application was the well known matrix multiplication. The method use to perform this calculation was to assign to each thread an element of the resulting matrix, so, the thread that needs to compute the value of the position i, j, has to transverse the row i from the first input matrix and the column j from the second input matrix. In the case of the column the naive approach would use the original matrix which would result on the values being dispersed by the input struct. To prevent this, and with the goal of taking advantage of the *SIMD* lanes available, the second matrix is transposed so that the values from the same column are contiguous. The results obtained are presented in Figure 2.6. These results show that, if the potential of the hardware is fully used, the performance can be tweaked to be better when compared with tools that do not offer such freedom.

The previous example was already really promising, however, the scope of application of Marrow is not limited to these simple computations. Hence, we also evaluated the relative performance of OpenCL, OpenMP and Cilk Plus when thread communication is

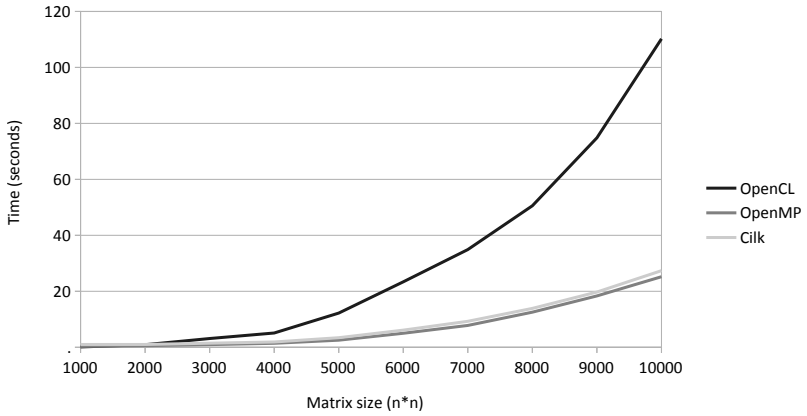


Figure 2.6: Execution times of integer matrix multiplication with different technologies

required. In this case, the example that was used was the N-Body problem. As it can be observed, in the results presented in Figure 2.7, both OpenMP and Cilk Plus have a better performance, of approximately less 50% of execution time, than OpenCL. These results support the hypothesis that with tools that provide more control over the runtime, like OpenMP, it is possible to achieve a better performance than with specialized tools like OpenCL, since the mechanisms can be fine tuned for each case.

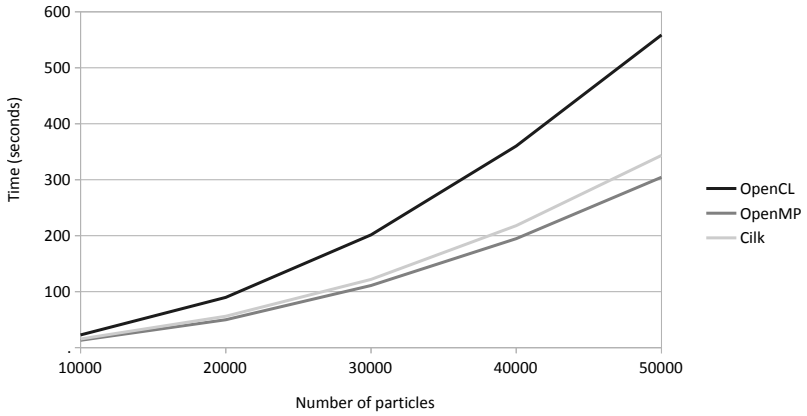


Figure 2.7: Execution times of the N-Body problem with different technologies

At this point, there were good results already favoring the use of the OpenMP technology but, given that all the previous tests were performed using applications developed by ourselves, we wanted to use a set of examples that were specially developed to measure the performance of a device with highly parallel applications. For that, we resorted to an OpenCL and OpenMP adaption of the NPB benchmark suite [SJL11], a suite of parallel benchmarks, originally developed by NASA[1], to evaluate the performance of parallel supercomputers, implemented in OpenMP and OpenCL.

In order to execute the benchmarks on the Xeon Phi, the OpenCL versions did not require any modifications because they were already adapted to receive as an argument

[1]http://www.nas.nasa.gov/publications/npb.html

the type of device to be used, which in this case was *accelerator*. However, the OpenMP versions had to be adapted to offload the computation to the co-processor, as these were written to be executed on CPUs. There were only two fundamental changes to be performed, the offload of the `main()` function and the compilation of the auxiliary modules. The first one consisted in creating a new main function, `main2()`, that simply offloads the execution of the original one using the `#pragma offload target(mic) inout(argc) in(argv:length(argc))` directive, and the second was the generation of Xeon Phi machine code for the included headers and macros, through the `#pragma offload_attribute` directive (see Section 2.2.2.1).

From all the available benchmarks, the ones used were the ones that, after the adaptations, compiled using the Intel compiler, namely:

IS Integer Sort, random memory access;

FT Discrete 3D fast Fourier Transform, all-to-all communication;

SP Scalar Penta-diagonal solver;

EP Embarrassingly Parallel.

For each benchmark, there are multiple predefined problem sizes that are represented by different classes, from A to D, increasing in size. When trying to execute the different benchmarks, some of the higher classes aborted with runtime errors so the examples were executed to the highest possible class.

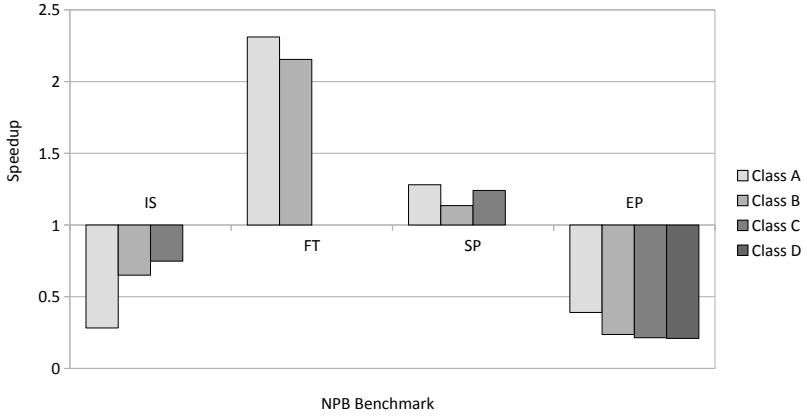


Figure 2.8: Speedup of the OpenMP versions of the IS, FT, SP and EP benchmarks relatively to their OpenCL versions, in Intel Xeon Phi

Figure 2.8 presents the speedup of OpenMP compared to the OpenCL version. The results show that OpenCL has a better performance in the EP benchmark, with speedups tending to 0.25. This is the type of computation OpenCL is tailored for, and hence is expected to deliver better performances than the remainder technologies. However, in the benchmarks that have a high level of communication, like the FT benchmark, the speedup tends to 2. For the other two benchmarks, IS and SP, the speedup gets close to 1,

hinting that the performance is the same with either technology for average applications considering the trivial modifications to offload the computations.

2.4 Conclusions

One of the advantages of using a device like Xeon Phi is that not only it provides an high level of parallelism, like GPUs, but at the same time, due to its x86 compatibility, the most common tools used in multi-core processors can also be used. This means that, in addition to the higher level of programming provided by such tools when compared to technologies like OpenCL, all the knowledge gathered in the context of multi-core processors can also be applied in this case.

From the OpenCL versus OpenMP versus Cilk experimental results, it can be concluded that, for hand written applications, if the hardware is correctly used, the performance of technologies that provide a lower level of programming, such as OpenMP and Cilk, can be better than more generic technologies like OpenCL. From these tests the tools that were chosen to be used as support for the proposed work was OpenMP for the parallel execution on the co-processor and the compiler pragmas for the data transfers because they provided the best performances. Both the results from our tests and from the SNU NPB tests were promising and, specially for OpenMP, the support provided by the community about this tool is very large.

3

The Marrow Algorithmic Skeleton Framework

Marrow [AMP14; Mar+13; SAP14] is a C++ ASkF (**A**lgorithmic **Sk**eleton **F**ramework) for general purpose computing in computing nodes featuring multiple CPUs and GPUs. It provides a library of skeletons for task- and data-parallelism, to orchestrate and hide the details of the programming model related to OpenCL, leaving only to the programmer the task of providing the parallel computations, in the form of OpenCL kernels. These skeletons may be nested onto a computation tree in order to build complex behaviors. Furthermore, the framework transparently performs a number of device-type specific optimizations, such as the overlap of computation with communication to reduce the overhead of data transfers between the host and the GPU (which is usually carried out over the PCIe bus), and the division of the CPU devices taking into consideration cache affinities.

The framework currently supports the following five types of skeletons:

`Pipeline` Applies multiple computations to the input data in a way such that the output of a stage becomes the input to the next stage. This skeleton is particularly suitable for accelerator computing, in general, because the intermediate results can be kept on the accelerator's memory, disregarding the overhead of transferring them back to the host.

`Loop` (and `For`) This skeleton iterativelly applies an OpenCL computation, being that the loop's control condition may, or not, depend on data computed by the nested kernel. The former is akin to a while loop, while the second is closer to a for. In fact, due to its wide use, the `For` specialization of the `Loop` skeleton is featured in

the framework's skeleton library. Besides, there are still two modes of execution in a multi device environment, one is when each iteration needs the data modified by previous iterations requiring the existence of a synchronization point at the end of each iteration, the other is when the iterations are independent so all GPUs can execute their work without waiting for the remainder devices.

`Map` and `MapReduce` When a computation is to be applied to independent partitions of the input data the `Map` skeleton should be used. In case of, at the end of applying the computation to all partitions, the multiple results need to be aggregated, the `MapReduce` skeleton is the one to be used. All of the skeletons support nesting, meaning that they can be used as inner nodes enabling the composition of more complex computations.

3.1 Architecture

Marrow's architecture, depicted in Figure 3.1, comprises two layers: the skeleton library and the runtime module. The programmer only has access to the library layer that provides the multiple skeletons.

As mentioned before, a computation is represented as a tree. In this tree, the inner nodes are the skeletons aforementioned, and, the leaf nodes, are instances of the class `KernelWrapper` that encapsulate the actual computation kernels. This wrapper is responsible for managing the `cl_program`'s and `cl_kernel`'s necessary for its execution and it is the one that sets the arguments of the kernel and issues its execution resorting to the OpenCL routines. In addiction, there are also the kernel data-types, of type `IWorkData`, that represent the different arguments of the kernel, namely:

BufferData represents a contiguous memory region. It is used to represent arrays. Beyond the type of the elements, the minimum indivisible size to which it can be partitioned into, the mode of access and the mode of partitioning can also be indicated;

FinalData/SingletonData are used for constant values that are always the same for every thread. When the value is known by the time the tree is being constructed, the `FinalData` class is used, whereas if the value changes for each execution, the `SingletonData` class is the correct one. There is no additional information for these data-type besides the type of the value for both types and the value itself for the `FinalData` type;

LocalData is used to represent data that is allocated only on the device and not on the host. Similarly to `SingletonData`, the type of the elements is the only information provided.

In order to be able to execute the same tree with different inputs, the buffers are only provided at the execution stage. The `Vector` data container serves as a wrapper

to those buffers, abstracting all inter-device communication and data partitioning (when such partitioning is required).

The runtime layer is responsible for dealing with all the details inherent to the management of OpenCL computations, namely, problem decomposition across multiple devices, scheduling, and communication (including synchronization). The `Scheduler` module is used to decide the distribution of partitions among all the available devices using the `Auto-Tuner` module to decompose the domain of a kernel. The transfer of data to a device and the consequent execution of the kernel is dealt by the `Task Launcher` module. The `ExecutionPlatform` module is the main responsible for interacting with the underlying platform. In the case of the OpenCL devices, is this module that manages, for example, the command queues associated with each device. Recently the support for multiple `ExecutionPlatform` modules was added so the computation tree can be executed exclusively, or in parallel, by multiple platforms.

