Developing and Evaluating clOpenCL Applications for Heterogeneous Clusters

Tiago Filipe Rodrigues Ribeiro

Relatório Final do Trabalho de Projecto apresentado à

Escola Superior de Tecnologia e de Gestão

Instituto Politécnico de Bragança

para obtenção do grau de Mestre em

Sistemas de Informação

Novembro 2012

Developing and Evaluating clOpenCL Applications for Heterogeneous Clusters

Tiago Filipe Rodrigues Ribeiro

Relatório Final do Trabalho de Projecto apresentado à

Escola Superior de Tecnologia e de Gestão

Instituto Politécnico de Bragança

para obtenção do grau de Mestre em

Sistemas de Informação

Orientador:

Prof. Dr. José Rufino

"Este Trabalho de Projecto não inclui as críticas e sugestões feitas pelo Júri."

Novembro 2012

Acknowledgements

I am grateful to Professor José Rufino, my supervisor, for all the support, dedication, patience and aid during this project. Without his critical spirit and constant motivation this project would not have been possible.
To my mother and father, who throughout my academic career and especially at this late stage, remained always supporters and motivators, for me to successfully overcome all the barriers. Mum, I know that, with all our strength we will overcome it all.
A reminder of my grandfather, who unfortunately, cannot witness my academic success, but in all my heart I know he is very pleased with me.
To my remaining family I appreciate the interest and support shown.
A special thanks to my cousins Ricardo and Tiago for their friendship, help, support and motivation, through difficult and needed times.
To my colleague Mário, who provided a great work environment and stayed with me all the scholarship period. To my other friends for their goodwill and friendship that always presented me with.
I also thank the Polytechnic Institute of Bragança for the conditions provided for carrying out this project.

To all my thanks.

We did it...

Abstract

In the last few years, the computing systems processing capabilities have increased significantly, changing from single-core to multi-core and even many-core systems. Accompanying this evolution, local networks have also become faster, with multi-gigabit technologies like Infiniband, Myrinet and 10G Ethernet. Parallel/distributed programming tools and standards, like POSIX Threads, OpenMP and MPI, have helped to explore these technologies and have been frequently combined, giving rise to Hybrid Programming Models.

Recently, co-processors like GPUs and FPGAs, started to be used as accelerators, requiring specialized frameworks (like CUDA for NVIDIA GPUs). Presented with so much heterogeneity, the industry formulated the OpenCL specification, as a standard to explore heterogeneous systems. However, in the context of cluster computing, one problem surfaces: OpenCL only enables a developer to use the devices that are present in the local machine. With many processor devices scattered across cluster nodes (CPUs, GPUs and other co-processors), it then became important to enable software developers to take full advantage of the full cluster device set.

This dissertation demonstrates and evaluates an OpenCL extension, named clOpenCL, which supports the simple deployment and efficient running of OpenCL-based parallel applications that may span several cluster nodes, thus expanding the original single-node OpenCL model. The main contributions are that clOpenCL i) offers a transparent approach to the porting of traditional OpenCL applications to cluster environments and ii) provides significant performance increases over classical (non-)hybrid parallel approaches.

Keywords: Hybrid Programming, Heterogeneous Computing, High-Performance Computing, MPI, OpenCL, clOpenCL.

Resumo

Nos últimos anos , a capacidade de processamento dos sistemas de computação aumentou significativamente, passando de CPUs com um núcleo para CPUs multi-núcleo. Acompanhando esta evolução, as redes locais também se tornaram mais rápidas, com tecnologias *multi-gigabit* como a *Infiniband*, *Myrinet* e *10G Ethernet*. Ferramentas e *standards* paralelos/distribuídos, como *POSIX Threads*, OpenMP e MPI, ajudaram a explorar esses sistemas, e têm sido frequentemente combinados dando origem a Modelos de Programação Híbrida.

Mais recentemente, co-processadores como GPUs e FPGAs, começaram a ser utilizados como aceleradores, exigindo *frameworks* especializadas (como o CUDA para GPUs NVIDIA). Deparada com tanta heterogeneidade, a indústria formulou a especificação OpenCL, como sendo um *standard* para exploração de sistemas heterogéneos. No entanto, no contexto da computação em *cluster*, um problema surge: o OpenCL só permite ao desenvolvedor utilizar dispositivos presentes na máquina local. Com tantos dispositivos de processamento espalhados pelos nós de um *cluster* (CPUs, GPUs e outros co-processadores), tornou-se assim importante habilitar os desenvolvedores de *software*, a tirarem o máximo proveito do conjunto total de dispositivos do *cluster*.

Esta dissertação demonstra e avalia uma extensão OpenCL, chamada clOpenCL, que suporta a implementação simples e execução eficiente de aplicações paralelas baseadas em OpenCL que podem estender-se por vários nós do *cluster*, expandindo assim o modelo original de um único nó do OpenCL. As principais contribuições referem-se a que o clOpenCL i) oferece uma abordagem transparente à portabilidade de aplicações OpenCL tradicionais para ambientes cluster e ii) proporciona aumentos significativos de desempenho sobre abordagens paralelas clássicas (não-)híbridas.

Palavras-chave: Programação Híbrida, Computação Heterogénea, Computação de Alto Desempenho, MPI, OpenCL, clOpenCL.

Contents

Acknowledgements **v**

Abstract **vii**

Resumo **ix**

List of Acronyms **xix**

1 Introduction **1**

1.1 Context 2

1.2 Objectives & Contributions 3

1.3 Dissertation Structure 4

2 Concepts and Technologies **5**

2.1 Parallel Computing Fundamental Concepts 5

2.1.1 Sequential Execution versus Parallel Execution 5

2.1.2 Scalability and Acceleration 7

2.1.3 Amdahl's Law and Gustafson's Law 8

2.2 Parallel Computing Platforms 10
2.2.1 SMP, Multi-Core and Many-Core Systems 10
2.2.2 Clusters 12
2.2.3 Grids/Clouds 13
2.2.4 Heterogeneous Systems 13
2.3 Parallel Programming Models and Frameworks 15
2.3.1 Shared Memory and Threads 15
2.3.2 Message Passing 16
2.3.3 Heterogeneous Systems 19
2.3.4 Hybrid Models 25
2.4 Cluster OpenCL 26
2.4.1 General Architecture 26
2.4.2 Operation Model 27
2.4.3 Using clOpenCL 28
2.4.4 Distributed OpenCL 29
2.5 Case Study: Matrix Product 29
2.6 Experimental Testbed 30
2.6.1 Computational Systems 31
2.6.2 Network(s) 32
2.6.3 Exploitation System 33

3 Preliminary Experiments **35**

3.1 General Experimental Conditions 35
3.2 Sequential Naive Approach 36
3.3 BLAS Approaches 37
3.4 ATLAS BLAS 39
3.5 ACML BLAS 40
3.6 GSL BLAS 41
3.7 Matlab 42
3.8 Preliminary Results Analysis 46

4 Parallel Approaches **49**

4.1 Parallelization Strategy 49
4.2 MPI-Only 51
4.2.1 Test Deployment 56
4.2.2 Memory Issues 56
4.2.3 Using MMAP 57
4.2.4 Evaluation Results 59
4.3 MPI-with-OpenCL 59
4.3.1 Test Deployment 64
4.3.2 Memory Issues 64
4.3.3 Evaluation Results 64
4.4 clOpenCL 68
4.4.1 Test Deployment 72

4.4.2 No Memory Issues . . . 72
4.4.3 Evaluation Results . . . 72
4.5 Results Discussion . . . 74

5 Conclusions **77**
5.1 Future Work . . . 78

Bibliography **79**

A Source Code **87**
A.1 Preliminary Experiments . . . 87
A.2 Parallel Approaches . . . 87

B OpenCL Details **89**
B.1 OpenCL Terminology . . . 89
B.1.1 OpenCL Standard . . . 91
B.1.2 OpenCL Specification . . . 91
B.1.3 Framework . . . 96
B.2 OpenCL API Supported Data Types . . . 96

C Scientific Contributions **99**
C.1 Published Paper . . . 99

List of Tables

3.1 Sequential Naive Approach Execution Time (seconds). 37

3.2 ATLAS BLAS Execution Time (seconds). 40

3.3 ACML BLAS Evaluation Execution Time (seconds). 41

3.4 GSL BLAS Evaluation Execution Time (seconds). 42

3.5 Matlab Matrix Product Results (seconds). 44

4.1 Number of C sub-matrices. 51

4.2 Memory Consumption for $n = 24K$ and $slice = 2K$ (Gb). 57

4.3 MPI-Only Evaluation Results. 59

4.4 MPI-with-OpenCL Evaluation Results – naive kernel. 66

4.5 MPI-with-OpenCL Evaluation Results – optimized kernel. 68

4.6 clOpenCL Evaluation Results – naive kernel. 73

4.7 clOpenCL Evaluation Results – optimized kernel. 74

B.1 Built-in Scalar Data Types. 97

List of Figures

2.1 Representation of the sequential execution model. 6

2.2 Representation of the parallel execution model. 6

2.3 Optimal Scalability (linear), limited by Amdahl's Law [Bri12]. 7

2.4 Speedup according to Amdahl's Law, for different values of P and N [Bar11a]. 9

2.5 SMP system representation. 10

2.6 Multi-core System representation. 11

2.7 A Cluster system representation. 12

2.8 A simplified view of the Message Passing Model. 17

2.9 CUDA architecture. 20

2.10 clOpenCL (a) operation model, (b) host application layers. 27

2.11 Matrix multiplication representation. 30

2.12 Cluster network(s) configuration. 32

3.1 Example of a matrix transposition. 36

3.2 Matlab evaluation results graphic. 44

3.3 Speedups for all combinations. 45

3.4 Preliminary tests results comparison. 46

4.1 Sliced matrix product representation. 50

4.2 Master – Slave Architecture. 51

4.3 A specific sliced matrix product. 52

4.4 Master-Slave Architecture of the MPI-with-OpenCL approach. 60

4.5 A deployment of the clOpenCL Architecture. 69

4.6 Host application execution method. 70

4.7 Parallel Matrix Product – best execution times (s). 75

4.8 Parallel Matrix Product – speedups relative to MPI-Only. 75

4.9 Speedups relative to the ATLAS implementation. 76

B.1 OpenCL platform 92

B.2 Representation of the NDRange where Work-items (WI) are grouped in Work-groups (WG). 94

B.3 Memory Model representation. 95

List of Acronyms

AMD Advanced Micro Devices

API Application Programming Interfaces

APP Accelerated Parallel Processing

APU Accelerated Processing Unit

ASIC Application-Specific Integrated Circuit

ATLAS Automatically Tuned Linear Algebra Software

BLAS Basic Linear Algebra Subprograms

BSD Berkeley sockets

CU Compute Unit

C99 C Programming Language Standard

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DSP Digital Signal Processor

FPGA Field-Programmable Gate Array

GPP General Purpose Processors

GPU Graphics Processing Unit

GPGPU General-Purpose Graphics Processing Units

GVirtuS Generic Virtualization Service

HPC High-Performance Computing

IO Input Output

ISA Instruction Set Architecture

ISO International Organization for Standardization

JIT Just-In-Time

MPI Message Passing Interface

OS Operative System

OpenCL Open Computing Language

OpenMP Open Multi-Processing

PE Processing Element

PVM Parallel Virtual Machine

RAM Random Access Memory

SDK Software Development Kit

SMP Symetric MultiProcessing

SIMD Single Instruction, Multiple Data

SPMD Single Program, Multiple Data

SSH Secure-Shell

VCL Virtual OpenCL

Chapter 1

Introduction

In the recent years, the computing systems processing capabilities have increased significantly, changing from one core CPUs to multi-core and many-core ones. Accompanying this evolution, local networks have also become faster, using Myrinet, Infiniband and 10G-Ethernet technologies. Parallel/distributed programming tools and standards, like MPI (Message Passing Interface) and OpenMP, have helped to explore these technologies and have been frequently combined, giving rise to Hybrid Programming Models. All of these contributed to the growing importance of Parallel Computing in the modern computing landscape.

In this context, clusters are still the main approach to Parallel Computing, they are being used to solve problems that need large computational power and/or storage capacity, and for such problems, clusters have the right features: large storage capacity, using resources available throughout its nodes; increased parallel processing capabilities, using the distributed CPUs. However, to solve increasingly demanding computational problems, an improve in performance is needed. For CPUs, higher clock speeds are no longer the only answer and, in their small package, one way to increase performance is by adding multiple cores. Because of this, devices like GPUs, with dozens/hundreads of cores have become programmable parallel processors, evolving from fixed function rendering devices

[Gro11], and are being coupled with traditional CPUs, creating heterogeneous systems.

Presented with so much heterogeneity, the process of developing software for such a wide array of architectures poses a number of challenges to the programming community [GHK+11], because programming approaches for multi-core CPUs and GPUs are very different. Thus, new standards where introduced to help for this kind of programming, like the OpenCL and CUDA standards. However, in cluster computing, one problem surfaces: these standards only enable a developer to use the devices that are present in the local machine.

Once recent clusters included highly parallel CPUs, GPUs and other types of co-processors, it became important to enable software developers to take full advantage of these heterogeneous processing devices present across all nodes. In recent years different projects started to address this problem, taking CUDA or OpenCL as a starting point and creating extensions of these specifications.

This dissertation introduces and demonstrates an OpenCL extension, named clOpenCL, which supports the simple deployment and efficient running of OpenCL-based parallel applications that may span several cluster nodes, expanding the original single-node OpenCL model [ARPS12].

1.1 Context

This work was conducted as part of a scholarship during the PERFORM (*Portability and Performance in Heterogeneous Parallel Systems*) research project (ref. PTDC/EIA/100035/2008). The project was funded by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010067.

In the context of the PERFORM project, an extension of the OpenCL standard (section 2.3.3) has been developed. This extension, called clOpenCL (more on section 2.4), is a library and a set of utilities and support services, which sustains an OpenCL execution environment extended to a cluster. clOpenCL allows OpenCL applications to benefit from operating in a cluster, in a quickly and almost transparent way.

1.2 Objectives & Contributions

The main objective of this dissertation was to assess the merits of the current clOpenCL implementation, regarding to its compatibility with the original OpenCL standard, its ease of use by programmers and its stability and operational performance. In this context, the starting point of this work was the familiarization with the OpenCL programming model. Later, clOpenCL was used as the execution platform of an OpenCL application in the area of Numerical Methods – a Matrix Multiplication application. This application was considered to be sufficiently representative of a typical OpenCL usage scenario and allowed to exploit the unprecedented potentialities introduced by clOpenCL. To complement the clOpenCL evaluation, other parallel variants were developed, in order to be used as comparison baselines (MPI-Only and MPI-with-OpenCL approaches).

The objectives initially set for this dissertation were achieved. Thus, its main contributions are: i) the offer of a transparent approach to the porting of traditional OpenCL applications to cluster environments and ii) the ability to provide significant performance increases over classical (non-)hybrid parallel approaches. These contributions were validated by a paper submited and accepted in an international conference: "clOpenCL – Supporting Distributed Heterogeneous Computing in HPC Clusters" [ARPS12], see Appendix C.1.

1.3 Dissertation Structure

The remaining of this dissertation is organized as follows:

- Chapter 2 describes the main concepts related to the development of the current work and introduces the reader to the main frameworks used for the applications development, as well as the parallel test environment used.
- Chapter 3 describes preliminary tests and evaluations of the matrix multiplication algorithm, using well known sequential approaches and libraries.
- Chapter 4 describes in detail the important aspects about the implemented parallel approaches, the issues encountered during their development and the evaluation results. A discussion is also provided on the results obtained.
- Chapter 5 (last chapter) summarizes the main contributions of the work, and presents some ideas for future work.

The dissertation also includes the appendices: Appendix A, Appendix B and Appendix C, containing the references to the complete implemented code of the developed applications, complementary content and the scientific contributions.

Chapter 2

Concepts and Technologies

In this chapter, the developed work is contextualized, through an assessment of approaches and technologies related to the dissertation main themes. The core study of the dissertation is also introduced, as well as the experimental platform used.

2.1 Parallel Computing Fundamental Concepts

Parallel Computing is considered the pinnacle of modern computing. This form of computing has been used to attack complex problems in various areas of the Fundamental Sciences (Physics, Biotechnology, Genetics, Chemistry, Geology, Mechanics, Mathematics, etc.), Engineering (e.g., calculation and modelling of structures), and even more recently, Economy and Finance (stock market, datamining, etc.). For example, more accurate simulations, or simulations of larger problems, need large computational power and/or storage capacity, thus being typical targets for the use of Parallel Computing.

2.1.1 Sequential Execution versus Parallel Execution

Traditionally, most of the software has been developed on the assumption that it will be executed sequentially, i.e., on a single computer and requiring only a single processor. In

this perspective, the problem to be solved is subdivided into a series of instructions that are executed one after the other so, at a given time, only one instruction of the program is running. Figure 2.1 illustrates this (simplified) description of the Sequential Computing.

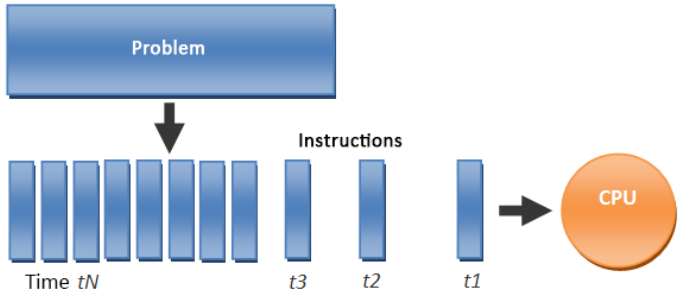

Figure 2.1: Representation of the sequential execution model.

In contrast, Parallel Computing includes the simultaneous use of multiple processors to solve, computationally, a particular problem, usually with the aim of accelerating its resolution. In this case, the problem addressed will be divided into sub-problems with a minimum possible of interdependencies, so that it can be resolved simultaneously, within specific tasks. Thus, the software must be parallelized, i.e. designed and developed, from the start, in order to be executed in parallel, by multiple processors. This vision is represented in Figure 2.2.

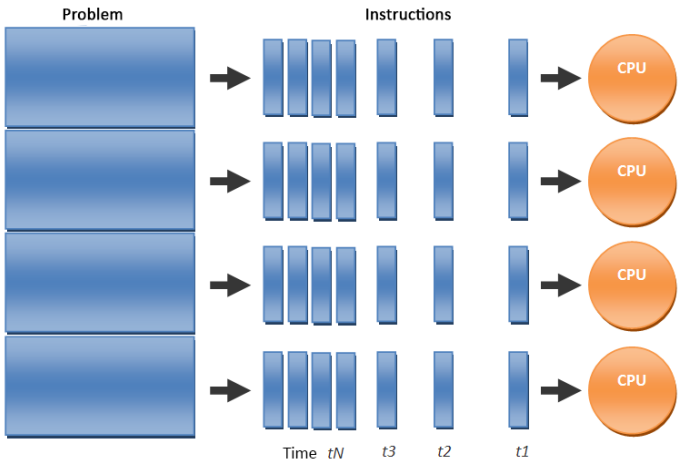

Figure 2.2: Representation of the parallel execution model.