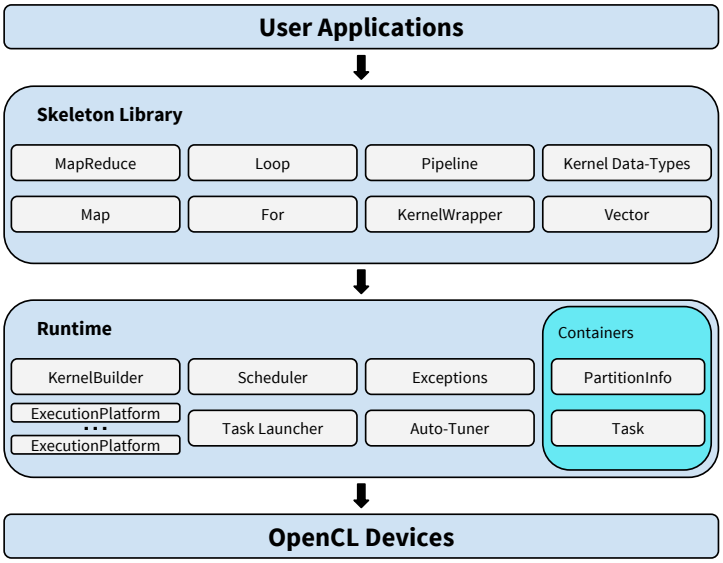


Figure 3.1: Marrow's architecture

3.2 Programming Model

The programming model in Marrow can be defined in three stages: allocation of the tree, execution requesting and de-allocation of the tree. To help understand how the framework is used, a simple example of how to build a three stage pipeline is illustrated in Listing 3.1. This example consists in applying three filters, namely the Gaussian Noise, the Solarize and the Mirror filters, to an image.

The tree allocation phase is represented by the instantiation of the three kernels and the pipeline skeleton (lines 3 to 9). For each kernel, akin to OpenCL, the file of the source

Listing 3.1: Image filter pipeline using the Marrow framework

```
1  //Create each kernel
2  //gauss
3  KernelWrapper gauss = kernel<Vector<uchar4, 2>, const int, uchar4*> ("
       gaussian_noise_kernel.cl", "gaussian_transform");
4  //solarize
5  KernelWrapper solarize = kernel<Vector<uchar4, 2>, const int, uchar4*> ("
       solarise_kernel.cl", "solarise_transform");
6  //mirror
7  KernelWrapper mirror = kernel<Vector<uchar4, 2>, uchar4*> ("mirror_kernel.cl",
       "mirror_transform");
8  //Create the pipeline with the multiple stages
9  Pipeline filter_pipeline = pipeline (gauss, solarise, mirror)
10 //Set input/output arguments
11 //input image
12 Vector<cl_uchar4, imgWidth > input_image ( imgWidth*imgHeight  )
13 //fill input_image
14 ...
15 //output image
16 Vector<cl_uchar4, imgWidth > output_image ( imgWidth*imgHeight  )
17 //Request tree execution
18 Future future = filter_pipeline(input_image, output_image);
19 //or Future future = filter_pipeline.run(input_image, output_image);
20 //Wait for results
21 future->wait();
22 //Cleanup
23 delete future;
```

code and the name of the kernel are provided. In addition, so that the runtime can characterize the kernel, the type of the arguments, which, in this case, are a vector of `uchar4`, a constant integer, and a pointer to an `uchar4`, denoting an output vector of that same type. When representing an array, both the type and the number of elements per threads are specified, which is 2 in the example because each thread will apply its filter to 2 pixels of the received image. After all the kernels are instantiated, the pipeline is defined by simply stating the order of the stages (line 9).

Once the tree is allocated, the input and output `Vector`'s are created (lines 12 to 16). Each one is parameterized with the type of its elements and the indivisible unit (which in this case is chosen to be an image line) and it is instantiated with the number of elements. The input `Vector`'s are filled just like with a normal array, using the indexation operator (`[]`).

Finally, the execution request (line 18), prompt through the `run` routine or the `()` operator, returns a future object. Hence synchronization is data-centric, host and skeleton execution may run concurrently until the former requires the result output by the latter by waiting for the result with the `wait` routine. After the computation is completed, the `Future` object must be deleted.

3.3 Execution Model

Marrow's internal execution model is depicted in Figure 3.2. The issuing of a Marrow tree execution request (step 1) triggers the creation of a future object (step 2) that is subsequently returned to the issuing thread (step 3). Concurrently the task to be executed (the Marrow tree) is submitted to the `Scheduler` so that the work can be distributed among the target devices (step 4). The resulting partitions are subsequently scheduled to each device, as they are pushed unto per-device work queues (step 5). When possible, the `TaskLauncher` removes a task from a queue (step 6), and requests the corresponding `Skeleton` to launch the data transfer and execution (step 7). The `Skeleton` issues the `ExecutionPlatform` of each device to transfer the corresponding data and to execute the kernels (step 8). Upon completion, and when all output data is available to be used, the `TaskLauncher` notifies the future object associated to the computation that the task is finished (step 9). The future object then notifies the application, by awaking it in case it had requested the results before they were available, to inform the results are ready to be used (step 10).

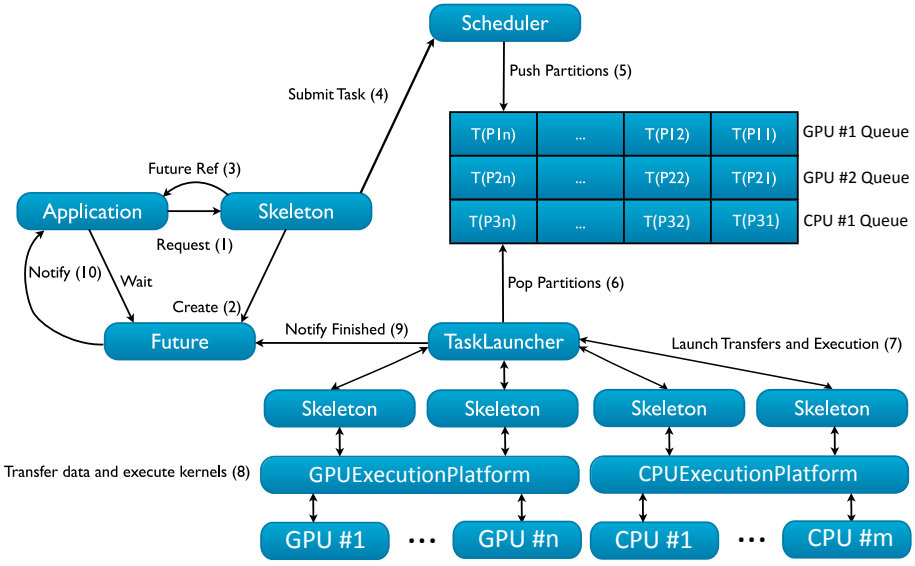


Figure 3.2: Marrow's execution model

As was explained, the `TaskLauncher` is the one that manages the execution of the computation on the multiple devices. The OpenCL SPMD (**S**ingle **P**rogram **M**ultiple **D**ata) execution model is preserved in the presence of multiple kernels and multiple devices, which means that each kernel is executed in parallel on all devices but only one kernel is executed at a time. This prevents, for the example presented, the Solarize filter from being applied to the already processed pixels from the first stage, while there are still pixels to be processed in the first stage. Additionally to the partition of the input data performed by the `Scheduler`, the `Skeleton` also runs some tests before the actual execution. These tests are performed to decide the level of overlapping and fission to be

used and all the combinations are considered.

The Marrow framework was designed from the beginning to be used with OpenCL, hence some of the underlying mechanisms, such as its execution model, were highly influenced by the latter. For example, just as it happens with OpenCL, where the data transfers are independent from the actual execution, on Marrow, there are three stages for the execution of a computation three, the upload of the input data to all the devices, the execution of the computation and the download of the results back to the host.

4

Marrow on Xeon Phi

This chapter will describe most of the work proposed in this thesis. Section 4.1 starts by explaining the efforts for adapting the Marrow framework for the Xeon Phi keeping the OpenCL runtime. In Section 4.2, the changes performed to the framework in order to support computations defined as C++ functions. The work described in Section 4.3 was developed to give the framework an initial runtime support for streaming of data. Section 4.4 concludes with the adaptations to the framework, in order to support multiple technologies.

4.1 OpenCL Version

OpenCL support for the Xeon Phi meets all the requirements of the current version of the Marrow framework. Accordingly, the addition of Xeon Phi support in Marrow can be tackled just at the OpenCL backend level, the `ExecutionPlatforms`. Prior to the work developed in the scope of this thesis, Marrow comprised two different `ExecutionPlatforms`, one for each type of supported processing units, namely CPUs and GPUs. The main differences between these platforms are at the optimization level. The `CPUExecutionPlatform` is tuned for exploring cache locality, using for that purpose the OpenCL CPU fission functionality. As for the `GPUExecutionPlatform`, in order to minimize the overhead associated with the CPU-GPU communication performed over the PCIe bus, GPU computation is overlapped with bi-directional data transfers to and from the GPUs - a technique that we refer to as overlap.

Similarly to GPUs, the Xeon Phi is a many-core accelerator plugged in the PCIe bus. Accordingly, the overlap feature is an optimization parameter that may well yield performance gains when applied in the co-processor. To that extend, the

`GPUExecutionPlatform` was chosen as the base for the `XeonExecutionPlatform`.

Due to OpenCL's device-type abstraction, the `GPUExecutionPlatform` and the `XeonExecutionPlatform` are virtually identical, differing only in the type of device that each one issues to the OpenCL runtime, namely `CL_DEVICE_TYPE_GPU` and `CL_DEVICE_TYPE_ACCELERATOR`. After this change the new `ExecutionPlatform` was added to the Marrow runtime, like depicted in Figure 4.1, and all the examples successfully ran on the co-processor.

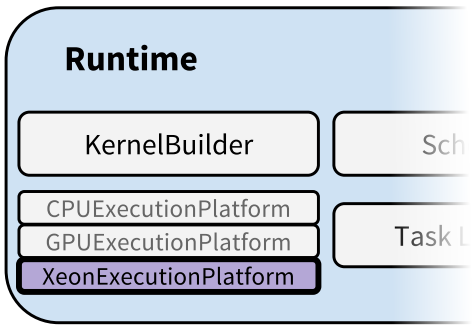


Figure 4.1: New Runtime with all `ExecutionPlatforms`

The OpenCL framework features many device type independent mechanisms, like the creation of command queues, transfer of data to the device or the build of `kernels`. Accordingly, with the purpose of factoring all these mechanisms, a generic `ExecutionPlatform` was created to enclose all these mechanisms so that more OpenCL platforms can be added to the framework without the need the re-write all these mechanisms. This platform is subclassed by the CPU, GPU and Xeon Phi `ExecutionPlatforms` as illustrated in Figure 4.2. The generic class features not only all the methods required to get information about the devices like the number of devices, the type of the devices or the amount of total and free memory available but also the methods for allocating and freeing memory and for uploading and downloading data to/from the devices. As for the specific `ExecutionPlatforms`, they only have the code responsible for collecting all the devices available on the system, the allocation and creation of contexts as well as for storing the information about the memory available on each device.

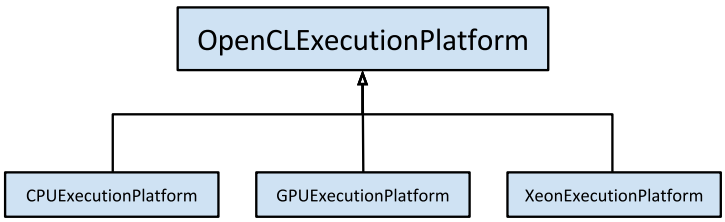


Figure 4.2: Class diagram of the multiple `ExecutionPlatforms`

4.2 C++ Version

Having in mind the Xeon Phi's compatibility with the x86 architecture, a pure C++ version was developed in order to take fully advantage of the hardware without requiring the programmer to express its computations in OpenCL, and, with that, raise the level of abstraction. Dropping the OpenCL dependence of the library lead to some significant changes. Firstly, the computations are no longer defined has OpenCL *kernels*, but rather as simple C++ functors, relieving the user from knowing the OpenCL syntax and semantics, which implies knowledge on parallel programming and computer architecture. Secondly, the parallization of the user-defined behaviors are no longer delegated on the OpenCL runtime. In addition to the orchestration, all parallelization of the execution of the leaf nodes needs to be handled by the framework. Finally, Xeon Phi allows the synchronization between all executing threads, something that was not possible on GPUs and was reflected on OpenCL. Therefore, with the help of the `pragmas` provided by Intel's compiler, the whole tree can be offloaded to the accelerator.

4.2.1 Programming Model

The OpenCL version of the Marrow framework already hides most of the underlying OpenCL mechanisms. Thus, the transition to the C++ version kept the programming model very similar to the existing one. Regarding the creation of the computation tree all the nodes remain the same except the leafs that now are `CFuncWrappers` (Listing 4.1). As for the OpenCL portion of the model, the implementation of each `kernel` was replaced by the definition of a Functor[1] class representing the computation to be applied by each leaf to the corresponding input data. By comparing Listing 4.2 and Listing 4.3, it can be observed that the notion of global id no longer exists and now, the programmer only has to define the computation applied to each element and does not need to deal with the access to such element, given that, on the new version, only the necessary elements are received as arguments, as opposed to the arrays in the kernel code. In order for the code of the Functor class to be compiled for the MIC architecture, its declaration must be enclosed between `#pragma offload_attribute` (lines 2 and 12). This detail could be masked by resorting to a C++ macro. The resort to this type of class relates to the impossibility of the compiler to vectorize the code without the access to the function, whereas with a parameterizable `CFuncWrapper`, its code is only compiled when the parameter is instantiated, allowing the compiler to access the function and vectorize it.

4.2.2 Execution Model

The shifting from OpenCL to pure C++ allows for the skeleton tree to be fully evaluated on the co-processor, instead of on the host. The advantage of this approach is that, with

[1] A Functor class is a class that implements the operator `()` so that it can be called like a function

Listing 4.1: Tree allocation with Functor class

```
// Tree allocation with Functor class
unique_ptr<IExecutable> func(
    new CFuncWrapper<Functor>(inDataInfo, outDataInfo));
skeletonTree = new xeonphi::Map(func);
```

Listing 4.2: Functor class example for the Saxpy problem

```
// Mark the class to be compiled for MIC
#pragma offload_attribute (push,target(mic))
class Functor {
public:
  Functor() {};
  ~Functor() {};

  void operator () (const float X, const float Y, const float a, float * const
      out){
      *out = X*a + Y;
  }
};
#pragma offload_attribute (pop)
```

Listing 4.3: Kernel example from the Saxpy problem

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

the orchestration taking place in the co-processor, there is no need to transfer the control between the host and the device. To illustrate the main differences between the OpenCL and the C++ versions, consider a computation tree that represents a 2-stage Pipeline, each performing a map operation (see Figure 4.3).

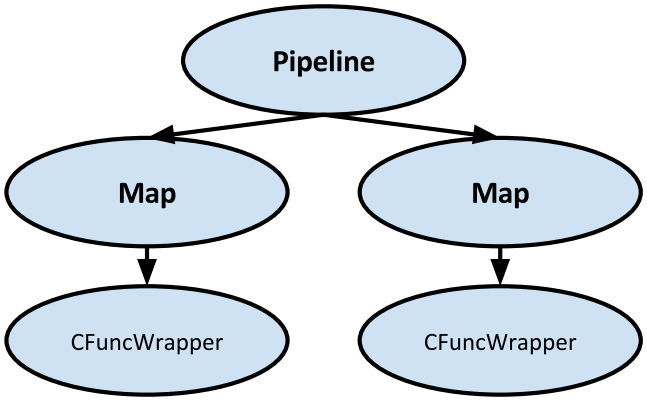


Figure 4.3: Computation tree of a two stage `Pipeline` with two `Maps`

For a tree evaluation request, the first step is to transverse the tree in order to allocate the memory necessary for the execution of each skeleton. Regarding the `Pipeline`, besides the memory for the input and output data, an intermediate memory is allocated to serve as a temporary storage for the results that are produced by the `Pipeline`'s first stage, and will be consumed by the next stage. As for `Map`, since there is no need for additional storage, and because the memory for the arguments is allocated by the parent node, no memory is allocated by this skeleton. After all memory has been allocated, the input data is then uploaded to the device.

Next, the execution starts at the root of the tree and is propagated downwards, and left to right, until it reaches the leaves. When a leave is reached the respective `kernel` is executed, leaving the task of its parallel execution to the OpenCL runtime, following the SPMD execution model. While the `Map` skeleton only has to request an execution of the `kernel` using the input and output data, the `Pipeline` needs to request the execution of the first stage passing the intermediate memory as the output memory and then, for the second stage, use it as the input memory. Marrow resorts to multiples queues of commands to avoid having the `Pipeline` waiting for the first stage to finish before requesting the execution of the next stage. However the execution of both stages is performed in a sequential way. This can have a cost in performance considering that the second stage could start processing the results of the first stage before its completion.

One limitation imposed by GPUs is the lack of global synchronization, entailing that, if the computation requires that the various instances synchronize at one point, the control needs to transferred back to the host. To achieve this behavior, the computation must to be split into multiple kernels so that, when each kernel finishes its execution, a synchronization happens. This approach can be expensive because when there is the need to

Listing 4.4: Example of how parallelization is achieved

```
#pragma omp parallel for collapse(3)
for (k = 0; k < size3; k++)
for (j = 0; j < size2; j++)
for (i = 0; i < size1; i+=CHUNK_SIZE){
  unsigned int pos = k*size1*size2 + j*size1 + i;
```

make some decisions based on the modified data, like it happens with the evaluation of the `Loop` skeleton's stop condition, there has to be a data transfer from the device to the host and then, possibly, the data is sent back to the device.

For the C++ version, taking into account that there is no restriction and the synchronization can be performed on the device, the control can be sent to the device itself (Section 4.2.4), along with the computations, thus eliminating the transfer with the host of both data and/or code.

When all the data is done been processed by the tree, the results are then downloaded back to the host, storing it on the memory provided by the user.

4.2.3 Parallel execution of the leaf nodes

Due to the different models of execution of OpenCL and C++, the task of executing the kernels in parallel needs to be implemented by the framework. Since every skeleton may receive a C++ functor as argument, they must be equipped with the mechanisms to parallelize the kernels. To perform this task, any tool which had its runtime system compiled for the Xeon Phi could be used. From the studied tools, OpenMP was chosen because, as opposed to Cilk plus, it allows the fine tunning of some aspects like the number of threads to use or the scheduling strategy for distributing the work among those threads.

For the `Map` skeleton the strategy is to spawn multiple threads that will execute the kernel over some portion of the input data. In the programming model defined for the C++ version, the provided function has to define the computation to be applied to an indivisible unit of the input data, so, in this context, it only has access to the data subset really needed for its implementation. From this, the problem of how to access a multi-dimensional structure emerges, since `Vector` is a flat representation of this type of structure, and so, to tackle it, a very common strategy of storing this type of structures was used. The access is accomplished by considering that the whole struct was flattened to a single dimensional space where the elements from each dimension follow a row-major order organization. For the parallelization itself, the iterations of three (number of dimensions supported by `Vector`) `for` constructs that iterate over the dimensions of the input data are distributed by the available threads. Listing 4.4 shows the pragma used, where $size_i$ denotes the size (in number of elements) on dimension i of the input data, `CHUNK_SIZE` is the maximum number of elements that each thread has to process per cycle iteration, and `pos` is the position where those elements start.

Optimizations Initial experimental results revealed that the execution time was bigger than the OpenCL version so, some optimizations were performed. The first tweak tested was the scheduling method used by OpenMP to distribute the work among all threads but after some testing it was concluded that both `dynamic` and `static` had the same performance, so the default setting was kept (`static`). Another issue that was addressed was the granularity of the work assigned to each thread (`CHUNK_SIZE`). Initially, the number of elements assigned to each thread was the number of elements which could be processed using the VPU available to each core considering that its width is 512 bits. This meant that, if the input elements where of type `int`, then 16 elements would be assign to each thread ($512\ bits/(4\ bytes * 8\ bits/byte) = 16$), or, in case the elements were of type `float`, each thread would receive just 8 elements at once ($512\ bits/(8\ bytes * 8\ bits/byte) = 8$). More experimental results revealed this value was so fine grained that the overhead of issuing the execution was bigger than the actual cost of the execution. To minimize this impact, the number of elements was changed to be $1/(n_{cores} * 2)$ of the input size.

Despite the performance improvements, the optimization of the granularity per se was not enough to match the performance achieved by the OpenCL version. A closer look of the behavior of the framework, revealed that the code that was executed by each thread was not being vectorized. This had a high impact on the performance considering that one of the key features of the Xeon Phi is the availability of VPUs on every core. This problem existed because the vectorization is performed by the compiler and, since the execution code was passed as a function pointer, it could not access it at compile time, therefore not being able to assume that the code was likely to be vectorized. The only way discovered to tackle this problem was by resorting to the C++ template feature. This feature consists in parameterizing a class with a generic type that is unknown when the class is (partially) compiled but that has to be provided when the code that uses that class is compiled. Since the vectorization is to be performed to the code that is applied to the multiple elements given to each thread, the class that was parameterized is `CFuncWrapper`. Instead of passing the function as an argument to the wrapper, a functor class is given as a parameter to `CFuncWrapper` that instantiates a new object of the functor class.

The speedup achieved by each optimization relatively to the previous one is depicted in Figure 4.4. As illustrated, by simply issuing the compiler to optimize the code, via `-O2` flag, the gain in performance was of over 2.5 times. The results show that the tuned grained was the one that provided the biggest improvement, with a speedup slightly over 5. The changes performed to the framework to apply the last optimization, the vectorization of the user-defined function, were rewarded with an increase of over 3.5 times more performance.

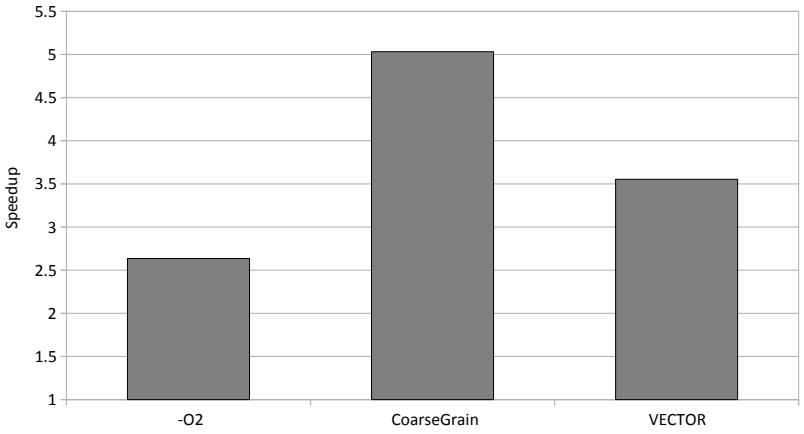


Figure 4.4: Speedup obtained from each optimization

Listing 4.5: Interface to offload skeletons to the co-processor

```
1 typedef void (*fptr)(arguments&, arguments&);
2
3 //Method that returns a function to be offloaded representing the Skeleton sub-
    tree.
4 virtual fptr getSkeletonFunction() = 0;
5
6 //Method that returns the arguments to be passed to the skeleton function.
7 virtual arguments* getSkeletonArguments(unsigned int partitionIndex, unsigned
    int overlapPartition) = 0;
```

4.2.4 Offloading the Computation Tree

The offloading of the computation's orchestration to the device imposed several design changes to be made to the skeletons. Instead of recursively going through the tree on the host, a mechanism had to be developed to aggregate all the orchestration into one region to be offloaded and then executed on the co-processor. An intuitive approach would be to simply offload the tree, as it is, to the device and then execute it there. This, however, cannot be done because there is the need to compile all code that will run on the accelerator for the MIC architecture and the current support for this is still early and the Intel compiler does not support the offloading of classes. Although Cilk Plus provides a mechanisms to offload classes, as described in Section 2.2.2.1, it does not support classes as complex as the ones representing the skeletons. Given these limitations, our strategy was to have all skeletons implement one function with the code for its orchestration and then combine all functions into a single one, thus representing the whole computation tree in one function that can be offloaded to the Xeon Phi.

Listing 4.5 presents the interface that every skeleton needs to implement. It is used so that every skeleton can export the code of its orchestration as an function pointer, that we refer to as *skeleton function*, as well has the arguments needed for such orchestration, defined as *skeleton arguments*. With this information available to the parent skeleton, the tree function is then built, virtually, because it is not actually a single function, by passing

Listing 4.6: Example of how to get the pointer to a function defined on the co-processor

```
1  __attribute__((target(mic)))
2  inline void funcMap(arguments& args, arguments& privArgs){
3    //... orchestration code
4  }
5
6  func_ptr getSkeletonFunction(){
7    void* f2;
8    #pragma offload target(mic) out(f2)
9    {
10     f2 = (void*) funcMap;
11   }
12   return (func_ptr) f2;
13 }
```

along with its skeleton arguments, the pointers to the skeleton functions of its children as well as their skeleton arguments' pointers. For a better understanding, lets consider a computation defined by a `Map` with a `CFuncWarpper`. The skeleton arguments for the `CFuncWrapper` are the constant arguments of the kernel function, and for the `Map` skeleton are: the size of each dimension of the input data; the pointer to the function that orchestrates the behavior of the `CFuncWrapper`; and a pointer to the `CFuncWrapper`'s arguments. In order for all the code to be executed on the accelerator side, the pointers must refer to memory addresses on the device. To achieve this, regarding the skeleton function and like it is shown in Listing 4.6, first, the function has to be declared with `__attribute__((target(mic)))` (line 1), so that the code is compiled for the accelerator, and then, to get the address of the function, the de-referencing must be done inside an offload block (lines 8 through 11). Concerning the allocation of the skeleton arguments the approach is the same. From a more technical point of view, explicitly defining the code that has to run on the device has the advantage of reducing the MIC code footprint to the one that is effectively necessary.