2.1.2 Scalability and Acceleration

A very important concept in parallel computing is Scalability. Scalability refers to the ability a parallel system has to support a proportional increase (i.e., linear) of performance (or, equivalently, an inversely proportional decrease in execution time) with the addition of more processors.

Ideally, the acceleration (also known as speedup) allowed by the parallelization of an application should be linear, i.e., the increase in the number of processors that perform the application, from N to $N+1$, should imply a reduction in execution time from $T_N = T_1/N$ to $T_{N+1} = T_1/(N+1)$, where T_1 is the sequential time (time with only one processor). In this context, the optimum speedup with N processors, is given by:

$$S_N = \frac{T_1}{T_N}$$

Therefore, in an ideal situation $S_N = N$, since $T_N = T_1/N$. However, very few parallel algorithms achieve this optimum acceleration. Most of these algorithms have a quasi-linear acceleration, for reduced numbers of processors, which stabilizes and tends to decline even with a large number of processors [Lab11a]. Figure 2.3 illustrates this case.

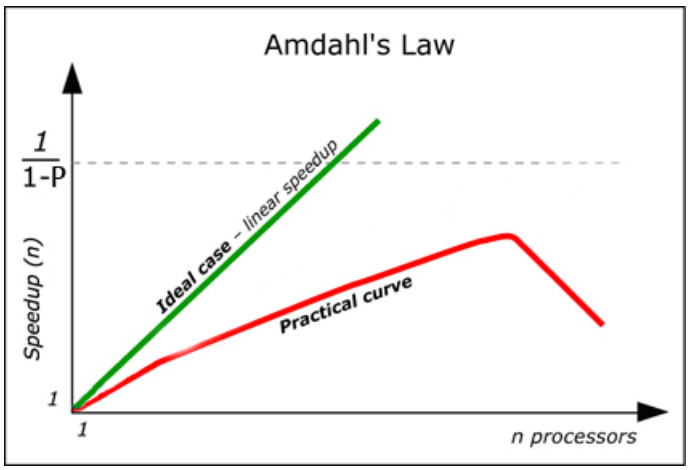

Figure 2.3: Optimal Scalability (linear), limited by Amdahl's Law [Bri12].

There are, however, some factors that can contribute to a good scalability, among which: i) the type of hardware used (particularly with regard to the bandwidth availability between the memory and the CPU, and of the local network); ii) the base algorithm and specific characteristics of the parallel application; iii) parallelization strategy and quality/expertise of the programmer.

2.1.3 Amdahl's Law and Gustafson's Law

The potential acceleration of an algorithm in a parallel computing platform is given by Amdahl's Law, formulated in 1960 by the computers architect Gene Amdahl. This law states that in any program there are typically one or more portions inherently not parallelizable, which will limit the ability to accelerate the implementation of the program through parallelization. In this context, if P is the fraction of the program that is parallelizable, the speedup with N processors becomes:

$$S_N = \frac{1}{(1-P) + P/N}$$

Again, if the entire program is parallelizable (ideal situation, with $P = 1.0$), then $S_N = N$. On the other hand, if N is too big, S_N tends to $1/(1-P)$; this value represents an effective limit to the maximum acceleration achievable, demonstrating that it is useless to indefinitely add more and more units of parallel execution in an attempt to improve performance (this idea is conveyed by the dashed line in Figure 2.3).

Figure 2.4 shows the speedup value according to Amdahl's Law, for different values of P and N. The theoretical limitation imposed to the speedup by the Amdahl's Law becomes clear. For instance, if the parallel portion of the program corresponds to 90% of runtime, it is not possible to obtain more than an acceleration of 10x against the purely sequential version, regardless of adding more than 512 processors.

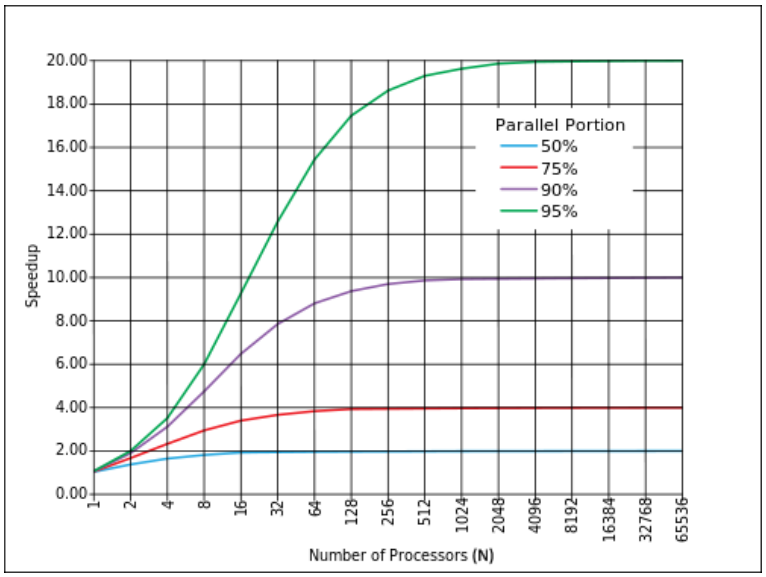

Figure 2.4: Speedup according to Amdahl's Law, for different values of P and N [Bar11a].

Another important law in parallel computing is Gustafson's law, which is closely related to Amdahl's law. This law can be formulated as follows:

$$S(P) = \ P - \alpha(P - 1),$$

where P is the number of processors, S is the acceleration and α the non-parallelizable part of the problem.

In a simplistic way, Gustafson's law states that problems with a large, repetitive data set can be efficiently parallelized. Thus, this law contradicts (to some extent) and complements Amdahl's Law, which describes the existence of a speedup limit that the parallelization can offer. The Amdahl's law also does not consider the variation in the availability of computing power as the number of systems increases. Therefore, the Gustafson's law proposes that programmers define the size of the problems in order to use the equipment available to solve these problems in a practical and fixed time. Thus, if a faster hardware is available, larger problems can be solved in the same amount of time.

Amdahl's law assumes a fixed size problem, implying that the sequential portion of the program does not change with the increase of the number of processors, and the parallel portion is evenly distributed over P processors. The impact of Gustafson's law, allowed problems to be reworked, so that the solution for large problems were possible, in the same amount of time [Sul12].

2.2 Parallel Computing Platforms

Parallel Applications may target a wide range of platforms. In this section we provide a brief description of the main parallel computing platforms used nowadays.

2.2.1 SMP, Multi-Core and Many-Core Systems

A computer system that uses symmetric multiprocessing is called SMP (Symetric Multiprocessor). This type of systems involve computational hardware that accommodates two or more identical processors connected to a single shared memory, and controlled by a single instance of an OS (Operating System) – see Figure 2.5.

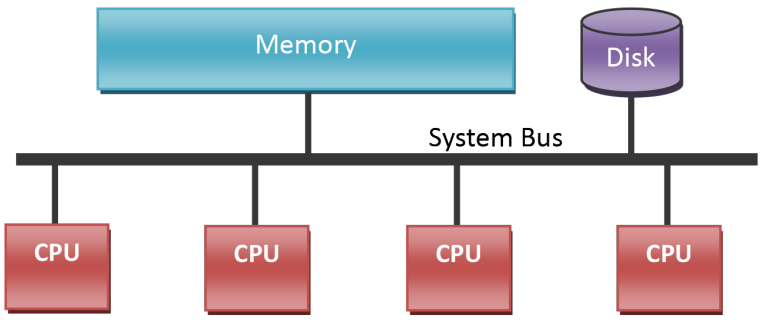

Figure 2.5: SMP system representation.

SMP systems allow any processor to work on any task, no matter the location in memory of this task, provided that each task in the system is not running on two or more processors simultaneously. With the correct support provided by the OS, SMP systems

can easily move tasks between processors in order to balance the load efficiently. However, one of the obstacles in the scalability of such type of systems is the bandwidth restrain and power consumption of the connections between the various processors, memory and disk. Another obstacle is that the maximum number of usable processors is relatively small (typically 32) [Wik12h].

Nowadays, SMP systems evolved to **Multi-core Systems**. In this type of systems, multiple processor cores are integrated in a single chip (see Figure 2.6), instead of a single core per chip.

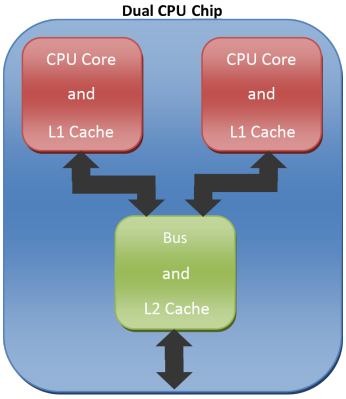

Figure 2.6: Multi-core System representation.

The Multi-core Systems were motivated by the high energy consumption (and consequent heat dissipation) of SMP systems, resulting from the strategy of increasing progressively the core frequencies. Introducing this type of systems has brought advantages, since the proximity of the processor cores lead to more economical manufacture: i) less raw material required; ii) sharing some components between multiple cores (e.g., L2/L3 caches), higher performance of the whole; iii) increased number of cores for the same physical area; iv) shorter paths of electrical signals.

The term Multi-core is related to **Many-core**. Many-core and Massively Multi-core are terms used to describe multi-core architectures with an especially high number of cores, in the order of the tens or hundreds, while Multi-core often refers only for dual-, triple-, quad- or octo-core units. Many-core is closely related to the GPUs architecture, for

example, which is being deployed in a broad spectrum of applications including Clusters, Clouds and Grids, in order to increase performance through parallelism.

2.2.2 Clusters

Clusters are another type of systems used in parallel computing. A cluster is a group of loosely coupled computers that work together, so that, in some aspects, it may be considered as a single computer. Clusters are composed by multiple individual machines connected by a network – see Figure 2.7. These individual machines, or compute nodes, are typically *SMP* or *Multi-core Systems*. While the machines in the cluster do not have to be homogeneous, load balancing is more difficult if they are not. As for the network, which links the cluster nodes, it may use widespread technologies (Ethernet) or very high-bandwidth proprietary technologies (Myrinet/Infiniband).

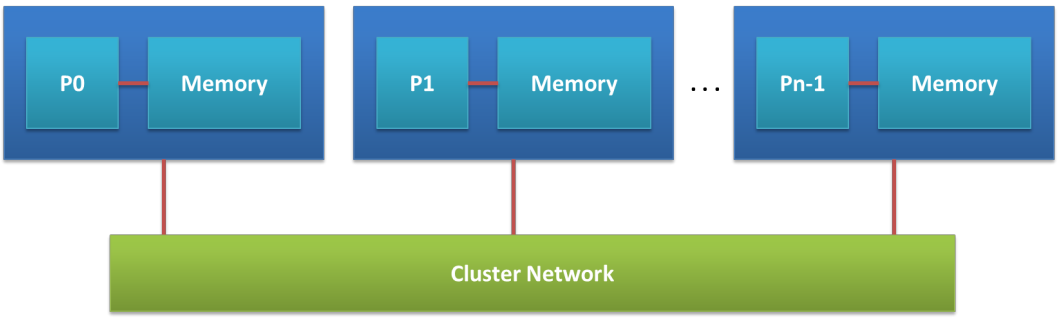

Figure 2.7: A Cluster system representation.

The most common type of cluster is the Beowulf cluster, which is implemented on multiple common commercial computers, connected by a local TCP/IP Ethernet network. A cluster with Multi-core Systems was the parallel computing platform used for the development of this project – see section 2.6.

2.2.3 Grids/Clouds

If a set of clusters is connected via the Internet, another parallel system arises, called **Grid**. Such systems are commonly used for computing as a service, for resolution of highly parallel problems, being the most distributed parallel computing form. Each cluster has its own independent management and specific hardware. What distinguishes grid computing from conventional high performance computing systems such as cluster computing is that grids tend to be more loosely coupled, heterogeneous, and geographically dispersed. Although a single grid can be dedicated to a particular application, commonly a grid is used for a variety of purposes. For certain applications, "distributed" or "grid" computing, can be seen as a special type of parallel computing that relies on complete computers (with onboard CPUs, storage, power supplies, network interfaces, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet. This is in contrast to the traditional notion of a supercomputer, which has many processors connected by a local high-speed computer bus.

Related to Grid computing is the concept of **Cloud Computing**. However it is not a completely new concept, since it is also closely related to cluster computing and distributed systems in general. The concept of Cloud computing resides in the fact that the same Grid architecture is delivered on demand to external customers over the Internet for standard or high demanding tasks, while Grid computing is normally set to solve high demanding problems. These concepts are what distinguish these two models. Usually the Cloud service is delivered and driven by big enterprises, since they have the resources to create large-scale systems containing hundreds of thousands of computers, providing continuous support and on demand service [Mye12, FZRL08].

2.2.4 Heterogeneous Systems

The world is heterogeneous in nature. The diversity given by this provides richness and detail, providing also, at the same time, complexity and interaction where different entities

are optimized specifically for certain tasks and environments [GHK+11]. This happens in our world but, in computing, heterogeneous electronic systems also add richness, because they allow a programmer to choose the best architecture to execute the task at hand or to choose the best task that makes an optimal usage of a particular architecture. These two ways of approaching an heterogeneous environment, allow to be aware of the flexibility of an heterogeneous system when used to solve computational problems that involve a variety of different tasks. Thus, recently the computer design community, driven by the high performance computing (HPC) community, started to experiment with building heterogeneous systems, which combine a number of different classes of architectures.

To sum up, in general, heterogeneous computing consists on using processors with different instruction set architectures (ISAs), to solve computational problems in order to achieve high performance or to solve the problem quicker, by using a variety of different types of computational units, namely: *general purpose processors* (GPPs) (i.e. a CPU), *special purpose processors* (i.e. digital signal processor (DSP) [Wik12d] or a graphics processing unit (GPU)), *a co-processor*, or a *custom acceleration logic* (application-specific integrated circuit (ASIC) or field-programmable gate array (FPGA)) [Wik12e, Wik12f]. Heterogeneous computing can thus be considered a way of breaking the high performance computing barrier imposed by the limitations of Moore's Law [Sha06].

In the past years, parallel computing devices have been increasing in number and in processing capabilities. From the computational units mentioned above, newer to the computing scene are the GPUs, which are providing unprecedented levels of processing capabilities at low cost. The demand for real-time three-dimensional graphics rendering (a highly data-parallel problem) increased and, because of this, GPUs have evolved as rapidly as very powerful, full programmable and capable of supporting task and data parallelism. Now the combination of CPUs and GPUs is usual, creating a new generation of heterogeneous computing platforms. In these platforms, compute- and data-intensive portions of an application can be offloaded to the GPU, providing significant performance gains, while the host CPU executes other, less intensive tasks.

2.3 Parallel Programming Models and Frameworks

Parallel Applications may be developed using various programming models, including: Shared Memory, Threads, Message Passing and Hybrid. These models exist as an abstraction above the hardware and memory architecture. Although not apparent, the models mentioned are **not** specific to one type of machine or memory architectures. In fact, any of these models can be implemented (theoretically) on any hardware [Bar11a].

The next sections provide a brief description of these models and some related frameworks and implementations, with a main focus on those used to support this dissertation.

2.3.1 Shared Memory and Threads

Shared memory refers to an extra large block of RAM that is attached to some address spaces, which can be accessed by several different CPUs in a multiple-processor computer system. In a shared memory system, all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. As a result, all of these processes share the same memory segment and have access to it [Tec12]. This is great when the system has only a few processors that take advantage of the quick communication but, as more processors are added to a bus, the chances that there will be conflicts over access to the bus increase dramatically [Pac11]. This will likely cause the CPUs to cache memory, resulting in some complications [Wik12g]:

- CPU-to-memory connection becomes a bottleneck. Shared memory computers cannot scale very well. Most of them have ten or fewer processors.

- Cache coherence, whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data. Such coherence protocols can, when they work well, provide extremely high-performance

access to shared information between multiple processors. On the other hand they can sometimes become overloaded and become a bottleneck to performance.

In shared memory multiprocessor architectures, such as those mentioned on the previous section, threads can be used to implement parallelism. Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

POSIX Threads

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard. Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a `pthread.h` header/include file and a thread library – though this library may be part of another library, such as *libc*, in some implementations [Bar12]. There are around one hundred Pthreads procedures, all prefixed "`pthread_`" and they can be categorized into four groups: thread management (creating, joining threads, etc.), mutexes, condition variables, and synchronization between threads using read/write locks and barriers.

2.3.2 Message Passing

The Message Passing model uses messages exchanges between processors that have their own memory, in order to share data or synchronization state. Communicating processors can co-exist on the same computer system, using the busses and local memories for

communication, or may be located on different systems, in this case interconnected via a data network, which can use common technologies (e.g., Ethernet) or very high speed ones (e.g., Myrinet, Infiniband, 10G Ethernet, etc.).

From the programming perspective, Message Passing models implementations typically include a library of subroutines that are embedded in the source code. The programmer is responsible for choosing and designing the whole parallelization strategy, including tasks synchronization. The programmer should also decide on how communication takes place: for example, if the most appropriate communication primitives are synchronous or asynchronous, what kinds of guarantees (delivery ones, sorting ones, etc.) are offered in message exchanges, etc..

Sets of tasks that use local memory during computation fit into this model; these tasks may exchange data through various communication patterns, which require cooperative complementary operations in the process involved (e.g., a send operation must have a corresponding receive operation). In Figure 2.8 it is possible to view a simplified representation of this model.

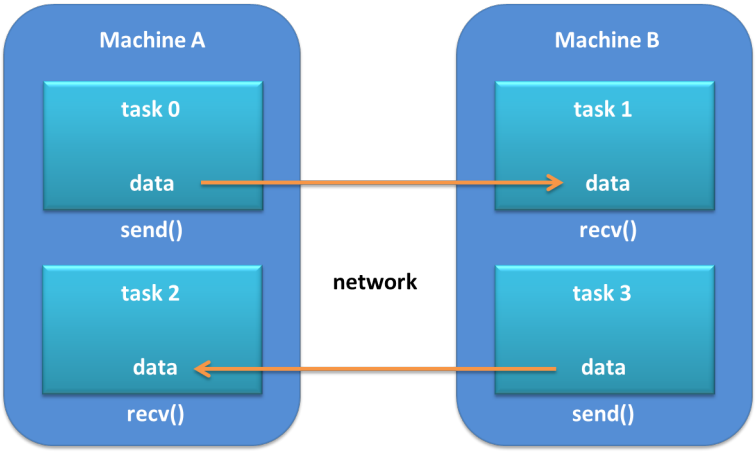

Figure 2.8: A simplified view of the Message Passing Model.

Examples of the Message Passing model are: PVM (Parallel Virtual Machine) [Lab11b], MPI (Message Passing Interface [For94], used on this project – see next – and MPL (Message Passing Library) [IBM].

MPI

In the 80s, there was a wide variety of Message Passing libraries but, these implementations differed substantially from each other, making it difficult for developers to create portable applications. Then, in 1992, the MPI Forum was formed, aiming to establish a standard interface for Message Passing implementations. Thus, in 1994, and subsequently in 1996, two Message Passing Interface[1] specifications were published.