4.2.5 Efficient memory transfer

One key aspect of working with accelerators is that, commonly, they are attached to the host machine via some interface that has a non negligible latency. Such is the case with the Xeon Phi. Hence, one of the runtime operations that requires special attention is the memory transfers between the host and the device. Being originally built with the OpenCL execution model in mind, Marrow follows its execution model where the allocation and transfer of memory are performed separately from the execution of the computation. This design prevents the use of the usual approach where the data that needs to be sent over to the accelerator is specified, on the `#pragma offload` used to execute the code on the device, using the clauses available. To abide to this architectural design, the first approach was explicitly allocate the memory on the device by offloading a `malloc` call, saving the returned address. Afterwards, when the data is uploaded to the device, it

is stored in the allocated space using the `memcpy` routine. This approach was considered because it allows to pass the address where the data was allocated in the co-processor to the execution stage. However, the performance results were not good because the data was being moved two times, first, from the host to some space used by the runtime that manages the communication between the accelerator and the host, and then from there to the allocated space.

Since the driver of this optimization phase was the efficiency of the memory transfer between the host and the device, we naturally continued our research in the quest of better solutions. A deep investigation, on how the runtime associates the memory on the co-processor with the memory provided by the user, revealed that the runtime automatically converts host addresses that are used inside offloaded regions, by the corresponding address on the co-processor's memory. This conversion works, as long as the memory has been previously allocated and correctly filled using the available pragmas to transfer data between the host and the Xeon Phi. From this knowledge, the memory transfers were changed to correctly use the `#pragma offload_transfer`. This change resulted in an improvement of 15 times on memory transfers, as shown in Figure 4.5. However, this mechanism required that the pointers provided by the user were saved by the framework, resulting in a modification to the `IWorkData` class (described in Section 3.3), making it store the pointer to the data when execution of the tree is issued.

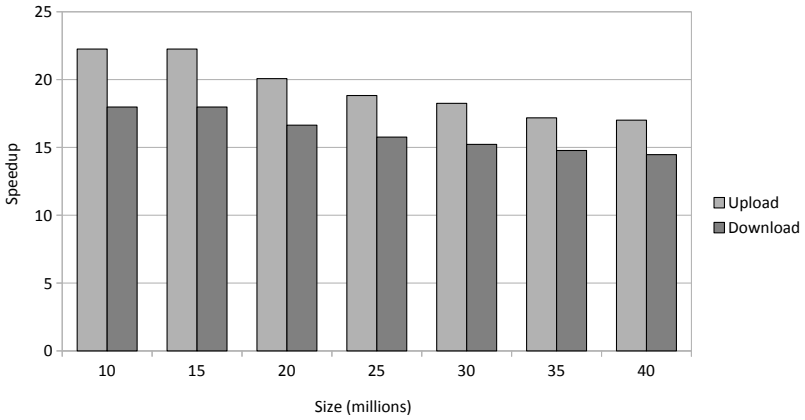


Figure 4.5: Speedup obtained from the usage of `#pragma offload_transfer`

4.3 Adding Data-flow Support

Currently, the framework follows an execution model in batch where the data to which the computation tree is to be applied has to be entirely provided up-front. With the goal of allowing a stream of data to be provided to the computation tree, an initial version of the underlying support was developed.

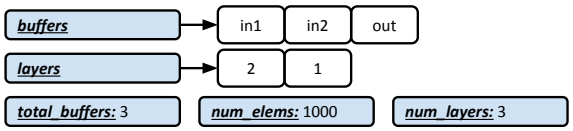
In order to support a stream of data, a concurrent queue was implemented from where each skeleton would consume data or, in case there was no data available, they

Listing 4.7: Structure of a `working_unit`

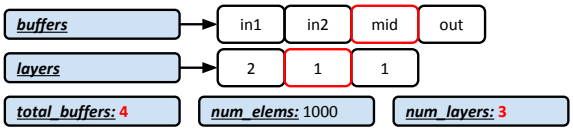
```
typedef struct working_unit{
  typedef struct buff {
    void* vector;
    unsigned int elem_size;
  } buff;
  buff** buffers;
  unsigned int* layers;
  unsigned int num_layers;
  unsigned int total_buffers;
  unsigned int num_elems;
  bool end_of_stream;
} working_unit;
```

would block waiting for new data. As such, there are always two queues between skeletons: the *input* and the *output* queues. The elements to be processed are enclosed inside a `working_unit`, whose structure is depicted in Listing 4.7.

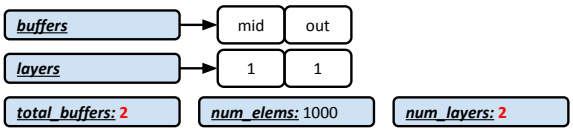
The initial approach was to have an array with the pointers to the input and output buffers of data, and their respective sizes, that the next level on the tree needs. This solution works fine for a single `Map` but it proved to be insufficient for more complex structures, such as a `Pipeline` of two stages, proved to be insufficient. The problem is that, if the `Pipeline` creates a `working_unit` with the input buffers and the middle buffers to where the first stage should write the results. When it has finished processing the unit and needs to pass it to the second stage, it does not know the pointer to where the second stage should write, because that pointer was given to the `Pipeline` and not passed to the first stage. This problem was tackled by keeping the information of how many layers of buffers there are. A layer of buffers is set of buffers that will be used as input or as output for a kernel. This information is constructed as the tree is covered and each `Skeleton` adds a new layer if the next level of the tree needs any extra buffer. For the `Pipeline` skeleton, a layer with the middle buffers is added after the first layer. For the example, if the first stage applies a function that receives elements from two buffers and writes to a single output buffer, and the second stage reads elements this latter buffer and writes to yet another output buffer, an `working_unit` received by the `Pipeline` has the structure shown in Figure 4.6(a). In turn, the unit passed to stage 1, illustrated in Figure 4.6(b), is augmented with the buffer allocated for the inter-stage communication, placed in position 2, and the layers are updated to state that the second one has size 1. When a stage receives an unit, it simply reads the values from the first layer as input and writes to the second layer every time. In this case, the second layer will correspond to the memory allocated for the temporary results. Upon completion, the unit that is passed to the next stage (Figure 4.6(c) for the example) is the one received, with the first layer removed, since that data was already consumed, which means, in our example, that second stage will read from the temporary memory and write for the final memory.



(a) Unit received by pipeline



(b) Unit received by stage 1



(c) Unit received by stage 2

Figure 4.6: Example of the `working_unit`'s received by the `Pipeline`, the stage 1 and the stage 2

To prevent the excess of working units, only the skeleton that executes the parallelization, in this case the `Map` skeleton, splits the working units into smaller chunks. This means that, the root of the tree only adds one unit to the queue, with the number of elements of the input data. When a `Map` consumes a unit with multiple elements, it splits then into chunks to be passed to the executing threads.

One advantage from this execution model is that, since the data now flows through the tree, the two stages of the `Pipeline` can be executed in parallel and, when the result of the first stage, the second one can start processing it right away.

4.4 From a Single Backend to a Multiple Backend Marrow Framework

This final section reports some code adjustment operations that had to be applied upon the pre-existing Marrow framework implementation, so that the skeleton library and runtime layers would be technology agnostic. Considering that, initially, the Marrow framework was designed to work with OpenCL, the runtime layer was filled with OpenCL references that needed to be removed so that the new version could still use all the existing mechanisms like automatic configuration selection and work scheduling. Initially, the references to the OpenCL memory objects where replaced by unique identifiers that translated into either OpenCL Memory Objects for the OpenCL version or C++ pointer for the C++ version. The mapping between the identifiers and the memories is enclosed in the `ExecutionPlatforms`. Another OpenCL mechanism that was used by the framework was the use of events to wait for the memory transfers or the execution to end. Given that this mechanism is useful for the framework, the interface `IEvent`

was created to raise the level of abstraction and enabling the usage of the event mechanism, independently from the technology used. The corresponding `OCLEvent` class was developed to encapsulate the OpenCL event objects.

Lastly, in order to keep the library layer independent from the underlying technology and its mechanisms, such as the ones described on Sections 4.2.3 and 4.2.4 for the C++ version, an abstract class representing each skeleton type was created. The details of each technology were moved to specialized versions of each skeleton, under the namespaces `marrow::opencl` and `marrow::xeonphi`, for the OpenCL and C++ versions, respectively. The wrappers of each version, namely `CFuncWrapper` and `KernelWrapper`, were moved to the corresponding namespace. The diagram representing the new structure is depicted in Figure 4.7.

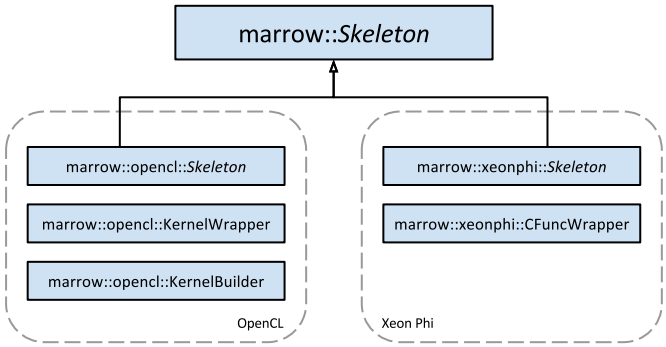


Figure 4.7: Class diagram after the framework generalization

5

Experimental Results

The effectiveness of the solutions developed in the scope of this thesis were evaluated from a performance perspective. More particularly we wanted to find the answer to two questions: "Is the Xeon Phi competitive against current GPUs for the execution of complex OpenCL computations orchestrated by Marrow?" and "How does the new version of Marrow compare to the OpenCL version?". To that end we conducted two different experimental analysis on distinct hardware infrastructures.

5.1 Metrics and Methodology

The Marrow framework includes a suite of benchmarks, developed specially in Marrow, that can be used to asses the performance of the framework. The metric used for each scenario was the execution time of each benchmark. At each run, the benchmark was executed 100 times and the times were sorted. The time of each test is then the averages of the middle 34 times.

5.1.1 Benchmarks

This section will describe the objective of each benchmark, as well as how it is defined as a Marrow computation tree. Except the Series benchmark, all the others were already implemented in the Marrow framework. The available benchmarks are:

Image Pipeline This example consists in a pipeline of image filters that are applied to an input PPM image. The filters are the Gaussian Noise, Solarize and Mirror. These kernels

have been taken from AMD's OpenCL Samples[1]. This benchmark could not be rewritten in C++ because each working unit needs to access two distinct pixels of the image. The images used as examples for the tests were Lena.ppm, Garden.ppm, Ocean.ppm and Andro.ppm having 1024x1024, 2048x2048, 4096x4096 and 8192x8192 pixels, respectively.

The computation tree is composed of two pipelines, resulting in a three stage pipeline with the filters as leaves, being applied in the following order: Gaussian Noise $\rightarrow$ Solarize $\rightarrow$ Mirror.

N-Body This benchmark performs the N-Body simulation, where the position and velocity of a set of bodies are updated, during a certain number of steps, based on the initial values and the forces applied by each body on the others. The kernel follows a classical direct-sum algorithm, of complexity $O(N^2)$ where N denotes the number of bodies, in which a single body is affected by all the other bodies in the set. Like the image filters, the kernel used for this benchmark was taken and adapted from AMD's OpenCL Samples. The input for this benchmark is a file with the particles and the number of simulations to perform.

The computation tree is composed of a For loop where the step function simply increments the counter of the number of steps.

Segmentation This is a tomography image enhancing algorithm. A tomography image is a set of pixels within a three dimensional space, where each pixel represents the amount of absorbed radiation at that position as a gray-scale value. This algorithm changes the value of each pixel, converting its value into either white, grey or black, depending on a threshold. The examples used for this benchmark were cubo100.mat, cubo200.mat and cubo400.mat, which represent cubes with dimensions 100x100x100, 200x200x200 and 400x400x400, respectively.

The computation tree is formed by simple map with the algorithm's kernel.

Solarize This is the application of the Solarize filter, that is used in the Image Pipeline benchmark, to the input image. The images from the Image Pipeline were also used for this benchmark.

The tree of this benchmark is the sub-tree of the Image Pipeline representing only the Solarize filter, namely a single map with the filter's kernel.

Saxpy This example is a matrix operation from BLAS (**B**asic **L**inear **A**lgebra **S**ubroutines). It consists in receiving two equally sized matrices as input, multiplying each element from the first matrix by a scalar and then adding the result to the corresponding value of the second matrix ($z[i] = a * x[i] + y[i]$).

The computation tree is built with a single map with the function's kernel.

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

Series This benchmark computes the first N fourier coefficients of the function $f(x) = (x+1)^x$ on the interval $[0, 2]$. It heavily exercises transcendental and trigonometric functions. The kernel for this benchmark was from the Java Grande Forum Benchmark Suite[2]. This benchmark receives the value of N as input.