Currently, MPI is the standard used by the industry, having replaced, virtually, all other Message Passing standards. Some of the main reasons for choosing MPI as a model for developing parallel programs are, among others [Bar11b]:

- **Standardization**: as mentioned previously, the MPI specification is the only one that can be considered a standard, and it is supported by virtually all platforms and computer architectures;

- **Portability**: there is no need to modify the code (or the changes will be minimal) when it is needed to transfer an MPI application to a different platform, provided this last one supports an implementation following the MPI standard;

- **Functionality**: over 115 routines are defined in MPI, covering a wide range of aspects;

- **Safety**: reliable communication interface, freeing the programmer from concerns about miscommunication;

- **Availability**: there are several MPI implementations, many open-source; the most used ones are OpenMPI [Tea11] and MPICH-2 [GLA+09], available for both Unix/ Linux and Windows.

[1] Part 1 and Part1 2 of MPI can be found at `http://www.mcs.anl.gov/research/projects/mpi/`.

MPICH2

The MPI implementation used in this project was MPICH-2, one of the most popular implementations in the industry. This is a high performance and widely portable implementation. It provides a MPI implementation that effectively supports different computation and communication platforms, including **clusters**: i) desktop systems; ii) shared memory systems; iii) multi-core architectures, **high-speed networks** (10G Ethernet, InfiniBand, Myrinet, Quadrics) and **proprietary computing systems** (Blue Gene, Cray, etc). MPICH-2 enables cutting-edge research in MPI, through a scalable modular framework that allows the use of other derived implementations [MPI11].

Choosing which MPI implementation to use may depend on how it behaves in executing a specific application. That is, it may be necessary to perform a benchmark on the program using different implementations, and using one or other depending on those benchmark results. In our case, the choice fell on MPICH-2, instead of OpenMPI. Due to memory limitations, OpenMPI is not suitable for large-scale problems [Sta11, fACR11], such as those used in this project. A slight increase in performance and less benchmarks variance were also observed with MPICH2, when compared to OpenMPI in our cluster.

2.3.3 Heterogeneous Systems

In this section we present the main models and frameworks currently used to exploit heterogeneous systems. Novel frameworks that extend those models to parallel/distributed execution environments are also present, with the exception of the clOpenCL framework, whose validation was one of the main purposes of this work, and thus is presented in a section of its own – see section 2.4.

CUDA

CUDA (Compute Unified Device Architecture) is a proprietary parallel computing platform and programming model developed by Nvidia [Nvi12] for graphics processing. CUDA

enables software developers to have access to Nvidia graphics processing units (GPUs) to perform general purpose computing, using variants of standard programming languages: "C, C++ and Fortran" (standard languages with Nvidia extensions and certain restrictions). Moreover, CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements, and requires developers to configure access of global memory, cache, and the amount of available threads. The developers will also be responsible for scaling the activities between the GPU and CPU [Wik12c].

Nvidia also provides what they call the "CUDA Toolkit", which gives a comprehensive development environment for C and C++ developers, including a compiler, math libraries and tools, for debugging and optimizing one's applications. CUDA also supports wrappers[2] for languages, like Python, Perl, Fortran, Java, Ruby, Haskell, Matlab, and so on, being these third-party ones.

Figure 2.9 explains the CUDA processing flow, which demonstrates how CUDA enables the interaction with the GPU and global memory and the parallel throughput architecture of an enabled CUDA GPU.

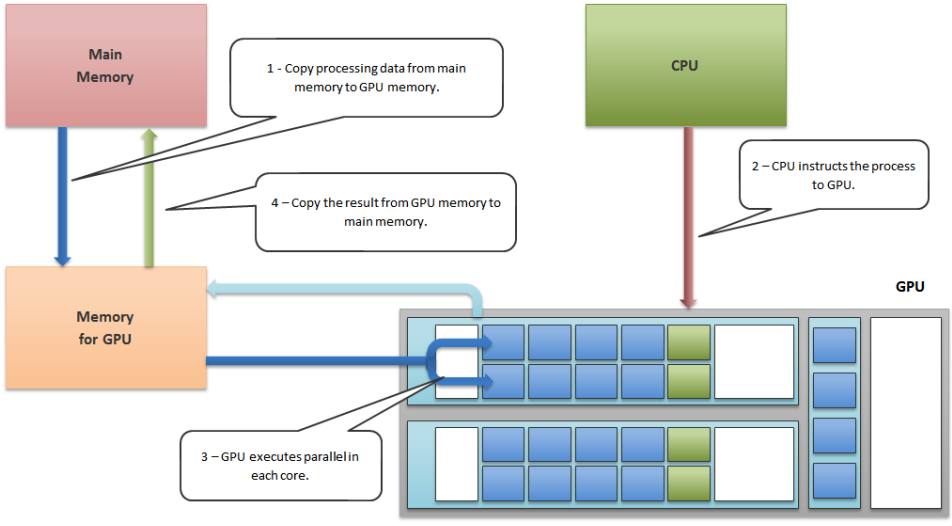

Figure 2.9: CUDA architecture.

[2] Wrapper libraries (or library wrappers) consist of a thin layer of code which translates a library's existing interface into a compatible interface.

Currently CUDA architecture shares the "market" with two competitors, namely: Khronos Group OpenCL [Gro12c] and Microsoft DirectCompute [Mic12]. But, unlike OpenCL, CUDA enabled GPUs are only available from Nvidia.

OpenCL

OpenCL is a parallel programming standard/framework for writing programs that execute across heterogeneous platforms (see section 2.2.4) and it was developed specifically to ease the programming burden when writing applications for this kind of platforms [GHK+11]. These platforms include CPUs, GPUs, Cell Broadband Engines [Wik12b], FPGAs (Field-Programmable Gate Arrays), DSPs (Digital Signal Processors), and other processor devices. The diversity in architectures allows the designer to provide optimized solutions for many kinds of problems. If a solution is designed within the OpenCL specification, it can scale with the growth and breadth of available architectures.

OpenCL is an open standard maintained by the non-profit technology consortium Khronos Group [Gro12b]. It has been adopted by Intel, AMD, Nvidia, and ARM Holdings. Each OpenCL implementation (i.e. an OpenCL library from AMD, Nvidia, etc.) defines *platforms* which enable the host system to interact with OpenCL-enabled *devices*. There are, presently, three major implementations of the OpenCL specification, being these the following: i) AMD APP SDK (for CPUs and AMD GPUs), ii) Nvidia's implementation (for NVIDIA GPUs only) and iii) Intel OpenCL SDK (for CPUs only).

This framework enables a language based on C99 standard [Wik12a] (with some limitations and additions) for writing kernels (programs that are to be executed on the device side, syntactically similar to a standard C function – see Appendix B.1). However, it ommits the use of function pointers, recursion, bit fields, variable-length arrays, and standard C99 header files. On the other hand, the language is extended to ease the use of parallelism with vector types and operations, synchronization and functions to work with work-items/groups. Memory region qualifiers were added (`__global`, `__local`,

`__constant`, `__private` (refer to Appendix B.1.2) and also many built-in functions, along with changes to the supported data types, which received the prefix "`cl_`" (see Table B.1 in Appendix B.2).

OpenCL also provides APIs that are used to define and control the platforms, and enables parallel computing using task-based and data-based parallelism. With OpenCL, developers can give any application access to the GPU for non-graphical computing, extending this way, the power of the graphics processing unit beyond graphics.

When comparing CUDA and OpenCL one should stress the fact that CUDA supports Homogeneous Computing, while OpenCL targets, by design, Heterogeneous Computing. The later is the main reason for choosing OpenCL for this project, despite the maturity of CUDA: while CUDA only targets Nvidia GPUs (homogeneous approach), OpenCL targets any processor device that supports input and output (heterogeneous approach). This way, with OpenCL, it is possible to have multiple CPUs and GPUs, from different manufacturers and models, working together to increase performance.

Despite being a great step towards heterogeneous computing, the original OpenCL specification is unable to meet the needs of HPC applications for clusters and other distributed environments. This is due to the fact that OpenCL applications can only utilize the local devices present on a single machine. Therefore, in the OpenCL model, an application can only run in a single node and the number of OpenCL devices available to an application may be rather limited. All of this applies also to CUDA.

Thus, new or modified models are needed for OpenCL applications to be able to use several cluster nodes. To address this issue, several projects were initiated, and there are now several approaches available. In the next section a brief description of some of those approaches is provided (excluding the clOpenCL approach, presented in section 2.4).

MOSIX VCL

The Virtual OpenCL (VCL) cluster platform [BS11] can transparently run unmodified OpenCL applications in many devices (including GPU and APU devices) in a cluster, as if all the devices are located in the hosting node. It can create the abstraction of a global OpenCL platform combining all compute devices present in a cluster. VCL benefits OpenCL applications that can use many devices concurrently; it's users can start parallel applications on an hosting node, then VCL will manage and transparently run kernels of the application on different nodes.

The VCL cluster platform consists of three components: the *VCL library*, which is a cluster implementation of the OpenCL standard; the *broker*, which performs cluster-wide allocation of resources; and a *back-end* daemon, which runs kernels on behalf of host applications. Therefore, the VCL structure is flexible enough, allowing the incorporation of many algorithms, such as *network optimizations*, *load-balancing* and *dynamic configurations* [BS11]. Note, however, that only VCL binaries are distributed.

Hybrid OpenCL

The Hybrid OpenCL [AONM11] project enables the utilization of OpenCL devices over the network. This platform consists of two elements: a *runtime system* that provides the abstraction of different OpenCL implementations and, a *bridge program* that connects multiple runtime systems over the network. The runtime system of the Hybrid OpenCL is divided in two parts: i) the *host* part, and ii) the *remote* part. The runtime systems of the remote part can be different. Also, the operating environment of the runtime systems of the remote part can be different from the host part.

This system was developed for a particular device independent OpenCL implementation, thus making it difficult to exploit high performance GPUs, for example. Also, although focusing on simplicity, some performance related issues still need to be tackled.

GVirtuS

The GVirtus (Generic Virtualization Service) [GML+11] is a framework for implementation of split-driver based abstraction components and the virtualization of hardware devices. GVirtuS fills the gap between in-house hosted computing clusters, equipped with custom devices, and pay-for-use high performance virtual clusters deployed via public or private computing clouds. It allows an instanced virtual machine to access CUDA powered GPGPUs in a transparent way, with an overhead slightly greater than a real machine/GPGPU setup. GVirtuS is hypervisor independent, and although it currently virtualizes nVIDIA CUDA based GPUs, it is not limited to a specific brand technology [fHPSC12]. The framework currently has full threadsafe support to CUDA drivers, CUDA runtime and OpenCL. Also, it partially supports OpenGL integration. GVirtuS approach is composed in two parts: *GVirtuS Frontend*, a dynamic loadable library with the same application binary interface that runs on the guest user space; *GVirtuS Backend*, a server application running on the host user space and performing concurrent requests [MCG+11].

rCUDA

rCUDA [DPnS+10] enables applications to concurrently use CUDA-compatible GPUs installed in remote computers as if they were local devices. To enable a remote GPU-based acceleration, rCUDA creates virtual CUDA-compatible devices on those machines without a local GPU. These virtual devices represent physical GPUs located in a remote host offering GPGPU services. As a result, all of the nodes are able to concurrently access the whole set of CUDA accelerators installed in the cluster.

This framework follows a client-server model where the client uses a wrapper library and a GPU network service listens for TCP requests. On the down side, the framework is limited in the number of compute devices that CUDA can handle (NVIDIA devices only).

2.3.4 Hybrid Models

Usually, the exploitation of heterogeneous systems is associated with hybrid models, although these models have been used, until recently, mainly in homogeneous systems. A hybrid model is a combination of several parallel programming models in the same program. They may be mixed in the same source; and they may be combinations of components or routines, each being in a single parallel programming model.

The most familiar hybrid model is the junction of MPI with OpenMP[3] [Gro12a], where OpenMP parallelization takes place at each node and MPI parallelization between nodes. Depending on the kind of communication and computation, these models can be used, generally, in two ways. In the first one, communication and computation do not overlap: MPI is called only outside of parallel regions and by the master thread; also, MPI is called by several threads. In the second one, communication and computation do overlap (while some of the threads communicate, the rest are executing an application): MPI is called only by the master thread; communication is carried out by several threads; each thread handles its own communication demands [vA08].

The hybrid approach is difficult: developers always want to use the best tools for each part of their program, and that sometimes means building their applications from pieces in different languages. However, this model usually translates in performance benefits, due to better computation/communication ratios.

In the work presented on this dissertation two different hybrid models were used: a hybrid model using i) MPI to manage a distributed set of worker processes, and ii) OpenCL to exploit computation on local worker-specific devices; another hybrid model (clOpenCL) based on i) a distributed set of OpenCL device-proxies (daemons), POSIX Threads to interact with a cluster-aware OpenCL routine, and iii) Open-MX for communications. The clOpenCL model is thoroughly explained in section 2.4.

[3] An API that can be used for multithreaded shared memory parallelization that enables parallelization of one part of the program at a time.

2.4 Cluster OpenCL

Frameworks like VCL, Hybrid OpenCL, GVirtuS and rCUDA, previously presented in section 2.3.3, try to cope with the limitations of specifications originally tailored to self-contained systems, by providing the necessary support for their instantiation in distributed environments, like clusters, grids and even clouds.

Making possible the running of unmodified OpenCL applications in a transparent way, on devices scattered in a cluster environment, was the main motivation of a novel framework, named clOpenCL (*cluster* OpenCL). clOpenCL was developed (as part of a R&D project to which this dissertation is related) using an hybrid programming model that mixes POSIX Threads, OpenCL and a low-level/high performance message passing library (Open-MX[4]) [Gog11].

2.4.1 General Architecture

clOpenCL comprises a wrapper library and a set of daemons. Every call from an application to an OpenCL primitive is intercepted by the wrapper library, which redirects its execution to a specific daemon at a cluster node or to the local OpenCL runtime. clOpenCL daemons are simple OpenCL programs that listen and handle remote calls and interact with local devices. The host component of a typical clOpenCL application will be multi-threaded (POSIX Threads)[5]. It will start in a particular cluster node and will create OpenCL contexts, command queues, buffers, programs and kernels across all cluster nodes [ARPS12].

Figure 2.10.*a)* shows the clOpenCL operation model, where a single OpenCL host application component interacts with multiple compute devices, whether local or remote. Figure 2.10.*b)* represents the different software and hardware layers that support the host

[4] An open-source message passing stack over generic Ethernet which provides low-level communication mechanisms at user-level space and allows to achieve low latency communication and low CPU overhead.

[5] This is mainly for performance reasons, since there is no fundamental limitation for not using the process model on top of the clOpenCL library.

application component.

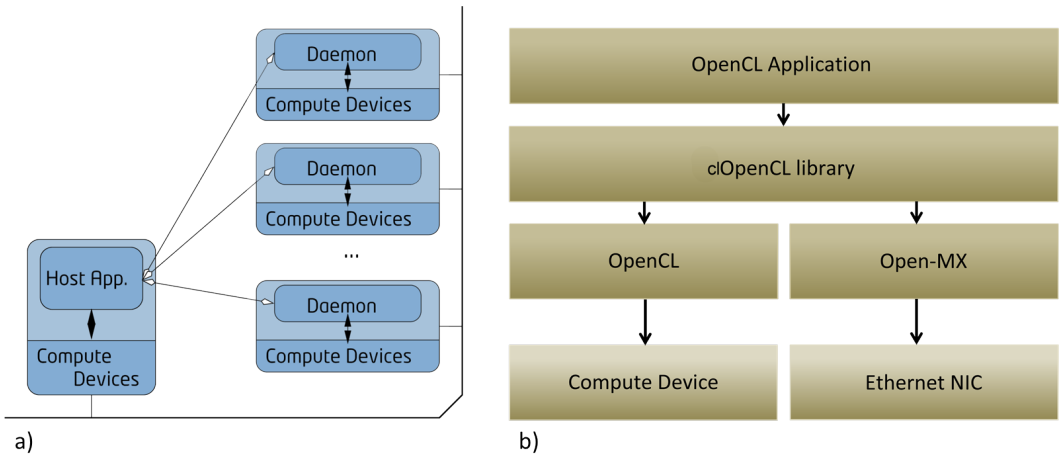

Figure 2.10: clOpenCL (a) operation model, (b) host application layers.

2.4.2 Operation Model

When the host program starts, the clOpenCL wrapper library, wich also wraps the `main` function, locates all active clOpenCL daemons by interacting with the Open-MX mapper service (this service creates a distributed directory, where each process, with an opened Open-MX end-point, is registered).

In a traditional OpenCL application, the programmer has to manipulate only the objects returned from the OpenCL API, namely: platforms and device identifiers, contexts, command queues, buffers, images, programs, kernels, events and samplers. These objects are actually pointers to complex OpenCL data structures, having their internal constitution not accessible to the programmer. In a distributed/parallel environment, where OpenCL primitives are executed in multiple daemons, those pointers cannot be used only to identify objects, because each daemon has its own address space. Therefore, for each object created by OpenCL, the wrapper library returns a "fake pointer" used as a global identifier, and stores the real pointer alongside with the corresponding daemon location.

Each time the wrapper library redirects an OpenCL primitive, its parameters are

packed into an Open-MX frame and sent to the remote daemon that will execute the primitive. Every parameters that reference OpenCL objects are previously mapped into their real pointers and the daemon is determined accordingly. The wrapper library and daemons do not maintain any information about calls to OpenCL primitives, i.e., they are stateless. Any data or state needed for subsequent primitive calls are maintained by the OpenCL runtime at each cluster node. Therefore, it is not necessary to manage complex data structures related to the OpenCL operation and state [ARPS12].

Typically, an OpenCL application starts with a discovery phase with the purpose of finding platforms and their respective devices locally (*i.e.*, available at the node where the application is hosted). In clOpenCL this discovery phase will return a set of all platforms and devices available (both *local* and *remote*) in the cluster nodes where the clOpenCL service (daemon) is running. In more detail, clOpenCL first returns all local platforms, if any (once it is not mandatory that any OpenCL platforms be locally available); afterwards it returns the remote platforms, node by node.

A problem with clOpenCL was to know to which cluster node a certain platform belongs. Thus, the OpenCL primitive `clGetPlatformInfo` was extended with a special attribute named `CL_PLATFORM_HOSTNAME`. Having the possibility of choosing specific cluster nodes where to run OpenCL kernels may be useful, *e.g.*, for load balancing purposes.

2.4.3 Using clOpenCL

Porting OpenCL programs to clOpenCL, only requires linking to the OpenCL and clOpenCL libraries. This is accomplished by taking advantage of the GCC directives, `-Xlinker --wrap`, for function wrapping during link-time. Currently, clOpenCL does not support mapping buffer and image objects. However, as its current state, the clOpenCL platform is enough to the purpose of testing its general concept, including running basic OpenCL applications [ARPS12], like a kernel for matrix multiplication.