The computation tree is assembled with a map containing the function's kernel.

5.2 Intel® Xeon Phi™ vs GPU

The goal of this first set of tests was to compare the performance of the OpenCL version of the framework when executing on three different systems, one with the Xeon Phi co-processor, and the other two systems with GPUs. The infrastructure of each system is the following:

C2050

- **CPU** - Quad-core Intel® Xeon E5506 @ 2.13 GHz
- **Motherboard** - ASUS P6T7 WS Super Computer LGA 1366 DDR3 7PCIE16X
- **Main Memory** - 12 GB RAM DDR3
- **GPU** - NVIDIA® Tesla ™ C2050
- **GPU Driver Version** - 295.41
- **Operating System** - Linux Ubuntu 10.04.4 LTS (kernel 2.6.32-41)

GTX 680

- **CPU** - Quad-core Intel® Xeon E5506 @ 2.13 GHz
- **Motherboard** - ASUS P6T7 WS Super Computer LGA 1366 DDR3 7PCIE16X
- **Main Memory** - 12 GB RAM DDR3
- **GPU** - NVIDIA® GeForce® GTX™ 680
- **GPU Driver Version** - 295.41
- **Operating System** - Linux Ubuntu 10.04.4 LTS (kernel 2.6.32-41)

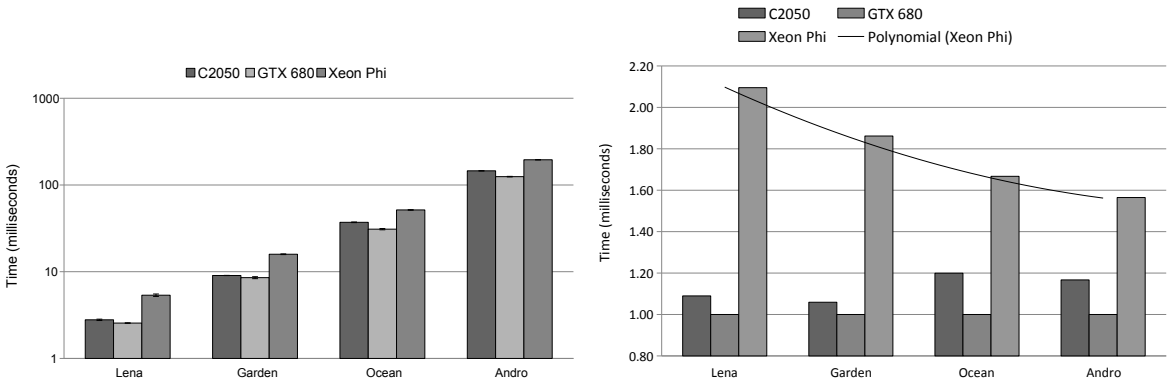
[2] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/seq/contents.html

Xeon Phi

- **CPU** - Quad-core Intel® Xeon E5-2603 v2 @ 1.80 GHz
- **Main Memory** - 16 GB
- **Co-processor** - Intel® Xeon Phi™ 5110P
- **MPSS version** - 13.1.3 20130607
- **Intel®compiler version** - 295.41
- **Operating System** - Linux Red Hat 4.4.7-4 (kernel 2.6.32-431)

5.2.1 Results

In results for the Image pipeline benchmark, it can be observed that behavior of each system is very similar (Figure 5.1(a)). For every input, the system with the GTX 680 is always one that provides the best performance whereas for the Xeon Phi, it is the one with the worst performance. However, as depicted in Figure 5.1(b), the slowdown of the co-processor tends to 1.5, as the input size increases.



(a) Execution times (b) Slowdown relative to the fastest time

Figure 5.1: Image pipeline benchmark on various GPUs and the Xeon Phi

For the N-Body problem, the results are different from the previous one. As it is shown in Figure 5.2, at first, the performance of the Xeon Phi is much worst than the other two system but, has the input size increases, the results are better each time, with a tendency to be below 1, which means a speedup, if the size input data is large enough. Again, the GTX 680 system is the one to deliver the best performance.

The Saxpy benchmark, like the previous ones, reveals that the Xeon Phi system is the one with the worst performance (Figure 5.3(a)). However, unlike the Image Pipeline benchmark, the tendency of the Xeon Phi slowdown is to get near 1.20 (Figure 5.3(b)), which means that, for large inputs, the performance of this system gets closer to the other systems. Just as before, the GTX 680 is the better system.

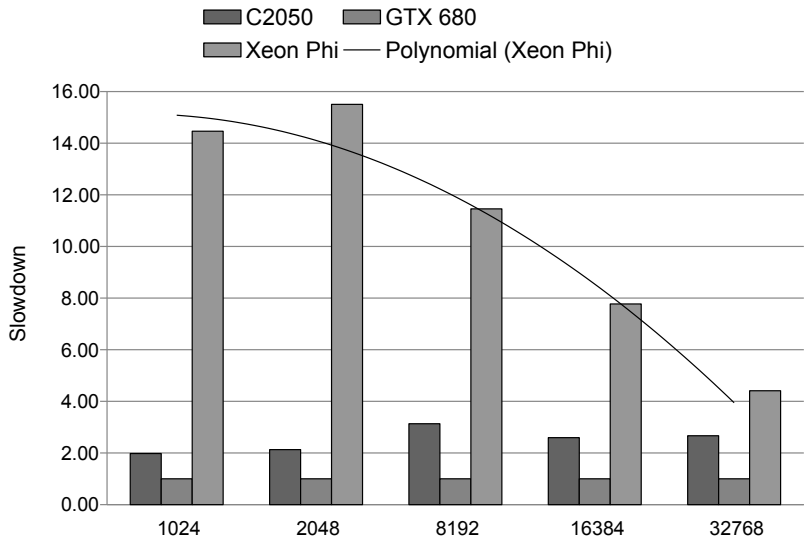
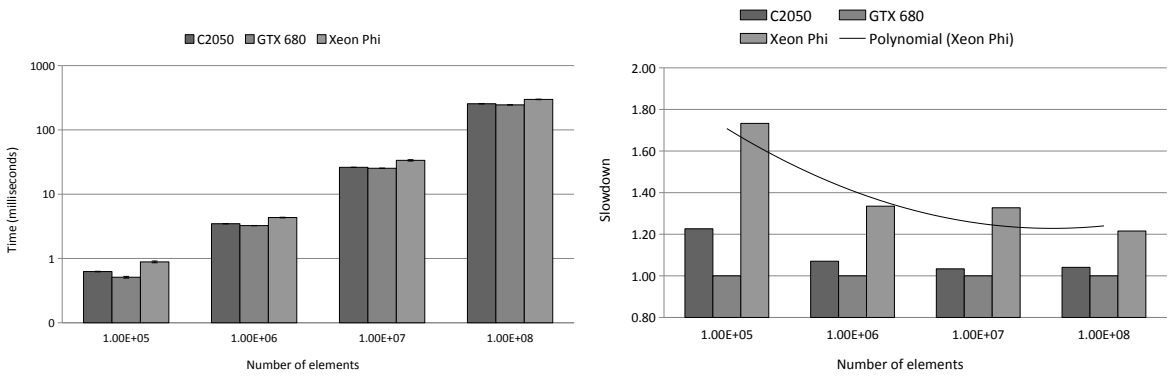


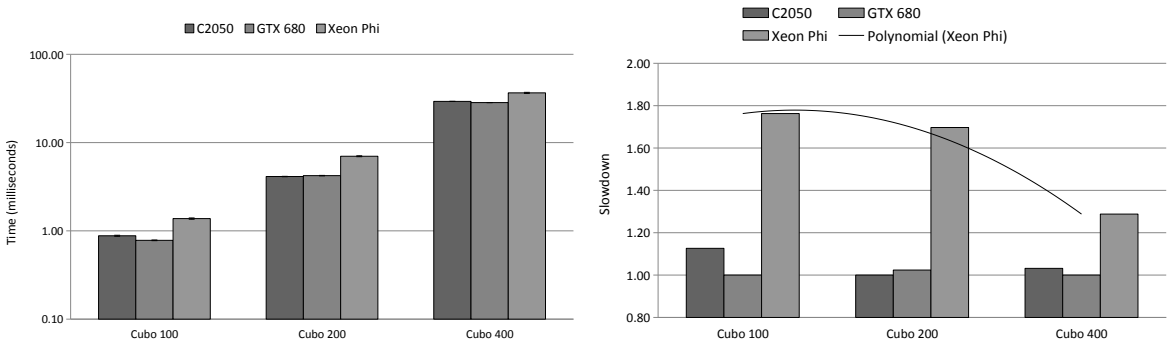
Figure 5.2: Slowdown of the N-Body benchmark on various GPUs and the Xeon Phi, relative to the fastest time



(a) Execution times (b) Slowdown relative to the fastest time

Figure 5.3: Saxpy benchmark on various GPUs and the Xeon Phi

In the Segmentation benchmark, as with the N-Body problem, the tendency of the slowdown of the Xeon Phi system (Figure 5.4(b)) does not stabilize to a fixed value, meaning that, if the input size is large enough, the performance of this system can be better than the other systems. Except from the cubo200 input, the better system is again the GTX 680.



(a) Execution times (b) Slowdown relative to the fastest time

Figure 5.4: Segmentation benchmark on various GPUs and the Xeon Phi

The Series benchmark reveals a different behavior from the other benchmarks. As shown in Figure 5.5, the Xeon Phi is no longer the system with the worst performance, achieving better results than the C2050 system, but still being worse than the GTX 680 system. As in the Saxpy benchmark, the tendency of the Xeon Phi slowdown is to stabilize around 1.20.

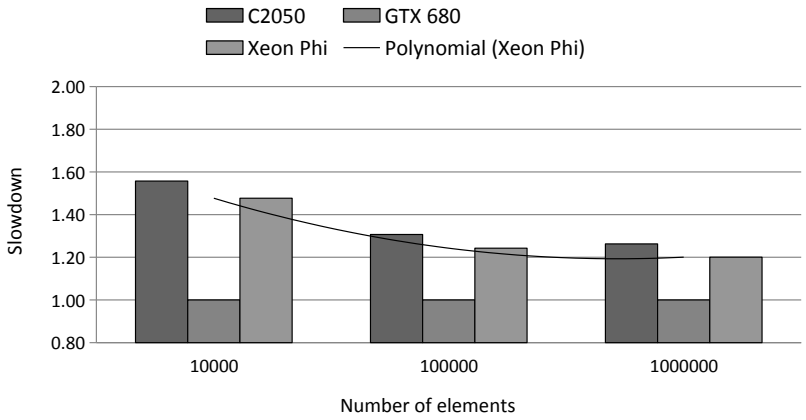


Figure 5.5: Slowdown of the Series benchmark on various GPUs and the Xeon Phi , relative to the fastest time

The final benchmark, and the one with the strangest results, is the Solarize. The results presented in Figure 5.6, show that, despite the decrease in performance of the Xeon Phi system in the Garden example, peaking at a slowdown of near 1.60, the performance of this system tends to keep increasing, and to even surpass the performance of the other systems. As the other benchmarks already shown, the GTX 680 is the system that performs the best.

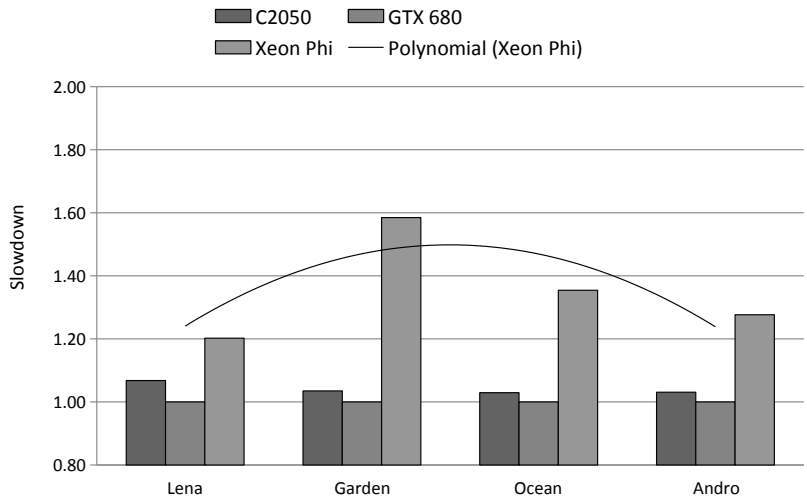


Figure 5.6: Slowdown of the Solarize benchmark on various GPUs and the Xeon Phi , relative to the fastest time

5.2.2 Conclusions

The OpenCL framework was initially very focused for GPUs, hence the performance of an application running on a Xeon Phi compared with its performance on a state of the art GPU is quite low. In [Kis14], report that the performance in Xeon Phi was worst than the performance on a NVIDIA® GeForce® GTX TITAN by a ratio of 82 times more. Although our tests support the same conclusion, the results we obtained did not show such a huge difference.

The results presented above, reveal that, first, the GTX 680 is the system with the better performance across all tests. Then, despite the performance of the Xeon Phi system being the worst in almost every benchmark, the tendency of this system is always to keep improving its performance as the input data size increases. This was expected because, as the cores available in the co-processor are weaker, one of its strengths is the high level of parallelism offered with the high number of cores available and the number of threads per core. It is important mentioning that in the Xeon Phi the overlap level chosen by the runtime was 1 almost every time, not causing any improvement in the performance, whereas for the GPUs this value was correctly adapted, highly favoring the execution time on these systems. Another aspect that was observed was that the Xeon Phi-based system can be quite competitive against GPU-based systems for some classes of problems, such as it happened in the N-Body, Segmentation and Solarize benchmarks. This was shown to be more likely in applications that have a high volume of transcendental functions, in the Series benchmark, where the Xeon Phi had a better performance than the C2050 GPU. To be noted that the OpenCL support for the Xeon Phi is quite recent, due to its youth, and thus not as matured and optimized as the one for GPUs. Accordingly, one can expect that the performance gap that could be observed in several of the benchmarks may be reduced, or even closed, in the future, as this device gets more attention from the community.

5.3 C++ vs OpenCL™ on Intel® Xeon Phi™

To answer our second question, "How does the new version of Marrow compare to the OpenCL version?", we compared the performance of the optimized C++ version against the OpenCL version. In order to perform a deep analysis of the behavior of both versions, the performance of the computation and the data transfers was measured by splitting the execution time into three components: the upload, the execute and the download stages. Given that not all skeletons are available in this version, some of the benchmarks available could not be ported to be used with only C++. So, the benchmarks used in this comparison where the Saxpy, Segmentation, Solarize and Series benchmarks. The infrastructure where this tests were performed was the Xeon Phi system described in Section 5.2.

5.3.1 Results

The first analysis presents an execution time breakdown for the Saxpy benchmark with an input size of 50.000.000 elements. The breakdown contemplates the actual execution time on the Xeon Phi, as well as the host to device and device to host transfer times, download and upload, respectively. The results, displayed in Figure 5.7, show that the data transfer is more efficient in C++, whereas the computation is faster in the OpenCL version. However, the difference between the data transfers times in both versions is bigger that the difference of the execution times, so, the time spent with the data transfers, have an higher impact on the total execution time, resulting on the C++ version having a lower overall execution time than the OpenCL version for an large input size.

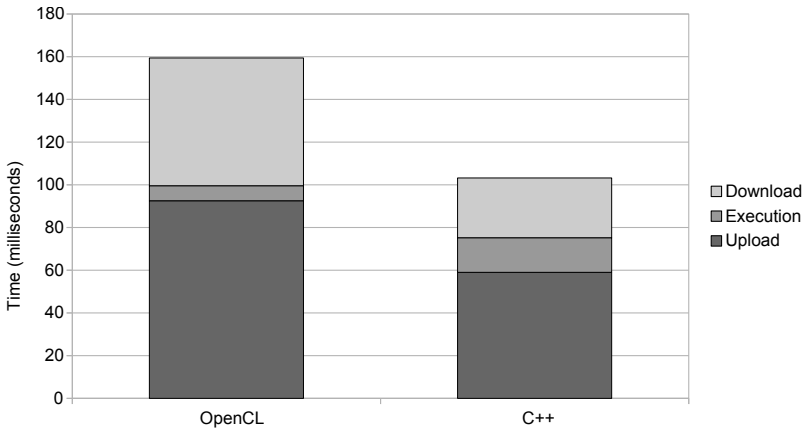


Figure 5.7: Comparison of the various stages between OpenCL and the optimized C++ version on Xeon Phi with a communication bound application

Since the previous example is a communication bound application, which means that it relies heavily in the data transfers, as the input size increases so do the memory transfers. To analyze how the framework behaves with a computation bound application, the Series benchmark with 1.000.000 was executed. The results, presented in Figure 5.8,

illustrate that, for applications where the most of the time is spent doing calculations, the performance of C++ is not much worse than OpenCL, having only a slowdown of approximately 0.8.

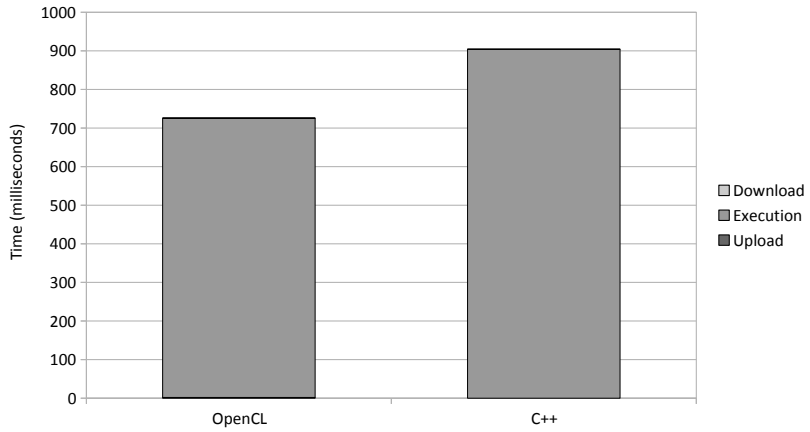
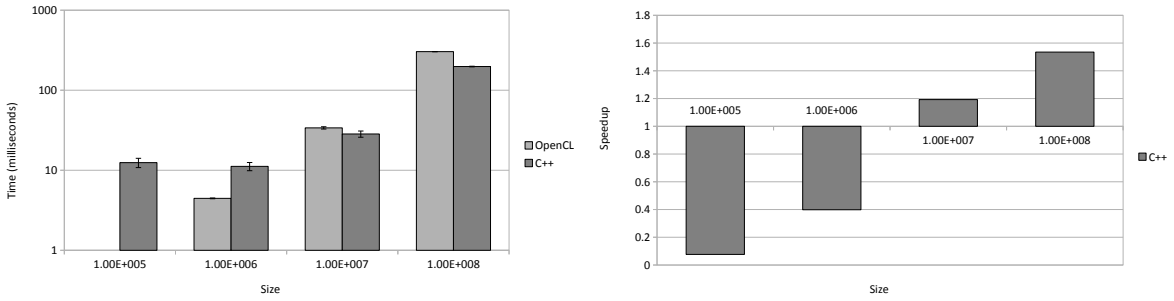


Figure 5.8: Comparison of the various stages between OpenCL and the optimized C++ version on Xeon Phi with a computation bound application

The following analysis addresses the scalability of both version as the problem size increases. The results for the Saxpy benchmark (Figure 5.9) reveal that, for small inputs, OpenCL out-performs C++, with the latter having speedups of 0.1. This was expected after the analysis of the time spent on each stage, but, as the input size grows, the data transfer time becomes more costly. This results in the performance of the C++ version getting closer to the OpenCL one, and, for inputs big enough, even surpass it, with speedups of 1.5.



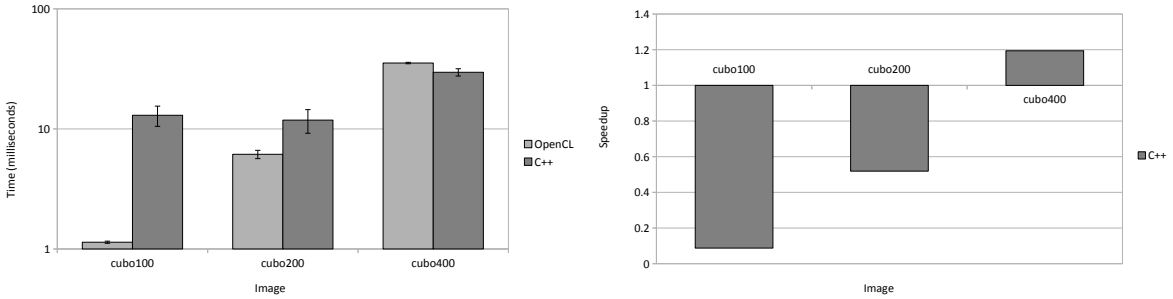
(a) Execution times

(b) Speedup relative to OpenCL

Figure 5.9: Saxpy benchmark between OpenCL and the optimized C++ version on Xeon Phi

Similarly to the Saxpy, in the Segmentation benchmark, the C++ version has a very bad performance for the smaller inputs, achieving speedups as low as 0.1 relatively to the OpenCL version. However, as it happened before, for the cubo400 example, the performance of C++ was better than OpenCL, reaching the speedup of 1.2.

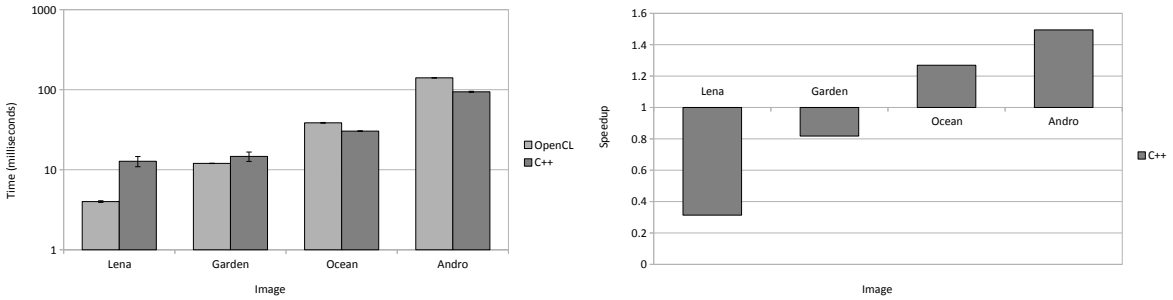
The Solarize benchmark had a behavior analogous to the previous tests, with OpenCL performing better with the smaller inputs, but this time, the C++ version had a slightly



(a) Execution times (b) Speedup relative to OpenCL

Figure 5.10: Segmentation benchmark between OpenCL and the optimized C++ version on Xeon Phi

better speedup for this type of input, of 0.3. Again, as it was already shown, C++ performs better as the input increases, surpassing OpenCL at a peak of 1.5.



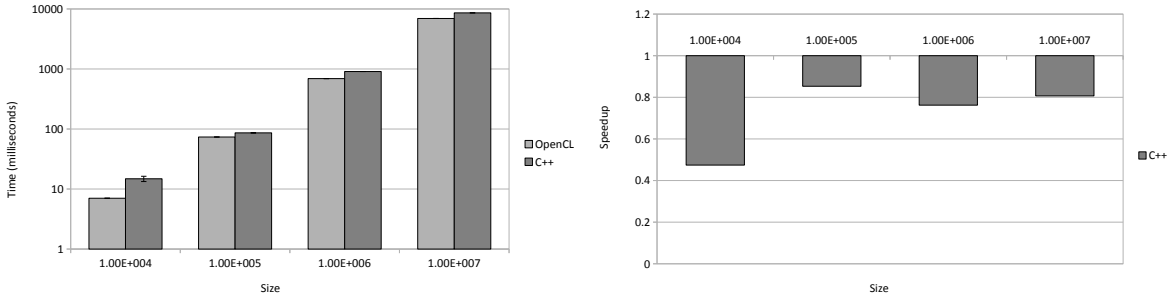
(a) Execution times (b) Speedup relative to OpenCL

Figure 5.11: Solarize benchmark between OpenCL and the optimized C++ version on Xeon Phi

The last test performed, with the Series benchmark, had a different outcome from the other tests since this application is computation bound, unlike the previous benchmarks. Although this benchmark registered the better speedup, of 0.5 for the smaller input size, the tendency for having a better performance each time the input size is increased was not observed. This time, the speedup stabilized at a value of 0.8 relatively to OpenCL.

5.3.2 Conclusions

From the breakdowns of the multiple stages of an execution, it can be observed that the computation time in C++ has a worst performance than with OpenCL. This can be related to the fact that the examples used were very data-centric, the type of application OpenCL is tailored for, thus taking better advantage of the hardware than the C++ version. Another important fact is how the orchestration is performed in C++, which resorts heavily to function pointers and function calls that are well known to be computationally heavy. Relatively to the type of application, the remaining tests clearly show that, for the computationally bound, the increase of the input size is not enough for C++ to take over the performance of OpenCL, because stage where OpenCL performs better, the execution



(a) Execution times (b) Speedup relative to OpenCL

Figure 5.12: Series benchmark between OpenCL and the optimized C++ version on Xeon Phi

stage, is the dominant one, as the Series benchmark proves. For communication bound applications, the type of the other benchmarks, it is clearly shown that the sizes of the problems to be solved should be large enough so that the time spent transferring data from the host to the data influences the overall execution time.