2.4.4 Distributed OpenCL

dOpenCL (*distributed* OpenCL) [KSG12] is a recent framework, developed at the same time as clOpenCL, with whom has resemblances: it supports transparent multi-node-multi-accelerator OpenCL applications and combines a wrapper client library with remote services. However, dOpenCL is oriented to general distributed environments, uses a TCP/UDP based communication framework, and devices may not be concurrently shared. In turn, clOpenCL targets HPC clusters, uses Open-MX to maximize the utilization of commodity Gigabit Ethernet links, and devices are fully shareable. Both approaches work on top of any OpenCL platform and so are able to exploit many device types.

2.5 Case Study: Matrix Product

To test, evaluate and demonstrate the usability of a new parallel programming framework, it is essential to choose appropriate case studies. In this work, clOpenCL was evaluated using a reference HPC "benchmark" algorithm: the Matrix Product algorithm.

Although simple, this "embarrassingly parallel"[6] algorithm is sufficient to test the scalability and correctness of the current clOpenCL implementation. Choosing this algorithm also allows for the quick development of alternate implementations (both serial and parallel) to be used for control (correctness verification) and performance comparison purposes. Note, however, that the aim is not an HPC-class performance of the clOpenCL version but, as already stated, to assess its correctness and scalability.

Matrix Product Basic Concepts

This section, quickly describes the concept of the matrix product in formal terms. The product of a matrix A of order $m \times n$ (m rows and n columns) by a matrix B of order

[6] The program that has virtually no communications between the system's parallel activities, is called embarrassingly parallel.

$n \times p$ (n rows and p columns), is a matrix C of order $m \times p$ (m rows and p columns) – see Figure 2.11. An element in C, on the i-th row and j-th column, is obtained by the dot product of the entire i-th row of A by the entire j-th column of B (this is why if A has n columns, B must have n rows). For example, if vector i is a row of A that contains $[a11, a12, ..., a1n]$, and vector j is a column of B that contains $[b11, b12, ..., b1p]$, their dot product can be computed as follows: $i.j = a11b11 + a12b12 + ... + a1nb1p$.

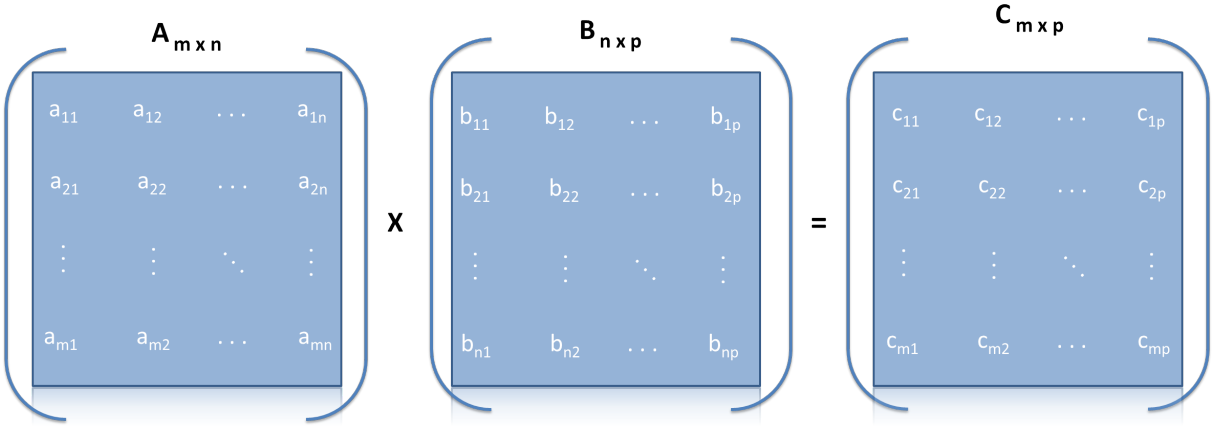

Figure 2.11: Matrix multiplication representation.

If A and B are square matrices of order 4×4, the full matrix multiplication will require 16 products, i.e., each of the four rows of A must be multiplied by each of the four columns of B. If A and B are not square matrices, the product requires vectors of equal length, so the rows of the first matrix must have the same size as the columns of the second.

2.6 Experimental Testbed

To test, evaluate and demonstrate the usability of a developed parallel application, there is always a need for a parallel computing platform where these activities can be performed. The testbed platform used in this work was a small scale commodity cluster, operated

by the Informatics Laboratory of ESTIG/IPB. The Cluster concept, as a parallel execution platform, was already discussed at section 2.2. Therefore, in this section, only the particular aspects that characterize the used cluster will be described.

2.6.1 Computational Systems

The cluster includes the following computational systems:

- a frontend (`beta`), with an Intel Core 2 Duo E6400 2.13GHz CPU and 4Gb of RAM;
- several subsets of compute nodes, defined taking into account their physical location, their hardware features and their degree of availability (continuous/intermittent):
  - *datacenter* subset, with 4 computing nodes, each one with an Intel Core 2 Quad Q9650 3GHz CPU, 8Gb of RAM and two Ethernet 1Gbps NICs (on-board Intel 82566DM-2 and a PCI64 SysKonnect SK-9871); plus, the nodes are fitted with NVIDIA GTX 460 GPUs (1Gb of GDDR5 RAM): 1 node (`compute-4-0`) with 2 GPUs and the 3 remaining nodes (`compute-4-1` to `compute-4-3`) with 1 GPU each; this subset is continuously available for work execution;
  - *labinf* subset, with 8 computing nodes (`compute-4-4` to `compute-4-11`), each one with an Intel Core 2 Quad Q9400 2.6GHz CPU and 8Gb of RAM; this subset is only available to work outside lessons hours;
  - *esa* subset, with 3 computing nodes (`esa-16-0` to `esa-16-2`), located at the ESA/IPB Informatics Resource Center, each one with an AMD Opteron 6128 Magny-Cours 2.0GHz "Octo-Core" CPU, and 16Gb of RAM; this subset is continuously available for work execution.

For this work only the nodes of the *datacenter* subset were used, taking advantage of all of the processing devices available there: 4 CPUs (16 cores in total) and 5 GPUs.

2.6.2 Network(s)

The *frontend* and the compute nodes are interconnected in a main private network (*eth0*), based on 1Gbps Ethernet; however, for logistical reasons, only the *datacenter* node subset is directly connected to the same switch on which the *frontend* also connects; the remaining subsets, external to the *datacenter*, bind to this switch through a succession of switches properly configured to support a specific cluster VLAN. A secondary private network (*eth1*) also exists, connecting the *datacenter* node subset with the *frontend*; the 1Gbps Ethernet switch used for this purpose is configured with jumbo frames (mtu 9000); this secondary network is used as a "poor-man's", high speed network, once the cluster currently lacks Myrinet or Infiniband. This configuration is illustrated in Figure 2.12.

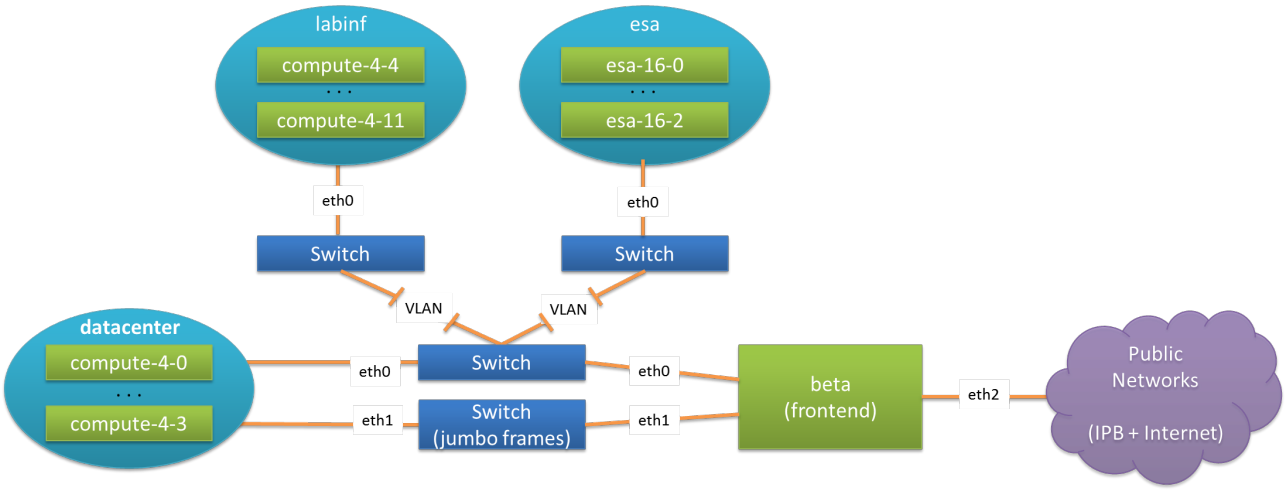

Figure 2.12: Cluster network(s) configuration.

It should be noted that the dispersion of the different node subsets, separated by multiple switches, is not advisable to perform work that use simultaneously different sets and have high communication needs. However that was not the case of this work, since only the *datacentre* node subset was used.

2.6.3 Exploitation System

All cluster systems run Linux Rocks [Gro12d] (version 5.4.3), a specific distribution for cluster environments, developed as part of a project that began in 2000. The ROCKS distribution is built upon the CentOS distribution, a community version of Red Hat Enterprise Server (RHEL), which is a commercial distribution. The choice of CentOS allows a solid and stable foundation, binary compatible with RHEL, which is a reference for the business market. The ROCKS distribution adds to CentOS a number of facilities: support to bulk (re)installation (automatic, over the network, via PXE) of compute nodes [Sac09]; a Ganglia system for node monitoring [Pro12b]; the SGE (Sun Grid Engine) system [Sun12] for queue management of deferred execution of work; the Condor system for the automatic load balancing of nodes through processes migration [Pro12a]; various MPI platforms (such as OpenMP and MPICH2), etc.

On ROCKS, the *frontend* node hosts all cluster user accounts and, in general, all the necessary software; these accounts are shared over the network via NFS, to the compute nodes; this sharing extends to a folder `/share/apps`, with applications installed on the *frontend* and accessible to all nodes; the nodes have a local partition (`/state/partition1`) immune to possible reinstallation, thus adequate to store, in a lasting way, large amounts of data.

OpenCL and Open-MX support software was also installed on the *frontend* and the *datacenter* node subset. The specific OpenCL platform and GPU driver versions used were AMD SDK 2.6 with driver 11.12, and CUDA 4.1.28 with driver 285.05.33. Open-MX 1.5.2 was used with the SysKonnect NICs (that provide better performance than the on-board Intel NICs), the ones used to build the secondary private network (*eth1*).

Chapter 3

Preliminary Experiments

As mentioned previously (see section 2.5), the Matrix Product algorithm was chosen to be the case study of this dissertation. Thus, in order to have an idea of the magnitude of the performance gains eventually achieved by the parallel approaches that would need to be developed (at least a clOpenCL based approach), a preliminary assessment was made of readily available and well-known sequential and parallel approaches, including naive approaches, approaches based on public domain linear algebra libraries and also commercial reference approaches.

3.1 General Experimental Conditions

All the experimental studies (including those of Chapter 4) were narrowed to square matrices of order $n \in \{8K, 16K, 24K\}$, filled with single-precision elements (4 byte floats). These 3 different orders were chosen to support a minimal scalability study. This configuration resulted into matrices of size no less than 256 Mbytes, 1 Gbyte and 2.25 Gbytes, respectively. As there are 3 matrices involved (the operands A and B, and the result matrix $C = AB$) the minimum theoretically RAM occupancy is 768 Mbytes, 3 Gbytes and 6.75 Gbytes, respectively.

For all experiments of this chapter, the complete matrices A and B were used to yield the complete matrix C, i.e., the whole A matrix was multiplied by the whole B matrix through a single execution of the matrix product function used. Furthermore, all experiments were performed in the cluster node `compute-4-0`, using a single core, except for Matlab. The compiler optimization levels used in each case are specifically stated. All execution times presented are averages of at least 3 runs.

3.2 Sequential Naive Approach

The first evaluated approach follows a naive sequential algorithm that served as a starting point to the development of the remaining variants.

Under this algorithm, the matrices A, B and C are first allocated (using the order n sizes previously mentioned); and initialized with $float$ values. Then, matrix B is transposed into B^T (see Figure 3.1 for an example); this is done to accelerate the product of A rows by B columns (since arrays are stored in row-major order in programs developed in C language, the transposition of matrix B will make the dot product of A rows by B columns much more cache-friendly, by minimizing cache-misses). Finally, matrices A and B^T are multiplied.

$$\begin{pmatrix} 1 & 2 & 8 \\ 3 & 4 & 3 \\ 5 & 6 & 1 \end{pmatrix}^{T} = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 8 & 3 & 1 \end{pmatrix}$$

Figure 3.1: Example of a matrix transposition.

Code Excerpt 3.1 shows the implementation and invocation of the naive multiplication function (the full algorithm implementation code is in Appendix A.1).

```
#define SIZE 24576 //16384//8192

void mulMatrix(float *a, float *b, float *c)
// assumes "b" has been transposed
{
        int i, j, k, cc;
        float v;
        int numCells;

        numCells = SIZE*SIZE;
        cc=0;
        for(i=0; i<numCells; i+=SIZE){
                for(j=0; j<numCells; j+=SIZE){
                        v = 0.0;
                        for(k=0; k<SIZE; k++)
                                v += a[i+k] * b[j+k];
                        c[cc] = v;
                        cc++;
                }
        }
}
int main()
{
        //...
        mulMatrix(a, bTrans, c);
        //...
}
```

Code Excerpt 3.1: Sequential Naive Approach Matrix Product.

Table 3.1 presents the evaluation results, with *GCC* optimization level "*O3*".

Table 3.1: Sequential Naive Approach Execution Time (seconds).

Order n (`SIZE`)	$8K$	$16K$	$24K$
Execution Time	610,50s	4901,19s	16569,40s

3.3 BLAS Approaches

BLAS (Basic Linear Algebra Subprograms) [For12] are routines that provide standard building blocks for performing basic vector and matrix operations. The Level 1 BLAS perform scalar, vector and vector-vector operations; the Level 2 BLAS perform matrix-vector operations; the Level 3 BLAS perform matrix-matrix operations. Because BLAS routines are efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software.

The Level 3 BLAS provides the following function for the multiplication of two matrices (single-precision): `sgemm(TransA, TransB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)`. The function `sgemm` (Single Precision General Matrix Multiply) is not limited to perform a single product between two matrices, since the input parameters aren't only the matrices A, B and C. Instead, this routine calculates $C = \alpha AB + \beta C$, where α and β are scalar coefficients. Thus, to perform the simple product $C = AB$, it is enough to have $\alpha = 1.0$ and $\beta = 0.0$. The `sgemm` function has also the option of using the transposed forms of A, B, or both. The function full list of parameters and their meaning follows:

- `TransA`: specifies whether to transpose matrix A.
- `TransB`: specifies whether to transpose matrix B.
- `M`: number of rows in matrices A and C.
- `N`: number of columns in matrices B and C.
- `K`: number of columns in matrix A; number of rows in matrix B.
- `ALPHA`: scaling factor for the product of matrices A and B.
- `A`: matrix A.
- `LDA`: the size of the first dimention of matrix A; if passing a matrix $A[m][n]$, the value should be m.
- `B`: matrix B.
- `LDB`: the size of the first dimention of matrix B; if passing a matrix $B[m][n]$, the value should be m.
- `BETA`: scaling factor for matrix C.
- `C`: matrix C.
- `LDC`: the size of the first dimention of matrix C; if passing a matrix $C[m][n]$, the value should be m.

All the non-naive approaches evaluated in the next sections (including Matlab) are based on Level 3 BLAS implementations. When using the `sgemm` function, special care must be taken to know which order (row/column major) is assumed by the implementation (the original specification is Fortran-oriented and so the original order is column-major).

3.4 ATLAS BLAS

The Automatically Tuned Linear Algebra Software (ATLAS) [Sou12] provides a highly optimized implementation of the BLAS interface, heavily used in high-performance computing, and with C and Fortran77 interfaces. It is characterized for having an optimization approach called Automated Empirical Optimization of Software (AEOS), which is able to produce code specifically tuned for the target execution system. In our case, the ATLAS version used was optimized for the *datacenter* node subset CPUs (Core 2 Quad Q9650).

Code Excerpt 3.2 is from a program that follows the same base algorithm of the naive approach (the full code is in Appendix A.1), except that the matrix B is not transposed and the matrix product function is, in this case, the BLAS-like function `cblas_sgemm (Order,TransA,TransB, M, N, K, alpha, A, lda, B, ldb, beta, C, ldc)`. This function adds a parameter `Order` to the original specification, specifying the use of row- or column-major order for all matrices involved. In our case, the A and B matrices are now supplied in column-major order and, for that reason, no transposition is requested.

```
1  #include "cblas.h"
2  #define SIZE 24576 //16384//8192
3
4  int main()
5  {
6          //...
7          cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, SIZE, SIZE, SIZE, 1.0,
8                      A, SIZE, B, SIZE, 0.0, C, SIZE);
9          //...
10 }
```

Code Excerpt 3.2: ATLAS BLAS Matrix Product.

Table 3.2 shows the evaluation results, this time with the default *GCC* optimization level ("*O*0"), since the ATLAS library used is pre-compiled and already optimized.

Table 3.2: ATLAS BLAS Execution Time (seconds).

Order n (`SIZE`)	$8K$	$16K$	$24K$
Execution Time	59,15s	471,08s	1588,61s

3.5 ACML BLAS

The AMD Core Math Library (ACML) [Cen12] provides a free set of thoroughly optimized and threaded math routines for HPC, scientific, engineering and related compute-intensive applications, supporting both Linux and Windows. This library, released by AMD, provides useful mathematical routines optimized for AMD processors. Furthermore, it offers an implementation of the BLAS specification.

In ACML, the BLAS Level 3 function for the matrix multiplication is `sgemm(TransA, TransB, M, N, K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)`. This function follows the specification faithfully (it has no additional parameters, differently to ATLAS), thus assuming column-major order for the matrices.

Code Excerpt 3.3 shows the invocation of this function in the developed ACML test (the full code is available in Appendix A.1). As with ATLAS, matrices A and B are given in column-major order and so no transpositions are necessary (which is conveyed by the value `'N'` of the first two parameters).

ACML requires the usage of the *AMD OPEN64* compiler [AMD12], whose version 4.2.5.2 was used to compile this test. Also, in this case, the "$Ofast$" optimization level was used. Table 3.3 shows the evaluation results under these conditions.

```
#include "acml.h"
#define SIZE 24576 //16384//8192

int main()
{
        //...
        sgemm('N', 'N', SIZE, SIZE, SIZE, 1.0, A, SIZE, B, SIZE, 0.0, C, SIZE);
        //...
}
```

Code Excerpt 3.3: ACML BLAS Matrix Product.

Table 3.3: ACML BLAS Evaluation Execution Time (seconds).

Order n (`SIZE`)	$8K$	$16K$	$24K$
Execution Time	48,60s	388,00s	1308,81s

3.6 GSL BLAS

The GNU Scientific Library (GSL) [GNU12] is a numerical library for C and C++ programmers. It is free software under the GNU General Public License. This library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite, covering a wide range of subject areas. It has also an implementation of BLAS.