6

Conclusion

With this thesis we provide two functional prototypes of an algorithmic skeleton framework to construct and execute complex computations on Intel's Xeon Phi. The first prototype can be used to perform OpenCL computations using all the skeleton provided by previous versions of the Marrow framework. The second prototype has some limitations but allows not only the execution of C++ kernels in parallel over a structure of data but also to chain multiple kernels in order to form a pipeline. Both were tested using some well known data and task parallel examples to assess its performance. In addition to this, an very initial support for streaming data was implemented.

These prototypes support the viability of the Xeon Phi as a tool for performing data intense computing. The second one, specially, demonstrates that it is possible to provide an high level programming tool, that can still efficiently execute on the co-processor, for applications like the execution of a single function to each elements of a structure, or the application of a filter to an image. Also, it can be concluded that building such tools using only C++ and a low level library like OpenMP, that allows some fine tuning of the parallelism, is much better to take advantage of the hardware when compared with a technology like OpenCL. Some of the main challenges encountered during the course of this thesis were the incapability of Intel's compiler to support some fundamental features, like the offloading of classes to co-processor, and, due to the co-processor's youth, the lack of support when dealing with some errors that happened.

6.1 Future Work

In order to offer a fully functional C++ framework, the remaining skeletons from the previous version of Marrow could be implemented. Also, to take advantage of the hardware, the `Pipeline` skeleton could be modified to concurrently execute both stages on the co-processor and then, even implement an automatic load balancer that could dynamically assign more cores to a more computational intensive stage. In addiction, the support for the application of structured computations to continuous streams of information, similarly to Spark streaming [Zah+13], could be added to the framework.

Regarding some optimizations that could be made, the template feature could be used to prevent the need to pass function pointers and deal with `void` pointers to pass arguments to the various skeletons. Furthermore, the reuse of already allocated memory for buffers of equal or smaller size could be performed to prevent the need to re-allocate new memory.

Bibliography

[Abd+13] D. Abdurachmanov, K. Arya, J. Bendavid, T. Boccali, G. Cooperman, A. Dotti, P. Elmer, G. Eulisse, F. Giacomini, C. D. Jones, et al. "Explorations of the Viability of ARM and Xeon Phi for Physics Processing". In: *arXiv preprint arXiv:1311.1001* (2013).

[AMD] I. Advanced Micro Devices. *AMD Accelerated Processing Units*. URL: http://www.amd.com/us/products/technologies/apu/Pages/apu.aspx (visited on 02/06/2014).

[AMP13] F. Alexandre, R. Marques, and H. Paulino. "Esqueletos Algorítmicos para Paralelismo de Tarefas em Sistemas Multi-GPU". In: *INForum 2013 Atas do 5º Simpósio de Informática*. Ed. by J. Cachopo and B. Santos. 2013, pp. 238–249. ISBN: 978-989-97060-8-8. URL: http://inforum.org.pt/INForum2013/docs/Atas_do_INForum2013.pdf.

[AMP14] F. Alexandre, R. Marques, and H. Paulino. "On the Support of Task-Parallel Algorithmic Skeletons for Multi-GPU Computing". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '14, Gyeongju, South Korea, March 24-28, 2014*. ACM, Mar. 2014.

[BME07] A. Basumallik, S.-J. Min, and R. Eigenmann. "Programming Distributed Memory Sytems Using OpenMP". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. Mar. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370397.

[BL99] R. D. Blumofe and C. E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748. ISSN: 0004-5411. DOI: 10.1145/324133.324234. URL: http://doi.acm.org/10.1145/324133.324234.

[Blu+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *Proceedings*

of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216. ISBN: 0-89791-700-6. DOI: 10.1145/209936.209958. URL: http://doi.acm.org/10.1145/209936.209958.

[Cla+13] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. "New system software for parallel programming models on the Intel SCC many-core processor". In: *Concurrency and Computation: Practice and Experience* (2013), n/a–n/a. ISSN: 1532-0634. DOI: 10.1002/cpe.3033. URL: http://dx.doi.org/10.1002/cpe.3033.

[Col13] Colfax International. *Parallel Programming and Optimization with Intel® Xeon Phi™ Coprocessors*. Mar. 2013. ISBN: 978-0-9885234-1-8.

[Dok+13] J. Dokulil, E. Bajrovic, S. Benkner, M. Sandrieser, and B. Bachmayer. "HyPHI - Task Based Hybrid Execution C++ Library for the Intel Xeon Phi Coprocessor". In: *Parallel Processing (ICPP), 2013 42nd International Conference on*. Oct. 2013, pp. 280–289. DOI: 10.1109/ICPP.2013.37.

[For12] M. P. I. Forum. *MPI: A Message-Passing Interface Standard Version 3.0*. Sept. 2012. URL: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[GH13] S. R. Garea and T. Hoefler. "Modelling Communications in Cache Coherent Systems". In: (Mar. 2013).

[Hem13] N. Hemsoth. *Saddling Phi for TACC's Stampede*. May 2013. URL: http://archive.hpcwire.com/hpcwire/2013-05-17/saddling_phi_for_tacc%E2%80%99s_stampede.html (visited on 02/06/2014).

[Int] Intel®. *Intel® Xeon Phi™ Product Family: Product Brief*. URL: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html (visited on 02/06/2014).

[Int13a] Intel®. *Intel® C++ Compiler XE 13.1 User and Reference Guide*. 2013. URL: http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-lin/index.htm.

[Int13b] Intel®. *Intel® Cilk™ Plus Language Extension Specification Version 1.2*. Sept. 2013. URL: https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[Int13c] Intel®. *Intel® Threading Building Blocks Documentation*. Sept. 2013. URL: https://www.threadingbuildingblocks.org/docs/help/index.htm.

[JR13] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann, Mar. 2013. ISBN: 978-0-12-410414-3.

[Joh+13] J. Johnson, S. J. Krieder, B. Grimmer, J. M. Wozniak, M. Wilde, and I. Raicu. "Understanding the Costs of Many-Task Computing Workloads on Intel Xeon Phi Coprocessors". In: *2nd Greater Chicago Area System Research Workshop (GCASR)*. 2013.

[Khr13] Khronos OpenCL Working Group. *The OpenCL Specification Version 2.0*. Nov. 2013. URL: http://www.khronos.org/registry/cl/specs/opencl-2.0.pdf.

[Kis14] Kishonti Ltd. *Compare OpenCL Performance of: NVIDIA® GeForce® GTX TITAN vs Intel® Xeon Phi™ Coprocessor 5110p*. 2014. URL: http://compubench.com/compare.jsp?config_0=14470292&config_1=15887974 (visited on 01/26/2014).

[Lei09] C. E. Leiserson. "The Cilk++ Concurrency Platform". In: *Proceedings of the 46th Annual Design Automation Conference*. DAC '09. New York, NY, USA: ACM, 2009, pp. 522–527. ISBN: 978-1-60558-497-3. DOI: 10.1145/1629911.1630048. URL: http://doi.acm.org/10.1145/1629911.1630048.

[Lim+13] J. Lima, F. Broquedis, T. Gautier, and B. Raffin. "Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor". In: *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*. Oct. 2013, pp. 105–112. DOI: 10.1109/SBAC-PAD.2013.28.

[Lu+13] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. S. M. Goh, and R. Huynh. "Optimizing the MapReduce Framework on Intel Xeon Phi Coprocessor". In: *Big Data, 2013 IEEE International Conference on*. IEEE. 2013, pp. 125–130.

[Mar+13] R. Marques, H. Paulino, F. Alexandre, and P. Medeiros. "Algorithmic Skeleton Framework for the Orchestration of GPU Computations". In: *Euro-Par 2013 Parallel Processing*. Ed. by F. Wolf, B. Mohr, and D. Mey. Vol. 8097. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 874–885. ISBN: 978-3-642-40046-9. DOI: 10.1007/978-3-642-40047-6_86. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_86.

[MRR12] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann, June 2012. ISBN: 978-0-12-415993-8.

[Men+13] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. "Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede". In: *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*. XSEDE '13. New York, NY, USA: ACM, 2013, 48:1–48:8. ISBN: 978-1-4503-2170-9. DOI: 10.1145/2484762.2484779. URL: http://doi.acm.org/10.1145/2484762.2484779.

[Nvi] Nvidia. *JUST THE FACTS*. URL: http://www.nvidia.com/object/justthefacts.html (visited on 02/06/2014).

[Ope13a] OpenACC. *The OpenACC™ Application Programming Interface version 2.0*. June 2013. URL: http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[Ope11] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.1*. July 2011. URL: http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[Ope13b] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4.0*. July 2013. URL: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[PP84] M. S. Papamarcos and J. H. Patel. "A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories". In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. ISCA '84. New York, NY, USA: ACM, 1984, pp. 348–354. ISBN: 0-8186-0538-3. DOI: 10.1145/800015.808204. URL: http://doi.acm.org/10.1145/800015.808204.

[PM12] M. Pharr and W. Mark. "ispc: A SPMD compiler for high-performance CPU programming". In: *Innovative Parallel Computing (InPar), 2012*. May 2012, pp. 1–13. DOI: 10.1109/InPar.2012.6339601.

[Pot+] S. Potluri, K. Hamidouche, D. Bureddy, and D. K. D. Panda. "MVAPICH2-MIC: A High Performance MPI Library for Xeon Phi Clusters with InfiniBand". In: ().

[Rah13] R. Rahman. *Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Sept. 2013. ISBN: 978-1-43-025926-8.

[Sch+13] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. Müller. "Assessing the Performance of OpenMP Programs on the Intel Xeon Phi". In: *Euro-Par 2013 Parallel Processing*. Ed. by F. Wolf, B. Mohr, and D. Mey. Vol. 8097. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 547–558. ISBN: 978-3-642-40046-9. DOI: 10.1007/978-3-642-40047-6_56. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_56.

[Sei+08] L. Seiler et al. "Larrabee: A Many-core x86 Architecture for Visual Computing". In: *ACM Trans. Graph.* 27.3 (Aug. 2008), 18:1–18:15. ISSN: 0730-0301. DOI: 10.1145/1360612.1360617. URL: http://doi.acm.org/10.1145/1360612.1360617.

[SJL11] S. Seo, G. Jo, and J. Lee. "Performance characterization of the NAS Parallel Benchmarks in OpenCL". In: *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. 2011, pp. 137–148. DOI: 10.1109/IISWC.2011.6114174.

[SAP14] F. Soldado, F. Alexandre, and H. Paulino. "Towards the Transparent Execution of Compound OpenCL Computations in Multi-CPU/Multi-GPU Environments". In: *Euro-Par 2014: Parallel Processing Workshops Part I*. Vol. 8805. Lecture Notes in Computer Science. Springer-Verlag, Dec. 2014, pp. 177–189.

[Wie+12] S. Wienke, P. Springer, C. Terboven, and D. an Mey. "OpenACC: First Experiences with Real-world Applications". In: *Proceedings of the 18th International Conference on Parallel Processing*. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870. ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_85. URL: http://dx.doi.org/10.1007/978-3-642-32820-6_85.

[Zah+13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737.

Examples

A.1 For-loop parallelism in OpenMP

An OpemMP example of the use of the clauses `schedule` and `reduction` is shown in Listing A.1. The `schedule` clause has 2 arguments, the type of scheduling which in this case is `dynamic` and the size of each chunk of iterations which is 10. The `dynamic` scheduling ensures that each thread will receive a balanced number of chunks, which may be bad in the situations where the iterations have an unbalanced amount of work required which can lead to some threads doing much more work than others. The chunk size is simply to state that all the iterations will be grouped into chunks of 10.

The `reduction` clause has the form `operator:list` in which the operator is the reduce function to be used and the list has the variables to which the function will be applied. The variables that appear in `list` must be shared among all threads in order to be possible to access its multiple values and apply the reduce function. The particular example of Listing A.1, since the final result is the sum of all multiplications the operator is + and the variable to which it is applied is the `result` variable.

A.2 Fork-Join parallelism in OpenMP

The example in Listing A.2 shows the usage of tasks to spawn multiple tasks, in this case 10, to execute the enclosed computation and subsequently print out the result. The creation of new tasks is achieved by the use of the `#pragma omp task` directive to state that the call of the recursive method `Recurse` is to be made by a new thread.

Listing A.1: Example of vector dot product with OpenMP

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char* argv[]){
  int i, n, chunk;
  n = 1000;
  chunk = 10;
  int a[n], b[n], result, result_check;
  result = result_check = 0.0;

  /* Some initializations */
  for (i=0; i < n; i++){
    a[i] = i; b[i] = i * 2;
  }

#pragma omp parallel for  \
  schedule(static,chunk)  \
  reduction(+:result)
  for (i=0; i < n; i++)
    result = result + (a[i] * b[i]);

  for(i = 0; i < n; i++)
    result_check += (a[i] * b[i]);
  printf("Final result = %d (should be %d)\n",result, result_check);
  return 0;
}
```

Listing A.2: Example of recursive parallelization with OpenMP[Col13]

```
#include <omp.h>
#include <stdio.h>

void Recurse(const int task_) {
  if (task_ < 10) {
    printf("Creating task %d...", task_+1);
#pragma omp task
    {
      Recurse(task_+1);
    }
    unsigned long foo=0; for (long i = 0; i < (1<<16); i++) foo+=i;
    printf("result of task %d in thread %d is %ld\n", task_, omp_get_thread_num
        (), foo);
  }
}

int main(int argc, char* argv[]) {
#pragma omp parallel
  {
#pragma omp single
    Recurse(0);
  }
  return 0;
}
```

Listing A.3: Example of vector dot product with array annotation [MRR12]

```
float cilkplus_sprod(
  size_t n,
  const float a[],
  const float b[] ) {
  return __sec_reduce_add(a[0:n] * b[0:n]);
}
```

Listing A.4: Example of vector dot product with `cilk_for` [MRR12]

```
float cilkplus_sprod(
  size_t n,
  const float a[],
  const float b[] ) {
  cilk::reducer_opadd<float> res(0);
  cilk_for (size_t i = 0; i < n; i++){
    res += a[i] * b[i];
  }
  return res.get_value();
}
```

One detail to be taken into account is how the first call is made. As depicted in Listing A.2 the `Recurse(0)` call is placed inside a `#pragma omp single` block that is in turn inside a `#pragma omp parallel` block. This is usual when using the Fork-Join model because the execution needs to be made by several threads in a parallel environment, hence the `parallel` block, but only one thread needs to run the first call. If the `parallel` block is missing, all tasks are executed by the master thread and if the `single` block is missing, all threads in the team will start with `Recurse(0)` making the tasks being run multiple times, which is not the desired behavior.

A.3 For-Loop parallelism in Cilk™ Plus

In Listing A.3 is shown an example of how a function that uses array annotations and reducers. In this case, the reducer used is sum all the results of multiplying the corresponding values for two arrays.

There are two ways to use the reducers, either using a function to reduce a section, as shown in Listing A.3, or by using a reducer variable that is used to aggregate the results of parallel loop iterations, as shown in Listing A.4.

Listing A.5 shows the declaration and usage of elemental functions.

A.4 Fork-Join parallelism in Cilk™

The original programming model for which Cilk was developed is the Fork-Join parallelism since in the first specification only the `cilk_spawn` and `cilk_sync` keywords were defined and the execution model is based on work-stealing schedule so the loop

Listing A.5: Example of use of `#pragma simd` and elemental functions [MRR12]

```
__declspec(vector(linear(a),uniform(b)))
void bar(float* a, float* b, int c, int d);

void foo(float* a, float* b, int* c, int* d, int n){
#pragma simd
  for( int i=0; i<n; ++i )
    bar( a+i, b, c[i], d[i] );
}
```

Listing A.6: Example of recursive parallelization with Cilk [Col13]

```
#include <stdio.h>
#include <cilk/cilk.h>

void Recurse(const int task) {
  if (task < 10) {
    printf("Creating task %d...", task+1);
    cilk_spawn Recurse(task+1);
    long foo=0; for (long i = 0; i < (1L<<20L); i++) foo+=i;
    printf("result of task %d in worker %d is %ld\n", task,
      __cilkrts_get_worker_number(), foo);
  }
}

int main() {
  Recurse(0);
}
```

parallelism is based on spawning multiple threads and then create a queue from where the threads steal work to do.

This model is very simple to use in Cilk, as shown in Listing A.6. To execute a function in a different thread in parallel simply precede the function call with the keyword `cilk_spawn`. This can be used not only in functions that return `void` but also on an assignment to some variable. The assignment occurs when the function returns so, after spawning the function, if the caller uses the variable it implicitly waits for the result to be available. The `cilk_sync` keyword is used to explicitly make the thread wait for all spawned functions in the spawning block (function, `try/catch` or `cilk_for`). An implicit sync is made at the end of each spawning block, only waiting for functions spawned within the block.

The Cilk programming model is tailored for task parallelism, while OpenMP is specially directed for data parallelism. Hence, to express fork-join computations in Cilk is much more simpler than in OpenMP. That can be confirmed by comparing the programming examples depicted Listing A.2 and Listing A.6. A common practice in fork-join parallelism is to allow the main thread to also contribute in the execution of the spawned tasks and, with that, the main thread is not blocked waiting for all spawned functions but also is one less thread that needs to be spawned.

Listing A.7: Example of vector dot product in TBB

```
size_t N = 1024;
size_t STRIDE = 20;
float a[N];
float b[N];
atomic<int> res;

void dot_range( const blocked_range<size_t> &r ){
     for( size_t i = r.begin(); i != r.end(); ++i )
         res += a[i]*b[i];
}

void dot_array(){
  parallel_for(blocked_range<size_t>(0, N, STRIDE), dot_range,
       auto_partitioner());
}
```

A.5 For-Loop parallelism in Intel® TBB

Listing A.7 shows an example of usage of the `parallel_for` routine with an atomic variable to avoid a data race between threads. The `parallel_for` shown has three arguments, the range of the for loop, the function to be applied to each sub-range and the partitioner to be used. The range is defined by the initial value, the limit of the range exclusively and, optionally, the minimum size of each sub-range, which is 1 by default if the value is not specified. The function used can have only one argument that is of type range, like in the example, representing the sub-range to be processed or the index of the position to be processed. The way the range is divided in sub-ranges, is defined by the partitioner used, which can be an `auto_partitioner`, that will try to divide the range so that the amount of work of all threads is balanced, a `simple_partitioner`, that will divide the range until the minimum size is achieved, or an `affinity_partitioner`, that will try to assign to each thread a similar sub-range of some previous parallel construct.

A.6 Flow graph in Intel® TBB 4.0

In Listing A.8 is demonstrated the code for a graph that consists in an input node, that feeds the values, a squarer and cuber nodes, that receive a value from the input node and apply the corresponding functions, a join node, that takes the result from the squarer and cuber nodes and produces a tuple with the values, and a final node summer that sums the values of the tuples.

Listing A.8: Example of a flow graph in TBB[Int13c]

```
int result = 0;
graph g;
broadcast_node<int> input(g);
function_node<int,int> squarer( g, unlimited, square() );
function_node<int,int> cuber( g, unlimited, cube() );
join_node< tuple<int,int>, queueing > join( g );
function_node<tuple<int,int>,int> summer( g, serial, sum(result) );

make_edge( input, squarer );
make_edge( input, cuber );
make_edge( squarer, get<0>( join.input_ports() ) );
make_edge( cuber, get<1>( join.input_ports() ) );
make_edge( join, summer );

for (int i = 1; i <= 10; ++i)
  input.try_put(i);
g.wait_for_all();
```

Listing A.9: Kernel function in OpenCL to calculate the addition of two arrays

```
__kernel void arr_add(
  __global const float *a,
  __global const float *b,
  __global float *c,
  const unsigned int nelems)
{
  int id = get_global_id(0);
  if(id < nelems)
    c[id] = a[id] * b[id];
}
```

A.7 OpenCL™

An example of a kernel function is shown in Listing A.9. All kernel functions must be preceded by a `__kernel` keyword. As previously mentioned the `get_global_id` routine is used to get the index of the arrays of each work item. The keyword `__global` states that the arguments' address space is global to all work items. The other address spaces available are `__private` for the address space local to each work item, `__local` for the address space shared by all work items in a work group and `__constant` for read-only arguments in the global address space, shared by all work items.

An OpenCL application can usually be split into five sections: initialization, allocation of resources, creation of programs/kernels, execution and finalization. Listing A.10 illustrates all these sections, that we next detail.

Lines 2 to 8 show the creation of the context for the device, which can be of an specific type by using the `clGetDeviceIDs` routine with the desired type, along with the corresponding command queue using `clCreateCommandQueue`.

In the allocation phase (lines 10 to 13) the buffer and image objects are created and the command to initiate them is enqueued with `clEnqueueWriteBuffer`. This write can be blocking or non-blocking depending on the value passed as the third argument. In this case the write is blocking so after the call return the pointer `ax` can be safely used.

A program is a set of kernels and functions like the one in Listing A.9. A program must be loaded and built using the `clCreateProgramWithSource` and `clBuildProgram` routines. After the program is built, the kernel defined can be obtained for execution using `clCreateKernel` routine, as demonstrated in lines 15 to 20 of Listing A.10.

Before issuing the command to execute a kernel the argument must be set using the `clSetKernelArg` routine. The enqueueing of the kernel is made through `clEnqueueTask` (Listing A.10, lines 22 to 24).

Listing A.10: Example of an OpenCL application

```
1  //Initialization
2  cl_int err;
3  cl_context context;
4  cl_device_id devices;
5  cl_command_queue cmd_queue;
6  err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU, 1, &devices, NULL);
7  context = clCreateContext(0, 1, &devices, NULL, NULL, &err);
8  cmd_queue = clCreateCommandQueue(context, devices, 0, NULL);
9  //Allocation
10 cl_mem ax_mem = clCreateBuffer(context, CL_MEM_READ_ONLY,
11     atom_buffer_size, NULL, NULL);
12 err = clEnqueueWriteBuffer(cmd_queue, ax_mem, CL_TRUE, 0,
13     atom_buffer_size, (void*)ax, 0,NULL,NULL);
14 //Program/Kernel creation
15 cl_program program[1];
16 cl_kernel kernel[1];
17 program[0] = clCreateProgramWithSource(context,1,
18     (const char**)&program_source, NULL, &err);
19 err = clBuildProgram(program[0], 0, NULL, NULL, NULL, NULL);
20 kernel[0] = clCreateKernel(program[0], "mdh", &err);
21 //Execution
22 err = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &ax_mem);
23 err = clEnqueueTask(cmd_queue, kernel[0],
24     0, NULL, NULL);
25 //Finalization
26 err = clEnqueueReadBuffer(cmd_queue, val_mem, CL_TRUE, 0,
27     grid_buffer_size, val, 0, NULL, NULL);
28 clReleaseKernel(kernel);
29 clReleaseProgram(program);
30 clReleaseCommandQueue(cmd_queue);
31 clReleaseContext(context);
```

After the computation, the results can be retrieved from the buffer using the `clEnqueueReadBuffer` routine (Listing A.10, lines 26 and 27). The reading from a buffer can be made in a blocking way to guarantee that the data will be available after the call to the `clEnqueueReadBuffer` routine using the `CL_TRUE` flag. To finish the execution, the resources are released with the routines `clReleaseKernel`, `clReleaseProgram`, `clReleaseCommandQueue` and `clReleaseContext`.

A.8 Intel® Compiler pragmas

Listing A.11 presents the usage of the `offload` directive to perform some computation concurrently, with the copy of `f1` and allocation of `f2`, and then the `offload_transfer` directive to wait for the result in `f2` to became available.

Listing A.11: Example of asynchronous computation [Int13a]

```
const int N = 4086;
float *f1, *f2;
f1 = (float *)memalign(64, N*sizeof(float));
f2 = (float *)memalign(64, N*sizeof(float));

// CPU sends f1 as input synchronously
// The output is in f2, but is not needed immediately
#pragma offload target (mic:0) \
          in(  f1 : length(N) ) \
          nocopy( f2 : length(N) free_if(0)) signal(f2)
{
  foo(N, f1, f2);
}

// Perform some computation asynchronously

#pragma offload_transfer (mic:0) wait(f2) \
            out( f2 : length(N) alloc_if(0) free_if(1))
// CPU can now use the result in f2
```

A.9 MYO Memory Model

An example of the dynamic allocation of shared classes is depicted in Listing A.12.

Listing A.12: Example of allocation of memory for a shared class [Col13]

```
#include <stdio.h>
#include <stdlib.h>
#include <new>

class _Cilk_shared MyClass {
    int i;
  public:
    MyClass(){ i = 0; };
    void set(const int l) { i = l; }
    void print(){
      printf("Value: %d\n", i);
    }
};

MyClass* _Cilk_shared sharedData;

int main(){
  const int size = sizeof(MyClass);
  _Cilk_shared MyClass* address = (_Cilk_shared MyClass*)
      _Offload_shared_malloc(size);
  sharedData=new( address ) MyClass;
  sharedData->set(1000); // Shared data initialized on host
  _Cilk_offload sharedData->print(); // Shared data used on coprocessor
  sharedData->print(); // Shared data used on host
}
```