In the previous approaches (naive and BLAS based), the 3 matrices were allocated and initialized using standard C data types (float), functions (malloc) and operators ("="). In GSL one needs to declare matrices of type *gsl-matrix-float*, the function `gsl_matrix_float_alloc(m, n)` allocates a matrix of floats of order $m \times n$, and the function `gsl_matrix_float_set(M, i, j, value)` translates into $M[i][j] = value$. The BLAS-based matrix multiplication is provided by the function `gsl_blas_sgemm (TransA, TransB, alpha, A, B, beta, C)`, a simplified version of the BLAS specification (it assumes column-major order and `LDA`, `LDB` and `LDC` parameters are absent).

Code Excerpt 3.4 shows how these functions were used together to implement the GSL matrix multiplication (the entire code can be viewed in Appendix A.1).

```
#include <gsl/gsl_blas.h>
#define SIZE 24576 //16384//8192

int main(){
        gsl_matrix_float *A, *B, *C;
        A = gsl_matrix_float_alloc(SIZE,SIZE);
        B = gsl_matrix_float_alloc(SIZE,SIZE);
        C = gsl_matrix_float_alloc(SIZE,SIZE);

        for(i=0; i<SIZE; i++){
                for(j=0; j<SIZE; j++){
                        gsl_matrix_float_set (A, i, j, i+1);
                        gsl_matrix_float_set (B, i, j, j+2);
                }
        }
        gsl_blas_sgemm (CblasNoTrans, CblasNoTrans, 1.0, A, B, 0.0, C);
        //...
}
```

Code Excerpt 3.4: GSL BLAS Matrix Product.

Table 3.4 shows the results obtained with GSL BLAS. A quick comparison with the previous BLAS-based approaches reveals that the GSL execution times are one order of magnitude above the ones achieved with those approaches. This observation is further discussed in section 3.8.

Table 3.4: GSL BLAS Evaluation Execution Time (seconds).

Order n (SIZE)	$8K$	$16K$	$24K$
Execution Time	731,13s	5753,23s	19460,31s

3.7 Matlab

Matlab [Mat12b] is not considered a library, but a high-level language and interactive environment for numerical computation, visualization, simulation and programming. It allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, Java, and Fortran.

Unlike the previous approaches, the Matlab environment allows algorithms to be executed in parallel using multicore processors, GPUs, and computer clusters. This is accomplished through the Parallel Computing Toolbox [Mat12c], and the Matlab Distributed Computing Server [Mat12a]. More specifically, the later allows to run the application on a computer cluster almost without changing the program code; with the Matlab Distributed Computing Server running on the cluster nodes it is possible to designate the number of workers[7] who will execute the intended application, while Matlab handles the parallelization and communication between nodes, without requiring the programmer interference.

Even though Matlab uses BLAS routines at its core, the programming procedures were a little different from the previous C-based approaches. The matrices A and B were initially created and filled in Matlab but, to accelerate successive runs, A and B are saved in *.txt* files, from which they are filled when necessary. The Matlab code is simple and straightforward, as shown by Code Excerpt 3.5 (the complete code is in Appendix A.1).

```
1  %%%%%%%%%%%%%%%%%%%%%%%%
2  size = 24576; //16384//8192
3
4  %declare and initialize the matrices
5  matA = zeros(size);
6  matB = zeros(size);
7  matC = zeros(size);
8  %%%%%%%%%%%%%%%%%%%%%%%%
9
10 %populate the matrices by reading the file
11 matA = dlmread('matA16K.txt');
12 matB = dlmread('matB16K.txt');
13 %...
14 %perform distributed matrix product
15 matlabpool open;
16 pmatA = distributed(matA);
17 pmatB = distributed(matB);
18 matC = pmatA * pmatB;
19 matlabpool close;
20 %...)
```

Code Excerpt 3.5: Matlab Approach Matrix Product.

In the code excerpt above, some instructions deserve special attention: `matlabpool open` enables the full functionality of the parallel features, starting a distributed worker pool (the pool size is the number of processes one wants to use); `distributed(m)`, ensures that the matrix `m` is distributed across the worker pool; the calculation of `matC` is

[7] Computational engines that run independently of client sessions, one per each core.

expressed in the usual Matlab syntax (`matC = pmatA * pmatB;`) but, beneath, is done in parallel; `matlabpool close` stops the worker pool, reseting Matlab to the usual (sequential) operation mode.

The previous Matlab code may thus be executed with different processor configurations: a) in one node, using one or more cores, and b) in a set of cluster nodes, also with a variable number of cores per node. Table 3.5 and Figure 3.2 present the evaluation results obtained in several of these configurations. Results with more than one node were produced by adding to the base node `compute-4-0` additional nodes, from `compute-4-1` to `compute-4-3` and using all the 4 cores of each node.

Table 3.5: Matlab Matrix Product Results (seconds).

Num. of Cluster Nodes (num. of cores)	Order n (`SIZE`) $8K$	$16K$	$24K$
1 (1)	60,06s	ND	ND
1 (2)	37,06s	ND	ND
1 (3)	30,06s	136,19s	ND
1 (4)	27,38s	104,85s	ND
2 (8)	20,27s	63,35s	190,23s
3 (12)	19,30s	48,88s	140,98s
4 (16)	19,16s	40,39s	112,72s

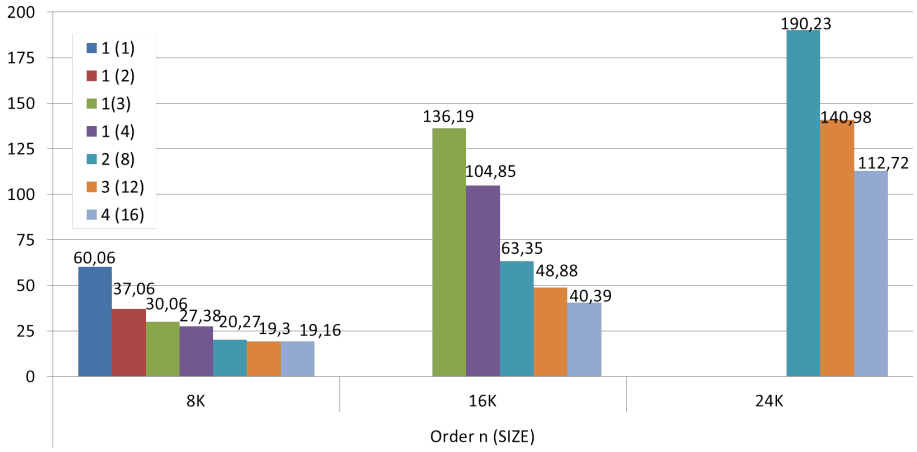

Figure 3.2: Matlab evaluation results graphic.

Differently than what happened with the previous approaches with Matlab, wasn't possible to get all the execution times for orders $16K$ and $24K$. Thus, only when using 3 or more cores was it possible to compute the product of order $16K$, meaning that, in Matlab, matrices of this order need a lot of processing power. This demand is even higher for order $24K$ which requires at least two nodes (8 cores) to execute.

Figure 3.3 shows the speedups achieved by increasing the number of nodes and/or cores. As may be observed, with order $8K$ the gains are modest with more than one node and negligible with 3 or more nodes; with order $16K$ and $24K$ the speedups are smaller but there's still room for improvement with more than 5 nodes (scenarios not tested).

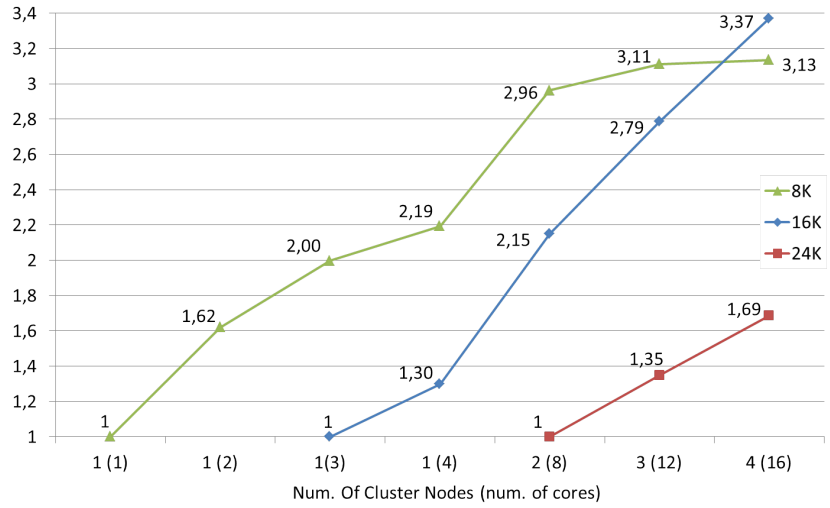

Figure 3.3: Speedups for all combinations.

Only the Matlab result for $8K$ in a single core $(60,06s)$ is comparable with the previous evaluated sequential approaches, having roughly the same performance as ATLAS $(59,15s)$. The remaining Matlab results provide a comparison basis for the results of the parallel versions discussed in the next chapter.

3.8 Preliminary Results Analysis

This chapter ends with a brief comparison of the most relevant results from the preliminary experiments conducted. These results are gathered in Figure 3.4.

The figure only presents results for sequential runs, and so Matlab appears only once (for only one processor core, and when the order of the matrices is $8K$). Also note that the Naive and GSL implementations are not referenced in the graphic, due to the fact that both are particularly time consuming, in comparison to the remaining ones. Therefore those two implementations were excluded from the parallel approaches of Chapter 4.

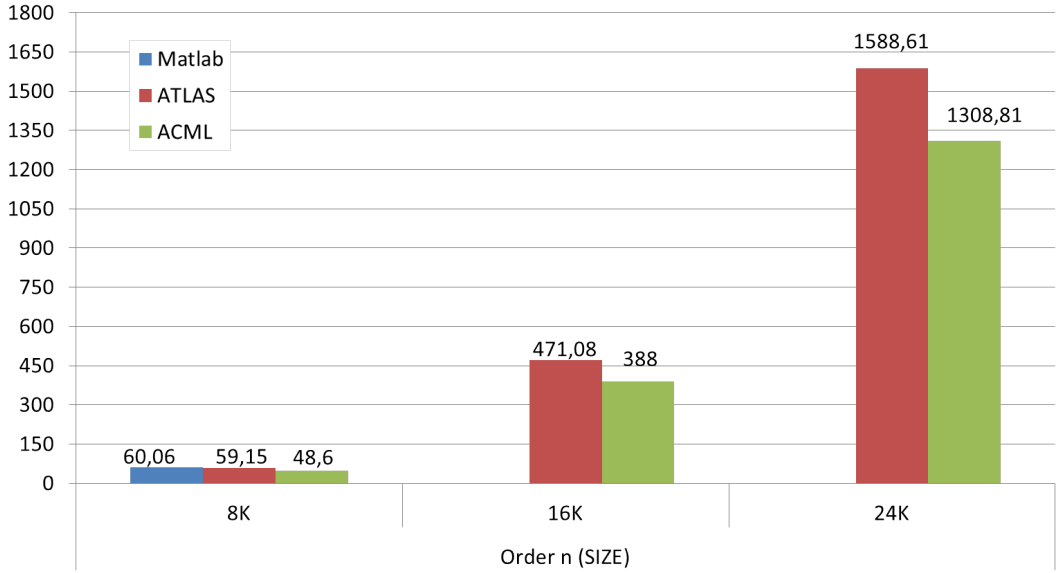

Figure 3.4: Preliminary tests results comparison.

For $8K$ size matrices, Matlab is comparable with ATLAS (taking $60,06$ seconds against $59,15$). However, both lag behind ACML, which is approximately 21% faster. This same observation applies to the $16K$ and $24K$ scenarios, where ATLAS and ACML keep the same relative distance.

So, it is clear that, from all methods, ACML produced the best results, being the most optimized library tested for matrix multiplication. However, ACML-based code only compiles using the "*opencc*" compiler, which is a limitation that prevents ACML to

be used in support of the MPI-based approach presented in Chapter 4. Since ATLAS is the second fastest implementation, while it enables compilation using both "*opencc*" and "*gcc*" compilers, it was the obvious choice to that role.

Chapter 4

Parallel Approaches

This chapter presents three parallel versions of the Matrix Product: a pure MPI approach, an hybrid approach that combines MPI and OpenCL, and another hybrid approach based on clOpenCL with POSIX Threads and Open-MX. For each version, its design and implementation is discussed and evaluation results are presented. A final comparison is also provided.

4.1 Parallelization Strategy

In order to distribute the work involved in the Matrix Product by different processors, a parallelization strategy is needed. The same basic strategy is followed by all the approaches explored in this chapter: Data Partitioning. In this kind of parallelization strategy the problem data domain is divided in sub-domains, which are distributed across the process/processor[8] set, involved in the problem resolution. Then, each processor applies the same algorithm to each sub-domain that was assigned to it. To maximize performance, each processor should work independently on its sub-domains and synchronization points or data exchanges among processors should be avoided/minimized.

[8] In this chapter, the terms process and processor are interchangeable because there will be as much worker processes as processors/devices (one process per processor/device).

In the case of the Matrix Product, the Data Partitioning strategy is easily applicable by performing a "sliced matrix product" (also called a "block matrix product"). This means that, for the product AB, the matrix A is divided in horizontal sub-matrices $subA$ of the same height, and the matrix B is divided in vertical sub-matrices $subB$ of the same width; each $subA$ must then be multiplied by each $subB$; the outcome is a set of sub-matrices $subC$ that, together, make up a result matrix C. Figure 4.1 represents this approach for square matrices of order $n \times n$ (or, more simply, order n) and sub-matrices (blocks) of height or width given by a parameter *slice*.

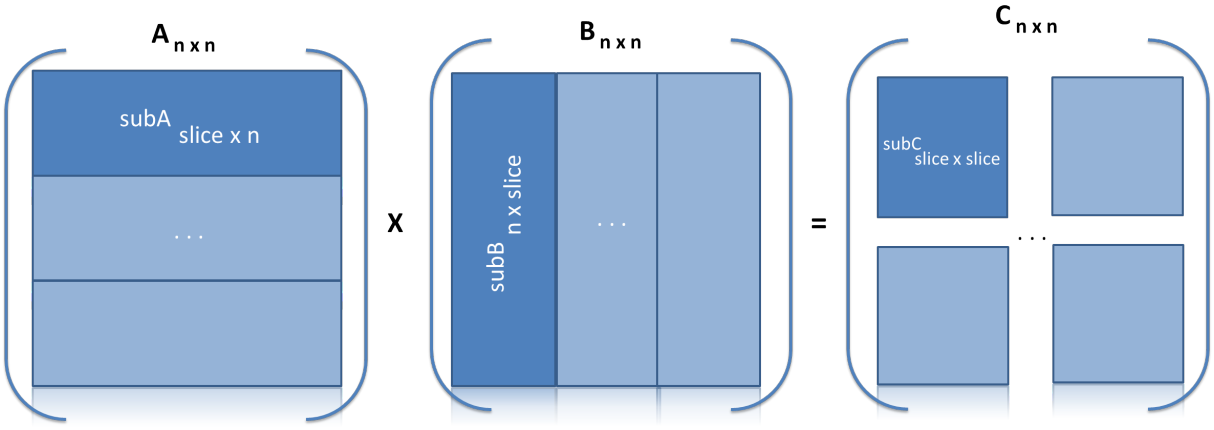

Figure 4.1: Sliced matrix product representation.

Let's clarify this approach with an example. For instance, with square matrices A and B of order $n = 8K$ and $slice = 1K$, A and B will be divided, each one, into $n/slice = 8$ sub-matrices $subA$ and $subB$, of order $slice \times n$ and $n \times slice$, respectively. The product of those sub-matrices results in $(n/slice)^2 = 64$ sub-matrices $subC$, of order $slice \times slice$, which compose the final result: the matrix C, of order n.

In the parallel approaches discussed in this chapter, three values of *slice* were used: $1K$, $2K$ and $4K$. By taking into account that the order n of the matrices tested is $8K$, $16K$ and $24K$, the overall number of sub-matrices C (or tasks) is, for each combination of *slice* and n, given by Table 4.1. The *slice* values were chosen to allow the generation of different task amounts (i.e., a different number of sub-matrices products), allowing to

balance the tasks by the processors as desired (i.e., more finely or more coarsely).

Table 4.1: Number of C sub-matrices.

Order n	$8K$			$16K$			$24K$		
Slice	$1K$	$2K$	$4K$	$1K$	$2K$	$4K$	$1K$	$2K$	$4K$
Tasks	64	16	4	256	64	16	576	144	36

For each parallel approaches, only the evaluation results pertaining to the best performance combinations of n and *slice* will be presented.

4.2 MPI-Only

In the pure MPI implementation – hereafter named MPI-Only – the sliced Matrix Product is easy implemented resorting to a Master-Slave architecture – see Figure 4.2: there will be one or more processes, of the Slave type, one per processor; one Master process will split the data domain, dynamically distribute its portions by the Slaves and gather the partial results in order to produce the final matrix. In each Slave, the product of sub-matrices will use the ATLAS `cblas_sgemm` function, a choice in accordance with the results of the preliminary tests of Chapter 3 – see section 3.8.

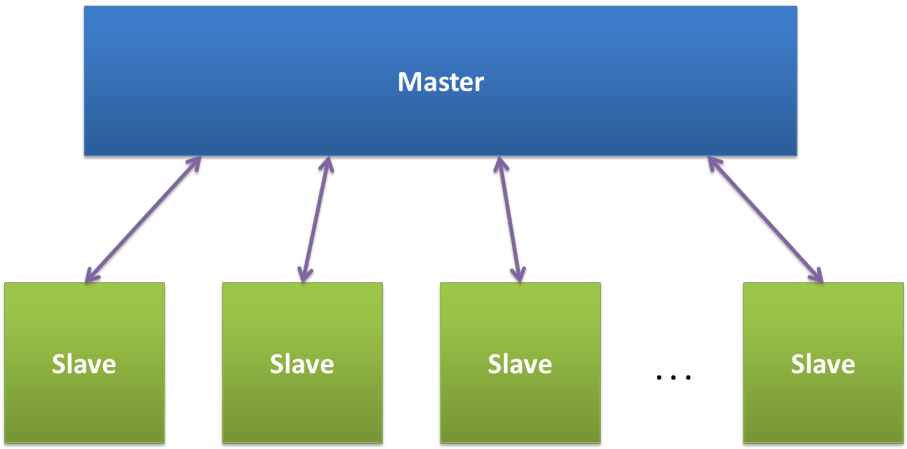

Figure 4.2: Master – Slave Architecture.

The Master process will begin by allocating and initializing the matrices A, B and C. After this initial phase, the Master's work can be divided in two phases: phase 1) dynamic work distribution, while collecting the data results of the multiplication coming from the Slaves; phase 2) Slaves termination, while collecting possible remaining results. On a side note: initially the Slaves were responsible to hold the results until no more work was available but, it was concluded that the application would not lose performance by collecting already available results during the work distribution.

Code Excerpt 4.1 shows the implementation of phase 1 (the complete code is in Appendix A.2). There are two nested "`for`" cycles that ensure that all combinations of sub-matrices of A and B (i.e., all tasks) are distributed to the Slaves. For example, if the matrices A and B are of order $8K$, and the *slice* value is $4K$, matrices A and B will be divided in 2 sub-matrices each, and there will be $2^2 = 4$ sub-matrix products – see Figure 4.3. Therefore, the Master will distribute, in order, the sub-matrices pairs $< subA_1, subB_1 >$, $< subA_1, subB_2 >$, $< subA_2, subB_1 >$ and $< subA_2, subB_2 >$, which will result in the sub-matrices $subC_1$, $subC_2$, $subC_3$ and $subC_4$.

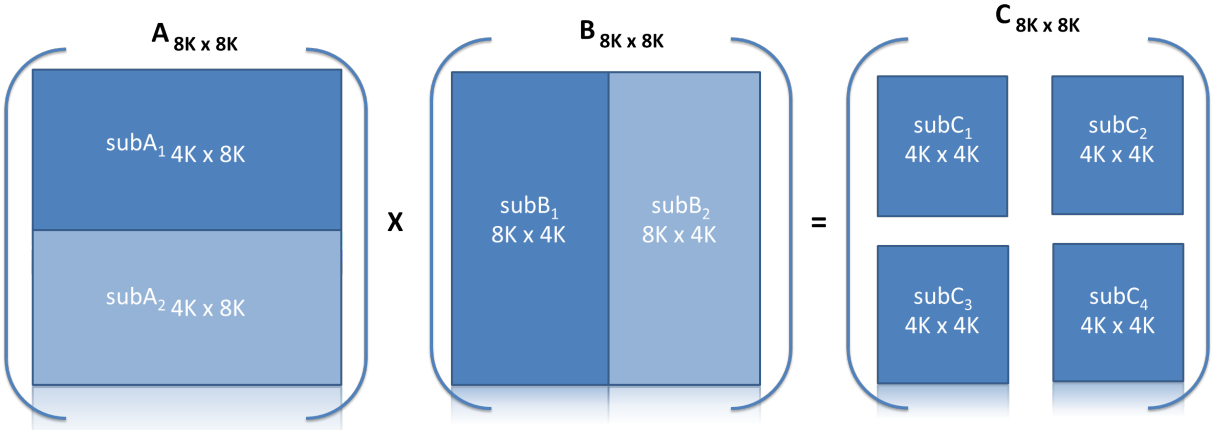

Figure 4.3: A specific sliced matrix product.

In each iteration of the inner "for" loop, the Master waits for a work request from the Slaves, with possible "piggybacked" results (**lines 11 to 17**), meaning that a Slave can be asking for work to perform having already executed the previous work received and so,

at the same time, is giving back the previous work results to the Master; the Slave can also be asking for work for the first time, which means that it has no results to return.

The Master then tests (**lines 18 to 25**) if the Slave has returned results (a *subC*), in which case they are copied to the final matrix C. The offsets `blockA` and `blockB` are fundamental to ensure that the results are set in the correct place on matrix C.

Right after, the Master sends work (a task) to the Slave, composed of work offsets, and A and B sub-matrices (**lines 26 to 36**). Note that the sub-matrices of B are originated from the matrix B transposed (B^T); as explained before (see section 3.2), the transposition of matrix B allows to compute much more cache-friendly dot-products.

```
1  #define SIZE 8192 //16384 //24576
2  #define SLICE 1024 //2048 //4096
3  //...
4  if (taskId == 0) { // MASTER
5     //...
6   int numSlicesPerMatrix = SIZE / SLICE;
7   for(int sliceA = 0; sliceA < numSlicesPerMatrix; sliceA++)
8   {
9     for(int sliceB = 0; sliceB < numSlicesPerMatrix; sliceB++)
10    {
11      //
12      // receive work request from slave, with eventual piggybacked results
13      //
14      retCode = MPI_Recv(&dataSlaveMaster, 1, MPI_DATA_SLAVEMASTER, MPI_ANY_SOURCE, TAG_SLAVEMASTER, \
15                         MPI_COMM_WORLD, &mpiStatus);
16      checkMpiError(retCode, "MPI_Recv", __LINE__);
17      mpiSlaveRank = mpiStatus.MPI_SOURCE;
18      //
19      // we have indeed received piggybacked results; copy them to the final destination in c
20      //
21      if (dataSlaveMaster.blockA != -1 && dataSlaveMaster.blockB != -1) {
22          for(int ii=0; ii<SLICE; ii++)
23              memcpy(  &(c[(dataSlaveMaster.blockA*SLICE+ii)*SIZE+dataSlaveMaster.blockB*SLICE]), \
24                       &(dataSlaveMaster.c2[ii*SLICE]), SLICE*sizeof(float) );
25      }
26      //
27      // send work data to slave
28      //
29      dataMasterSlave.blockA = sliceA; dataMasterSlave.blockB = sliceB;
30
31      memcpy(&(dataMasterSlave.a), &(a[sliceA*SLICE*SIZE]), SLICE*SIZE*sizeof(float));
32      memcpy(&(dataMasterSlave.b), &(bTrans[sliceB*SLICE*SIZE]), SLICE*SIZE*sizeof(float));
33
34      retCode = MPI_Send(&dataMasterSlave, 1, MPI_DATA_MASTERSLAVE, mpiSlaveRank, TAG_MASTERSLAVE, \
35                         MPI_COMM_WORLD);
36      checkMpiError(retCode, "MPI_Send", __LINE__);
37    }
38  }
39  // PHASE 2 //
40 } // END MASTER
```

Code Excerpt 4.1: MPI-Only – Master Phase 1.

The data exchanged between the Master and Slave processes fits into two data types: `dataMasterSlave_t` (for the direction Master→ Slave) and `dataSlaveMaster_t` (for the opposite direction). The definition and instantiation of these structures is in Code Excerpt 4.2: the first has a matrix A offset (`blockA`), a matrix B offset (`blockB`) and sub-matrices of A and B (`a` and `b`); the last includes the offsets and a sub-matrix of C (`c2`). Both structures were used to derive the MPI datatypes `MPI_DATA_MASTERSLAVE` and `MPI_DATA_SLAVEMASTER`, in order to facilitate their exchange.

```
1  //...
2  typedef struct {
3     int blockA;
4     int blockB;
5     float a[SLICE*SIZE];
6     float b[SIZE*SLICE];
7  } dataMasterSlave_t;
8  //...
9  typedef struct {
10    int blockA;
11    int blockB;
12    float c2[SLICE*SLICE];
13 } dataSlaveMaster_t;
```

Code Excerpt 4.2: MPI-Only – Datatype definition.

The phase 2) of the Master (Code Excerpt 4.3) begins when the Master has no more work to distribute by the Slaves; thus, for each Slave, the Master needs to ensure that all possible results still returned by the Slave are properly set in matrix C (**lines 12 to 17**); then, the Master will inform the Slaves that there is no more work, by replying with both work offsets set to "`-1`" and no sub-matrices attached (**lines 18 to 25**).

```
1  if (taskId == 0) { // MASTER
2     // PHASE 1 //
3     //...
4     for (int s=1; s <= numSlaves; s++)
5     {
6       // receive work request from slave, with eventual piggybacked results
7       retCode = MPI_Recv(&dataSlaveMaster, 1, MPI_DATA_SLAVEMASTER, MPI_ANY_SOURCE, TAG_SLAVEMASTER, \
8                          MPI_COMM_WORLD, &mpiStatus);
9       checkMpiError(retCode, "MPI_Recv", __LINE__);
10      mpiSlaveRank = mpiStatus.MPI_SOURCE;
11      //
12      // we have indeed received piggybacked results; copy them to the final destination in c
13      if (dataSlaveMaster.blockA != -1 && dataSlaveMaster.blockB != -1) {
14          for(int ii=0; ii<SLICE; ii++)
15              memcpy( &(c[(dataSlaveMaster.blockA*SLICE+ii)*SIZE+dataSlaveMaster.blockB*SLICE]), \
16                      &(dataSlaveMaster.c2[ii*SLICE]), SLICE*sizeof(float) );
17      }
```

```
18      //
19      // send termination data to slave
20      dataMasterSlave.blockA = dataMasterSlave.blockB = -1;
21
22      retCode = MPI_Send(&dataMasterSlave, 1, MPI_DATA_MASTERSLAVE, mpiSlaveRank, TAG_MASTERSLAVE, \
23                         MPI_COMM_WORLD);
24      checkMpiError(retCode, "MPI_Send", __LINE__);
25    }
26    //...
27  } // END MASTER
```

Code Excerpt 4.3: MPI-Only – Master Phase 2.

A Slave is simpler than the Master – see Code Excerpt 4.4. The Slave loop begins by asking the Master for work (**lines 6 to 10**); at this stage it may already have executed previous work and so, while asking for more, it sends "piggybacked" results to the Master; or, it may be the first time it is asking for work and so it sends "`-1`" work offsets to inform the Master of this situation. In **line 15**, the Slave receives the potential work coming from the Master. Then, the Slave checks if what has received from the Master is indeed more work or is a termination message (**line 21**). If it is not a termination message, the Slave will submit the sub-matrices received to the ATLAS library (**line 29**), to be multiplied. The next time the Slave asks the Master for work, the matrix product results and the work offsets are sent back "piggybacked". If it is a termination message coming from the Master, then the Slave terminates its work (**line 33**).

```
1   else { // SLAVE
2        dataSlaveMaster.blockA = dataSlaveMaster.blockB = -1;
3
4       while (1) {
5
6         //
7         // ask work from master, with eventual piggybacked results
8         //
9         retCode = MPI_Send(&dataSlaveMaster, 1, MPI_DATA_SLAVEMASTER, 0, TAG_SLAVEMASTER, MPI_COMM_WORLD);
10        checkMpiError(retCode, "MPI_Send", __LINE__);
11
12        //
13        // receive work data from master
14        //
15        retCode = MPI_Recv(&dataMasterSlave, 1, MPI_DATA_MASTERSLAVE, 0, TAG_MASTERSLAVE, MPI_COMM_WORLD, \
16                           MPI_STATUS_IGNORE);
17        checkMpiError(retCode, "MPI_Recv", __LINE__);
```

```
18        //
19        // check if its actual work (not termination)
20        //
21        if (dataMasterSlave.blockA != -1 && dataMasterSlave.blockB != -1) {
22
23            dataSlaveMaster.blockA = dataMasterSlave.blockA;
24            dataSlaveMaster.blockB = dataMasterSlave.blockB;
25
26            //
27            // multiply, using ATLAS library
28            //
29            cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasTrans, SLICE, SLICE, SIZE, 1.0, \
30                        dataMasterSlave.a, SIZE, dataMasterSlave.b, SIZE, 0.0, dataSlaveMaster.c2, SLICE);
31        }
32        else
33           break;
34
35     } // while(1)
36 } // END SLAVE
```

Code Excerpt 4.4: MPI-Only – Slave Code.

4.2.1 Test Deployment

The evaluation of the MPI-Only approach was done using all the 4 nodes of the IPB's cluster *datacenter* subset (characterized at section 2.6), each one with a Quad-core CPU, for a total of 16 cores. It was decided to have, in each node, one MPI process per each core, after preliminary tests that showed this configuration to be the most performant. As such, the node `compute-4-0` hosted the Master and 3 Slaves, and nodes `compute-4-1` to `compute-4-3` hosted 4 Slaves each, for a total of 1 Master and 15 Slaves.

4.2.2 Memory Issues

As already stated, all tests were done with matrices of order $n \in \{8K, 16K, 24K\}$. However, for $n = 24K$ and slice $\geq 2K$, the RAM of the cluster node that hosts the Master process it's simply not enough. This is explained next, for $slice = 2K$.

Each $24K$ matrix takes 2.25Gb of RAM (recall section 3.1). In addition to the 3 matrices A, B and C there is a structure `dataMasterSlave_t` that demands 375Mb to take a sub-matrix of A and a sub-matrix of B; there is also a structure `dataSlaveMaster_t` that requires roughly 16Mb to hold a sub-matrix of C. Thus, the memory space needed by the

Master alone amounts to roughly 7.14Gb. A Slave only needs to allocate 375Mb for a structure `dataMasterSlave_t` and approximately 16Mb for a structure `dataSlaveMaster_t`; however, these values must be tripled or quadrupled, depending on the number of Slaves per node. Table 4.2 synthesizes the specific memory requirements of the individual Master and Slave processes, as well as the overall memory consumption in the involved cluster nodes, when $n = 24K$ and $slice = 2K$.

Table 4.2: Memory Consumption for $n = 24K$ and $slice = 2K$ (Gb).

	matrix A	matrix B	matrix C	structure `dataMasterSlave_t`	structure `dataSlaveMaster_t`	Total
Master	2.25	2.25	2.25	0.375	0.015625	7.140625
Slave	-	-	-	0.375	0.015625	0.390625
`compute-4-0`	2.25	2.25	2.25	$(1+3) \times 0.375$	$(1+3) \times 0.015625$	8.3125
`compute-4-[1-3]`	-	-	-	4×0.375	4×0.015625	1.5625

It thus becomes clear that, with 8Gb of RAM and only 1Gb for disk swap, the node that hosts the Master has insufficient resources to execute our MPI-Only Matrix Product, when $n = 24K$ and $slice = 2K$. This scenario becomes worse when $slice = 4K$.

So, in order to solve this issue, the MPI implementation had to be slightly modified to support the using of memory mapped files through *mmap*: a POSIX-compliant Unix system call that maps files into memory (file contents are not entirely read from the disk; the actual reads from disk are performed in a "lazy" manner, after a specific location is accessed [Ker12]). The following section explains how the MPI-Only version uses *mmap*.

4.2.3 Using MMAP

The general algorithm is still the same as the one without *mmap*; only this time, instead of allocating the matrices in the "usual" way (with *malloc*), they are created in files that are mapped in main memory (refer to Appendix A.2 for the complete code implementation). For instance, when using the *mmap* technique the allocation and initialization code of the

matrix A turns from:

```
1  a=(float *)malloc(sizeof(float)*SIZE*SIZE);
2  if (a == NULL) { perror("malloc"); exit(errno); }
3  initMatrixA(a);
```

into

```
1  // alloc and initialize matrices
2  //
3  fd_a=open("/state/partition1/tmp/matrixA.bin", O_CREAT | O_TRUNC | O_RDWR, 00700);
4  if (fd_a < 0) { perror("open"); exit(errno);}
5  offset=lseek(fd_a, sizeof(float)*SIZE*SIZE-1, SEEK_SET);
6  if (offset != sizeof(float)*SIZE*SIZE-1) { perror("lseek"); exit(errno); }
7  retValue=write(fd_a, &dummy, 1);
8
9  if (retValue != 1) { perror("write"); exit(errno); }
10
11 a = mmap(0, sizeof(float)*SIZE*SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd_a, 0);
12 if (a == MAP_FAILED) { perror("mmap"); exit(errno); }
13
14 initMatrixA(a);
```

In the code above, a specific matrix file is created ("`matrixA.bin`", at **line 3**); then, the file is expanded to receive the matrix, setting its offset to the desired size (**line 5**) and writing a single byte (**line 7**); next (**line 11**), the file is mapped in memory and bound to a pointer; after this, the file can be accessed as a "normal" array, during its initialization (**line 14**) and throughout the rest of the code; thus, the remaining of the code is exactly the same as in the first MPI implementation.

Note that, for performance reasons, the matrices files are created in the local disk of the node that hosts the Master (**line 3**), thus avoiding the usage of possible network file systems exported by the *frontend*.

By exploring *mmap* it then becomes possible to evaluate the Matrix Product with matrices of order $n = 24K$, and $slice = 2K$ or $slice = 4K$. In fact this MPI version was solely used for that purpose; the other (lighter) combinations of n and $slice$ were evaluated using the initial MPI version.

4.2.4 Evaluation Results

The *malloc* based version was always used, except for order $24K$ and slices $2K$ and $4K$, in which case the *mmap* based version was used. It should be noted that the *mmap* version doesn't have a significant performance degradation (for the same combinations of order and slice) when compared to the *malloc* version; thus it is not unfair to mix the results of both versions.

The results are presented in Table 4.3, which restrains to the best results.

Table 4.3: MPI-Only Evaluation Results.

Order n	$8K$	$16K$	$24K$
Slice	$2K$	$4K$	$4K^*$
Time (s)	$32,55s$	$130,51s$	$536,34s$

**mmap* results

The best results for orders $8K$, $16K$ and $24K$ are obtained when using the slices $2K$, $4K$ and $4K$, respectively. This translates into 16, 16 and 36 tasks, in accordance to the metric $(n/slice)^2$ introduced in section 4.1. Note that these task numbers are those that fit more closely to the number of slave processes (15), thus ensuring that all Slaves are as evenly busy as possible, while minimizing message exchanges.

4.3 MPI-with-OpenCL

The MPI-with-OpenCL approach is a hybrid approach that uses: 1) MPI to spawn worker (Slave) processes and to handle communications between them, and 2) OpenCL to allow the workers to exploit the computing devices of their hosting nodes. This is one way to surpass a limitation of the original OpenCL specification, by which an OpenCL program is only able to access the computing devices of the node where its host component starts (another way is using clOpenCL, to be discussed in section 4.4).

Figure 4.4 shows how the Master-Slave architecture already used in the MPI-Only approach, may now be deployed to fully exploit the OpenCL devices offered by the testbed cluster. Thus, there's a MPI Slave process for each device used, in each node; a Slave will interact with its specific local device through OpenCL. Therefore, it is as if many OpenCL applications (the Slaves) were launched and executed at the same time. The Slaves are given tasks by the Master that, like in the MPI-Only approach, shares a node with some Slaves.

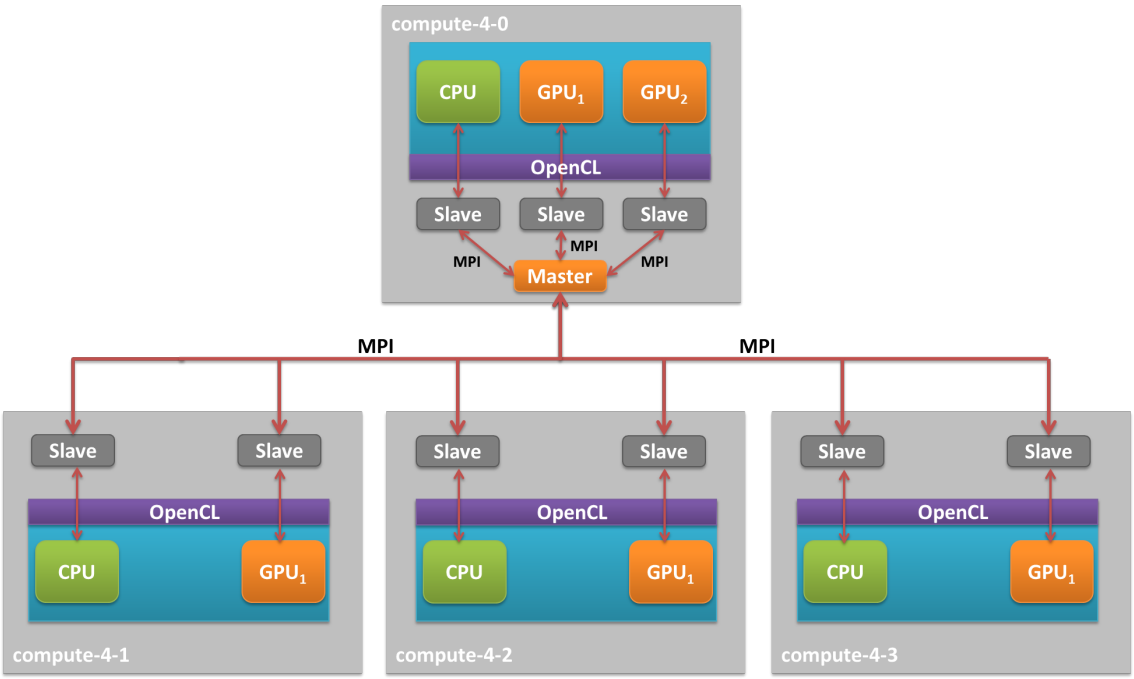

Figure 4.4: Master-Slave Architecture of the MPI-with-OpenCL approach.

For this approach a matrix multiplication OpenCL kernel is needed instead of a matrix multiplication function. The kernel will be executed by all OpenCL devices, with different input data (following the Single Program Multiple Data (SPMD) model). Two different OpenCL kernels were used, with a strong influence on the evaluation results – see section 4.3.3.

In the implementation of the MPI-with-OpenCL approach (see the complete code in Appendix A.2) several things had to be done differently from the MPI-Only approach.

To start with, when using OpenCL in a cluster environment, it is necessary to collect information about the platforms and devices available in the cluster in order to select those to be exploited. For this purpose a query operation was implemented, being triggered by the command line argument "`-q`" of the developed test application – see Code Excerpt 4.5. The full code of the `clQueryPlatformsAndDevices` function invoked in **line 4** is available in Appendix A.2 (it is standard OpenCL code for platforms and devices querying).

```
1 //...
2 if ( !strcmp(argv[1], "-q") ) {
3     if (taskId > 0) // SLAVE
4         clQueryPlatformsAndDevices(processorName);
5 }
6 //...
```

Code Excerpt 4.5: MPI-with-OpenCL – Slave OpenCL Platform and Devices Querying.

This querying procedure is done only by the Slaves, since these are the MPI processes that are going to interact with the devices. The local (Slave specific) results of the queries are formatted and appended to a device file (named "`Devicefile`", by default). In our testbed cluster, when using all OpenCL devices available with a Slave per device, the content of this file is as follows:

```
1 compute-4-0.local:0:0:AMD Accelerated Parallel Processing: Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00GHz
2 compute-4-0.local:1:0:NVIDIA CUDA: GeForce GTX 460
3 compute-4-0.local:1:1:NVIDIA CUDA: GeForce GTX 460
4 compute-4-1.local:0:0:NVIDIA CUDA: GeForce GTX 460
5 compute-4-1.local:1:1:AMD Accelerated Parallel Processing: Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00GHz
6 compute-4-2.local:0:0:AMD Accelerated Parallel Processing: Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00GHz
7 compute-4-2.local:1:0:NVIDIA CUDA: GeForce GTX 460
8 compute-4-3.local:0:0:AMD Accelerated Parallel Processing: Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00GHz
9 compute-4-3.local:1:0:NVIDIA CUDA: GeForce GTX 460
```

Thus, for each cluster node, there are as much lines in the file as slaves running in the node. Each line is a sequence of 5 fields (separated by ":") whose name is self-explanatory: `processorName:platformID:deviceID:platformName:deviceName`.

When not in query mode the Slaves will react based on the supplied device file, and the Master will operate exactly in the same manner as in the MPI-Only approach[9]: the Master allocates and initializes the matrices A, B and C, and then the two same phases

[9] In fact, the Master code is exactly the same as before.

follow (phase 1 - dynamic work distribution by the Slaves, while collecting results from them; phase 2 - Slaves termination, while collecting (possible) remaining results).

However, a Slave now operates differently. Before asking the Master for work (as it did on the MPI-Only approach), a Slave needs to select and bind to a device of its own. A Slave does this based on the device file previously created, whose path is passed to the hybrid MPI executable through the command line. A Slave will select the device whose line (in the device file) matches its MPI *rank* (note that both file line numbers and MPI slave ranks start at 1[10]). For this to be effective, the MPI machine file must be created accordingly. For instance, in our cluster the following machine file matches the device file previously shown:

```
1 compute-4-0:3
2 compute-4-1:2
3 compute-4-2:2
4 compute-4-3:2
```

The processing of the device file by a Slave is shown in Code Excerpt 4.6. In **line 4**, `readDeviceFile(taskId, &argc_file, &argv_file, argv[2])` will read the `taskID`'th line of the device file given in `argv[2]`, returning the full tokenized line in `argv_file` (note that `taskID` is the Slave rank). The Slave then grabs the *platform id* and the *device id* (**lines 10 and 11**), that are needed for the initialization of its OpenCL device (**line 12**). This initialization also includes the usual OpenCL proceedings, like creating a context and buffers, and setting the kernel and its arguments.

```
1  else { // SLAVE
2     // read my platform and device from device file
3     int argc_file=0; char **argv_file=NULL;
4     readDeviceFile(taskId, &argc_file, &argv_file, argv[2]);
5
6     if (argc_file == 0){
7         printAlways3("[SLAVE %d]: aborting: file %s absent, empty or malformed\n", taskId, argv[2]);
8     }
9     else {
10        int p = atoi(argv_file[1]);
11        int d = atoi(argv_file[2]);
12        clInitPlatformsAndDevices(processorName,p,d);
```

Code Excerpt 4.6: Slave device file processing and OpenCL initialization.

[10] The Master *rank* is zero.

The next steps (Code Excerpt 4.7) are exactly the same as in the MPI-Only approach, except this time the `multMatrix` function (**line 32**) doesn't perform the actual multiplication; instead, it contains the necessary host-side OpenCL code to send the input data to the device, to trigger the kernel execution, and to collect the output data (see Code Excerpt 4.8). The result of the sub-matrices multiplication is sent back to the Master in the next work request (**line 18** in Code Excerpt 4.7), again like in the MPI-Only approach.

```
13        dataSlaveMaster.blockA = dataSlaveMaster.blockB = -1;
14        while (1)
15        {
16           // ask work from master, with eventual piggybacked results
17           retCode = MPI_Send(&dataSlaveMaster,1,MPI_DATA_SLAVEMASTER,0,TAG_SLAVEMASTER,MPI_COMM_WORLD);
18           checkMpiError(retCode, "MPI_Send", __LINE__);
19
20           // receive work data from master
21           retCode = MPI_Recv(&dataMasterSlave, 1, MPI_DATA_MASTERSLAVE, 0, TAG_MASTERSLAVE, \
22                              MPI_COMM_WORLD, MPI_STATUS_IGNORE);
23           checkMpiError(retCode, "MPI_Recv", __LINE__);
24
25           // check if its actual work (not termination)
26           if (dataMasterSlave.blockA != -1 && dataMasterSlave.blockB != -1){
27
28               dataSlaveMaster.blockA = dataMasterSlave.blockA;
29               dataSlaveMaster.blockB = dataMasterSlave.blockB;
30               // multiply
31               mulMatrix(dataMasterSlave.a, dataMasterSlave.b, dataSlaveMaster.c2);
32           }
33           else
34               break;
35        } // while (1)
36     } // else (argc_file > 0)
37  } // else SLAVE
```

Code Excerpt 4.7: MPI-with-OpenCL – Slave main loop.

```
38   void mulMatrix(float *a, float *b, float *c){
39     cl_int result; cl_event event[3];
40     result = clEnqueueWriteBuffer(GLOBAL_command_queue, GLOBAL_bufferA, CL_FALSE, 0, sizeof(float)*SLICE \
41                                   *SIZE, a, 0, NULL, &event[0]);
42     clTestSuccess("clEnqueueWriteBuffer", result);
43
44     result = clEnqueueWriteBuffer(GLOBAL_command_queue, GLOBAL_bufferB, CL_FALSE, 0, sizeof(float)*SIZE \
45                                   *SLICE, b, 0, NULL, &event[1]);
46     clTestSuccess("clEnqueueWriteBuffer", result);
47
48     result = clEnqueueNDRangeKernel(GLOBAL_command_queue, GLOBAL_kernel, 2, NULL, GLOBAL_global_work_size,\
49                                     GLOBAL_local_work_size, 2, event, &event[2]);
50     clTestSuccess("clEnqueueNDRangeKernel", result);
51
52     result = clEnqueueReadBuffer(GLOBAL_command_queue, GLOBAL_bufferC, CL_TRUE, 0, sizeof(float)*SLICE \
53                                  *SLICE, c, 1, &event[2], NULL);
54     clTestSuccess("clEnqueueReadBuffer", result);
55  }
```

Code Excerpt 4.8: MPI-with-OpenCL – Host-side code of the matrix product.

4.3.1 Test Deployment

Figure 4.4 shows all OpenCL devices usable in the testbed cluster, and how MPI and OpenCL may be combined to fully exploit them. As mentioned at section 2.6, `compute-4-0` has 3 devices available (1 CPU and 2 GPUs) while the rest of the cluster nodes have 2 devices each (1 CPU and 1 GPU). Therefore a total of 9 OpenCL devices are available to support this approach. However, as explained in section 4.3.3, two different configurations were evaluated.

4.3.2 Memory Issues

The same insufficient memory issues that emerged in the MPI-Only approach, with matrices of order $24K$, and *slice* $2K$ or $4K$, were also faced in this approach. The solution was again to create the matrices in the file system and then use the *mmap* primitive to map them in RAM, following the same method as in the MPI-Only approach.

4.3.3 Evaluation Results

The evaluation of the MPI-with-OpenCL approach was done with two different matrix multiplication kernels: a naive kernel, and an optimized one. At the same time, two different deployment configurations were tested: one fully utilizing the OpenCL devices of the cluster (4 CPUs and 5 GPUs, requiring 9 Slaves), and another one using solely the GPU devices (thus requiring 5 Slaves). This unfolding (both in kernels and deployments) was done in order to assess the possible influence that a set of "slow" devices (the CPUs) could have when operating in conjunction with "fast" devices (the GPUs).

In the next section the kernels are discussed and their evaluation results presented.

Naive Kernel

Code Excerpt 4.9 shows the kernel used to perform the first evaluation of the MPI-with-OpenCL approach. It is a simple/naive kernel, easy to understand.

In this kernel, `subA` and `subB` are the A and B sub-matrices to be multiplied, `subC` will store the results, `size` is the order n of the original A and B square matrices and `slice` is the parameter introduced in the begining of this chapter (section 4.1).

```
1  __kernel void matrix_mult(const int size, const int slice, __global float *subA, \
2                            __global float *subB, __global float *subC){
3
4                            int i, j, k; float v=0;
5
6                            i = get_global_id(0); j = get_global_id(1);
7                            for(k=0; k<size; k++)
8                                v += subA[i*size+k] * subB[j*size+k];
9                            subC[i*slice+j] = v;
10 }
```

Code Excerpt 4.9: Naive Kernel for sliced matrix multiplication.

The kernel works as follows: the `__kernel` qualifier indicates that the function `matrix_mult` is to be run on an OpenCL device; when this function is called from the host code, it will generate a grid of threads on the device; the keyword `__global` designates that the input matrices are in global memory; the keywords `get_global_id(0)` and `get_global_id(1)` refer to the indices of a thread inside the running kernel (since all threads execute the same kernel code, there needs to be a mechanism to allow them to differentiate themselves and determine what part of the data structure they are supposed to work on); each invocation of the kernel uses the two thread indices to 1) identify the row of `subA` and the column of `subB` that are going to be targeted by a dot product operation in the "`for`" loop, and 2) to set the result element in matrix *subC*.

The evaluation results with the naive kernel are shown in Table 4.4.

The best results for orders $8K$, $16K$ and $24K$ are obtained when using the slices $2K$, $4K$ and $4K$, respectively like in the MPI-Only approach (the same number of tasks is involved – 16, 16 and 36, respectively – and this number still provides the best matching

Table 4.4: MPI-with-OpenCL Evaluation Results – naive kernel.

Order n	$8K$	$16K$	$24K$
Slice	$2K$	$4K$	$4K^*$
9 Devices - Time (s)	$42,45s$	$236,94s$	$840,03s$
5 GPUs - Time (s)	$25,01s$	$212,77s$	$563,50s$

* *mmap* results

(in excess) with the number of Slaves).

The results are not better than the MPI-Only results (except for order $8K$ and slice $2K$ with 5 GPUs) and the differential between the two approaches is bigger when using CPU devices mixed with GPUs. However, it should be stressed that the MPI-Only approach uses the ATLAS matrix multiplication function, which was tuned to the particular CPUs of the cluster. Thus, for the comparison to be fair, the results of the MPI-with-OpenCL approach with the naive kernel should be compared to a MPI-Only version with a naive matrix multiplication function. An alternative is to improve the performance of the MPI-with-OpenCL approach by using another, more optimized kernel (see below).

Optimized Kernel

One way to increase the execution speed of an OpenCL kernel is to take advantage of faster memory types available in the OpenCL memory hierarchy. The memory used in the naive kernel is global memory, the slowest level of OpenCL memory. However, global memory is shared by all device threads, which leads to simpler kernels (like the naive). The fastest OpenCL memory level is local memory that is only visible to the threads of the same work-group (Appendix B.1.2 describes these concepts) and thus usually requires more sophisticated kernels to be properly exploited.

Code Excerpt 4.10 shows a matrix multiplication kernel that explores local memory. Since local memory has limited size, it is necessary to implement yet another level of sliced matrix multiplication in the kernel, to address subsets (blocks) of the original

slices. The work-item threads must explicitly fill local memory before it can be used, being this the first task of the kernel (**lines 10 to 26**). A thread loads one element of the block from global memory and then a synchronization barrier waits for all the other threads to do the same (**line 29**). The memory write pattern is heavy on local memory and only writes once to the global memory at the end (**line 40**). The value 16 for the constant `BLOCK_SIZE` was defined to be the same as the `local_work_size` parameter of the `clEnqueNDRangeKernel` (see Code Excerpt 4.8). In turn, this value was chosen in accordance with the *slice* parameter (which must be multiple of the `local_work_size`) and the hardware characteristics of the GPUs used (NVIDIA GTX460).

```
1  #define BLOCK_SIZE 16
2  __kernel void matrix_mult(int size, int slice, __global float *subA, __global float *subB, \
3                            __global float *subC) {
4    int wA=size; int wB=slice;
5    int bx = get_group_id (0); // 2D Thread ID x
6    int by = get_group_id (1); // 2D Thread ID y
7    int tx = get_local_id (0) ; // 2D local ID x
8    int ty = get_local_id (1) ; // 2D local ID y
9
10   // first and last sub-matrix of A and B for this block
11   int aBegin = wA * BLOCK_SIZE * by ;
12   int aEnd = aBegin + wA - 1 ;
13   int aStep = BLOCK_SIZE ;
14   int bBegin = BLOCK_SIZE * bx ;
15   int bStep = BLOCK_SIZE * wB;
16
17   float Csub = 0.0 ;
18   // Iterate over all sub-matrices of A and B
19   for(int a = aBegin , b = bBegin ; a <= aEnd ; a+=aStep, b+=bStep){
20      // Static work-group local allocations
21     __local float As[BLOCK_SIZE][BLOCK_SIZE] ;
22     __local float Bs[BLOCK_SIZE][BLOCK_SIZE] ;
23
24     // Each thread loads one element of the block from global memory
25     As[ty][tx] = subA[ a + wA * ty + tx] ;
26     Bs[ty][tx] = subB[ b + wB * ty + tx] ;
27
28     // Barrier to synchronize all threads
29     barrier (CLK_LOCAL_MEM_FENCE) ;
30     // Now the local sub-matricies As and Bs are valid
31
32     // Multiply the two sub-matrices. Each thread computes one element of the block sub-matrix
33     for( int k = 0 ; k < BLOCK_SIZE ; ++k )
34        Csub += As[ty][k] * Bs[k][tx] ;
35
36     // Barrier to synchronize all  threads before moving
37     barrier (CLK_LOCAL_MEM_FENCE) ;
38   }
39   int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx ;
40   subC[ c + wB * ty + tx ] = Csub ; // write to global memory
41 }
```

Code Excerpt 4.10: Kernel taking advantage of the varying speeds in memory hierarchy.

Table 4.5 shows the execution times with the new kernel for the best combinations of order and slice (the same combinations used with the naive kernel). The speedups relative to the naive kernel are also shown.

Table 4.5: MPI-with-OpenCL Evaluation Results – optimized kernel.

Order n	$8K$	$16K$	$24K$
Slice	$2K$	$4K$	$4K^*$
9 Devices - Time (s)	$124,89s$	$443,65s$	$2735,28s$
Speedup to Naive	$0,34$	$0,53$	$0,31$
5 GPUs - Time (s)	$16,64s$	$65,99s$	$236,71s$
Speedup to Naive	$1,50$	$3,22$	$2,38$

* *mmap* results

As may be observed, the performance impact of the new kernel is completely different between the two device deployments. When CPUs are involved, the execution time deteriorates to the point that they become significantly worse than the times produced by the naive kernel; this is clearly shown by the speedup values (< 1). With GPUs only, the new kernel brings improvements, as shown by the speedup values attained. The improvements are stronger with bigger matrices and slices: the sub-matrices produced from matrices A and B are also bigger, thus more blocks are created inside the optimized kernel; since these blocks are transferred to local memory, it is faster to work on these big local sets than it is to write several fine grained subsets to the local memory.

The results achieved with both the naive kernel and the optimized one, seem to imply that the CPU devices are too slow when compared to GPUs, thus delaying the parallel execution. This is aggravated with the last kernel, which is particularly optimized for the GPUs used.

4.4 clOpenCL

As mentioned before (see sections 2.3.4 and 2.4), the clOpenCL approach is a hybrid approach that allows multi-threaded (POSIX threads) based OpenCL applications to surpass

the limitations imposed by the original OpenCL specification, allowing local OpenCL devices to be complemented by external devices (e.g., scattered across cluster nodes) in order to increase performance.

Figure 4.5 shows a deployment of the clOpenCL architecture in the testbed cluster: the host application component (*Host APP*) launches as much threads as devices available (both local and external); each thread interacts with its own device via the clOpenCL library; interaction with local devices is direct, through the local OpenCL routine; for remote nodes the clOpenCL library uses OpenMX to handle communication with proxy daemons that will interact with their own local OpenCL devices.

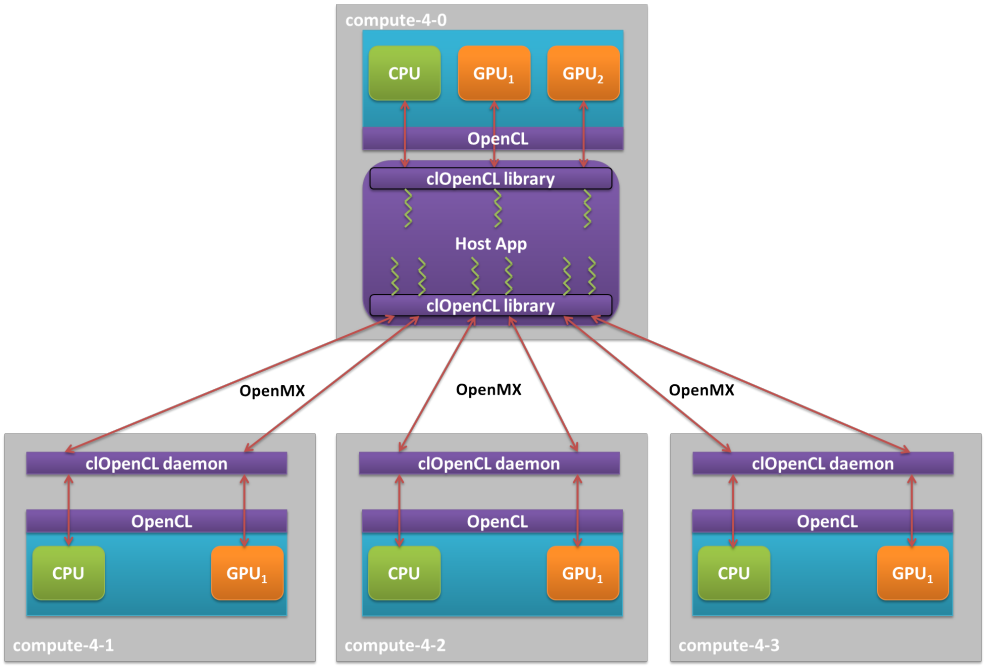

Figure 4.5: A deployment of the clOpenCL Architecture.

To evaluate the matrix multiplication in this kind of deployment, it was first necessary to fit the test application to the POSIX threads framework. The host component of the test application thus became multi-threaded, following a dynamic model of work (auto-)-assignment: a thread is created for each OpenCL device involved in the matrix product; while there is work available, each thread selects mutually exclusive sub-matrices pairs

($< subA, subB >$), queues them to its device, triggers the kernel execution and de-queues the results (a sub-matrix $subC$). Figure 4.6 represents this non-hierarchical operation model (in opposition to the hierarchical Master-Slave model of the previous MPI-Only approach).

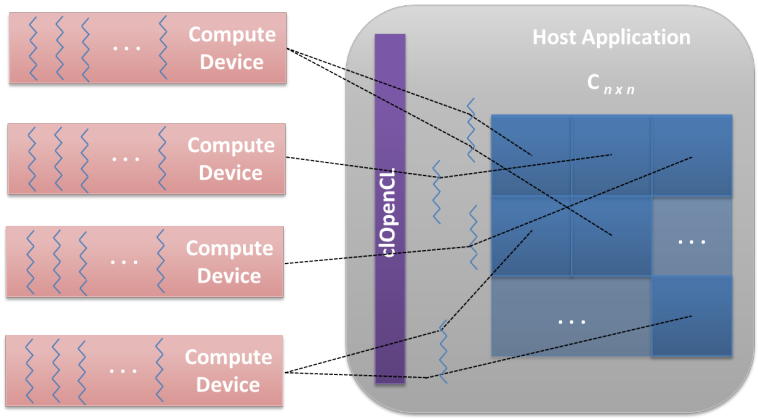

Figure 4.6: Host application execution method.

Similarly to the MPI-with-OpenCL test application, the clOpenCL instance also has a query option to discover all local and remote OpenCL platforms and devices available. This mechanism takes advantage of the special platform attribute `CL_PLATFORM_HOSTNAME`, a specific extension of OpenCL introduced by clOpenCL (see section 2.4.2). So, while in the MPI-with-OpenCL approach each Slave queries its own local platforms and devices, in clOpenCL only the main thread of the *Host APP* does the querying, since the OpenCL primitives invoked (`clGetPlatformIDs` and `clGetDeviceIDs`) are now cluster-aware. Afterwards, the specific set of platforms and devices to be used are passed to the application explicitly through the command line or via a properly formatted file. These platforms and devices are initialized by the main thread (in order to collect their platform and device IDs) and then a worker thread will be created for each device – see Code Excerpt 4.11.

```
1  //...
2  num_threads=GLOBAL_num_devices;
3  //...
4  for(d=0; d<GLOBAL_num_devices; d++){
5      //...
6      if(pthread_create(&GLOBAL_dev_thread[d], NULL, dev_thread_routine,(int *)(long)d) != 0){
7         perror("pthread_create");
8         exit(1);
9      }
10 }
11 //...
```

Code Excerpt 4.11: clOpenCL – *Host App* Thread creation.

Each worker thread will execute the same routine (`dev_thread_routine`, **line 6**), where it will start by creating the usual OpenCL objects (a context specific to the thread's platform and device, a command queue and buffers), compiling the matrix multiplication kernel and setting the kernel arguments. Right after, a worker thread enters a work auto-assignment loop (see Code Excerpt 4.12), where it picks up sub-matrices of A and B yet to be multiplied and invokes the OpenCL code necessary to do so (**lines 4 to 23**). Tests are made in order to check if the sub-matrices that were last sent to the devices can be reused (**lines 4 to 16**), improving the overall application performance.

The OpenCL code inside the loop is very similar to the one used in the MPI-with-OpenCL approach (see code Excerpt 4.7), a good indication about the transparency of the clOpenCL approach.

```
1  get_work(&i, &j);
2  while((i != -1) && (j != -1)){
3
4       if(i != last_i){
5          result = clEnqueueWriteBuffer(command_queue, bufferA, CL_FALSE, 0, \
6                                        sizeof(float)*GLOBAL_size*GLOBAL_slice, GLOBAL_A+GLOBAL_size*i, 0, \
7                                        NULL, &event[0]);
8          cclTestSuccess("clEnqueueWriteBuffer", result);
9       }
10
11      if(j != last_j){
12         result = clEnqueueWriteBuffer(command_queue, bufferB, CL_FALSE, 0, \
13                                       sizeof(float)*GLOBAL_slice*GLOBAL_size, GLOBAL_B+GLOBAL_size*j, 0, \
14                                       NULL, &event[1]);
15         cclTestSuccess("clEnqueueWriteBuffer", result);
16      }
17      result = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL, global_work_size, local_work_size, \
18                                      2, event, &event[2]);
19      cclTestSuccess("clEnqueueNDRangeKernel", result);
20
21      result = clEnqueueReadBuffer(command_queue, bufferC, CL_TRUE, 0, \
22                                   sizeof(float)*GLOBAL_slice*GLOBAL_slice, C2, 1, &event[2], NULL);
23      cclTestSuccess("clEnqueueReadBuffer", result);
```

```
24      for(i2=0; i2<GLOBAL_slice; i2++)
25          for(j2=0; j2<GLOBAL_slice; j2++)
26              GLOBAL_C[(i+i2)*GLOBAL_size+j+j2] = C2[i2*GLOBAL_slice+j2];
27
28      last_i = i;
29      last_j = j;
30      get_work(&i, &j);
31  } // while
```

Code Excerpt 4.12: clOpenCL – *Host App* Thread main loop.

4.4.1 Test Deployment

Figure 4.5 shows all OpenCL devices available in the testbed cluster, and how clOpenCL is able to fully exploit them. A total of 9 OpenCL devices are available. However, like with MPI-with-OpenCL, two distinct configurations were evaluated – se section 4.4.3.

4.4.2 No Memory Issues

The memory issues that occurred in the previous approaches (MPI-Only and MPI-with-OpenCL) when using matrices of order $24K$ and slice $>= 2K$, are now absent. The *Host App* still need to reserve memory for the matrices A, B and C; however, the sub-matrices exchanged with clOpenCL daemons are passed by reference, between the *Host App* threads and the Open-MX communication layer; this is enough to allow the RAM working-set of the Host App to fit in the 8Gb of RAM of the hosting node.

4.4.3 Evaluation Results

The evaluation of the clOpenCL approach was done with the same two different matrix multiplication kernels used on the MPI-with-OpenCL approach: a naive kernel and the GPU-optimized one (see Code Excerpt 4.9 and Code Excerpt 4.10, respectively). Also, the same two different deployment configurations were tested: one with all the OpenCL devices of the cluster (4 CPUs and 5 GPUs, requiring 9 worker threads), and another with all GPU devices (thus requiring 5 worker threads).

Naive Kernel

Table 4.6 presents the evaluation results when using the naive kernel.

Table 4.6: clOpenCL Evaluation Results – naive kernel.

Order n	$8K$	$16K$	$24K$
Slice	$1K$	$2K$	$4K$
9 Devices - Time (s)	$21,78s$	$156,76s$	$489,22s$
9 GPUs - Time (s)	$25,83s$	$197,77s$	$575,87s$

The best results for orders $8K$, $16K$ and $24K$ are now obtained when using the slices $1K$, $2K$ and $4K$, respectively. Thus, for order $8K$ and $16K$ the best slices ($1K$ and $2K$) are different from those of MPI-Only and MPI-with-OpenCL (where the best slices were $2K$ and $4K$). This seems to imply that with faster communications (Open-MX) a finer grain distribution of work (smaller slices) pays off, by preventing slower devices (the CPUs) from getting too much tasks (which delays the overall execution).

In comparison with MPI-with-OpenCL (naive kernel), the execution times improved significantly for the 9 devices scenario, but are very similar with GPUs only; the late alternative seems to imply that, with less efficient kernels, the computation time is dominant over the communication time and so, it is less relevant to use fast communications for work distribution.

Optimized Kernel

Table 4.7 shows the execution times with the optimized kernel and the speedups in relation to the naive kernel. The best combinations of order and slice are the same of the naive kernel. The considerable performance improvements, over the naive kernel, are immediately noticeable, specially with the GPUs-only deployment, which is from ≈ 4 to ≈ 5 times as fast as with the naive kernel.

Table 4.7: clOpenCL Evaluation Results – optimized kernel.

Order n	$8K$	$16K$	$24K$
Slice	$1K$	$2K$	$4K$
9 Devices - Time (s)	$13,06s$	$99,63s$	$511,53s$
Speedup to Naive	$1,67$	$1,57$	$0,96$
5 GPUs - Time (s)	$6,53s$	$39,63s$	$111,53s$
Speedup to Naive	$3,96$	$4,99$	$5,16$

A quick comparison with MPI-with-OpenCL results (Table 4.5) also allows to conclude that clOpenCL improves on the performance of that approach (a global comparison is provided in the next section).

4.5 Results Discussion

With all parallel approaches evaluated, it is now possible to make a general comparison, bringing also into this discussion the previous evaluation on Matlab (from Chapter 3).

Figure 4.7 shows the best execution times obtained by using these parallel approaches, on the testbed cluster of this dissertation. Figure 4.8 also shows the speedups over the "classical" MPI-Only approach.

The MPI-Only approach, that only uses CPUs, it's clearly the slowest approach, despite using the ATLAS BLAS matrix product implementation. This is expected since CPUs are slower than GPUs when performing the Matrix Product, which is why MPI-with-OpenCL is faster (though with GPUs only and a GPU-optimized kernel). When using OpenCL, the clOpenCL is clearly superior performance-wise (again assuming GPUs only and a GPU-optimized kernel). Comparing only the three developed parallel approaches, the performance doubles from MPI-Only to MPI-with-OpenCL, and increases ≈ 3 to ≈ 5 times with clOpenCL – see Figure 4.8.

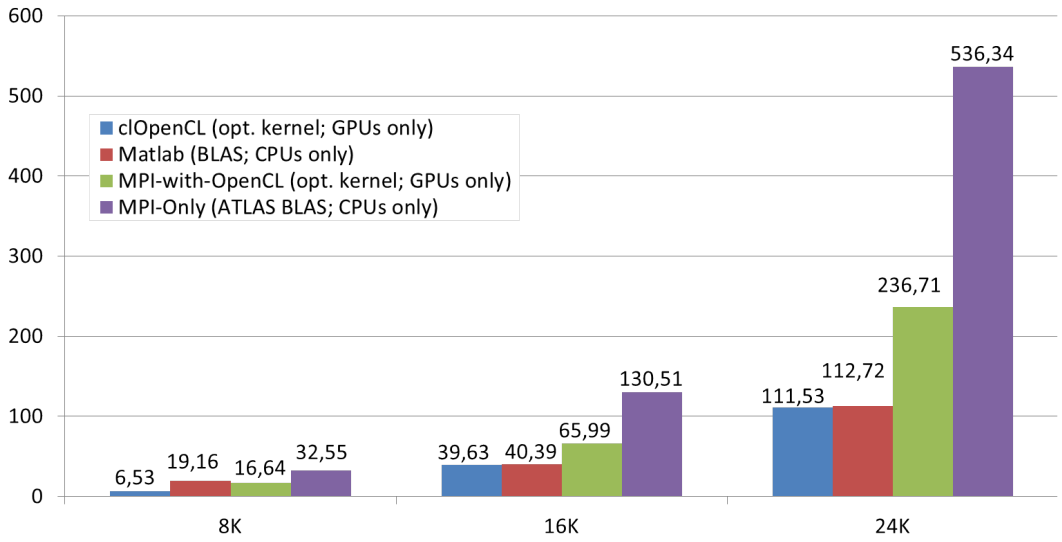

Figure 4.7: Parallel Matrix Product – best execution times (s).

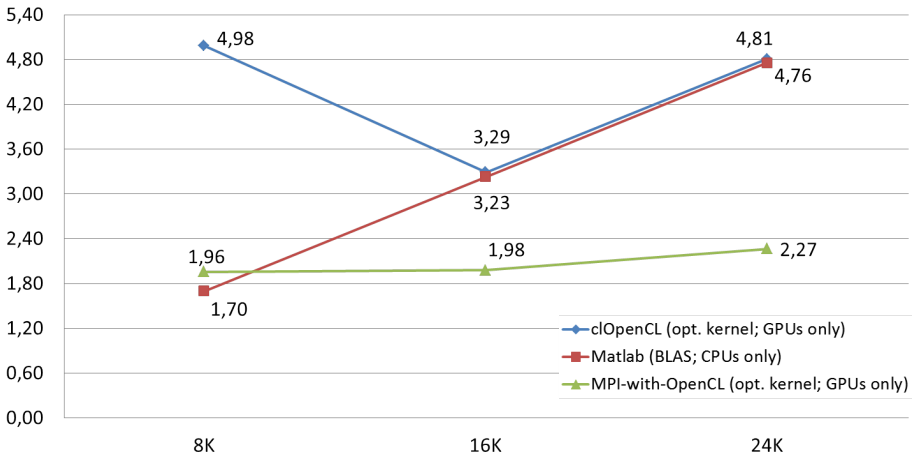

Figure 4.8: Parallel Matrix Product – speedups relative to MPI-Only.

Matlab is a special case. Although it uses only CPUs, its performance is hand-in-hand with clOpenCL (except with small order ($8K$) matrices). However, Matlab is a highly optimized commercial product and so, judging the merits of clOpenCL (in a prototype level (at best) in its current stage) against Matlab is not entirely fair[11].

To complete the picture, the speedups of all parallel approaches against the reference serial implementation (ATLAS based) are also provided in Figure 4.9. As may be observed, clOpenCL is able to out-perform the serial implementation in one order of magnitude ($\approx 10\times$).

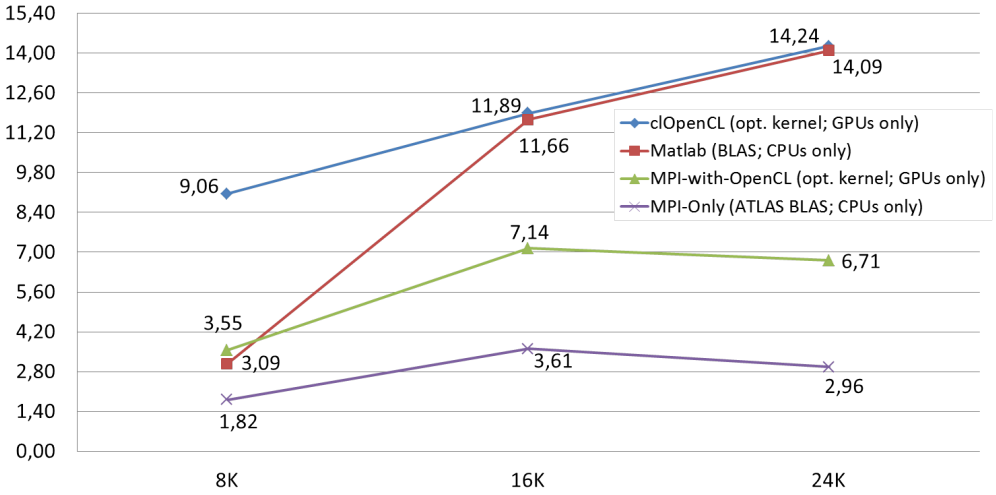

Figure 4.9: Speedups relative to the ATLAS implementation.

[11] It would be of no surprise if using Matlab with GPUs (a scenario that was not tested) produced better results than clOpenCL.

Chapter 5

Conclusions

Computing devices that take advantage of multi-core and many-core technology, are becoming pervasive. With the utilization of GPUs and other co-processors as accelerators, new levels of processing capabilities are added, capable of providing, in many situations, significant performance gains in relation to CPUs. In this context there is a need for algorithms, models and frameworks that are adapted to take advantage of the new heterogeneous parallel environments.

OpenCL is an industry standard that targets the heterogeneity of parallel environments by providing an uniform programming and execution model. It is, however, restrained to the set of parallel devices of a single isolated system.

clOpenCL extends OpenCL to allow the execution of OpenCL-based applications in devices scattered across heterogeneous cluster nodes. While using a Matrix Product application for evaluation purposes, clOpenCL demonstrated its capacity of significant performance increases in comparison to serial and other parallel approaches. At the same time, porting "classical" OpenCL applications to clOpenCL proved to be a transparent and straightforward process.

Other projects have also focused on the same objective, but clOpenCL has two main advantages: it is able to take full advantage of commodity networking hardware through

Open-MX, and programmers/users do not need special privileges or exclusive access to scarce resources to deploy the desired running environment.

5.1 Future Work

The work described in this dissertation was evaluated using a Matrix Product application, a well known embarrassingly parallel case study. clOpenCL should also be evaluated with other (more irregular) tests, covering both performance and OpenCL compliance issues, like the Rodinia [oV12] and Vienna CL [Rup12] benchmark suites, or Image Processing/ Computer Graphics tests (e.g., the pathtracer tool developed in the R&D project that hosted this dissertation). Also, adding BSD sockets (Berkeley sockets) [Inc12] as an alternative communication layer to Open-MX will allow clOpenCL to be used in a broader set of parallel/distributed scenarios.

Bibliography

[AMD12] AMD. x86 open64 compiler suite. `http://developer.amd.com/tools/cpu-development/x86-open64-compiler-suite/`, November 2012.

[AONM11] Ryo Aoki, Shuichi Oikawa, Takashi Nakamura, and Satoshi Miki. Hybrid opencl: Enhancing opencl for distributed processing. In *Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, ISPA '11, pages 149–154, Washington, DC, USA, 2011. IEEE Computer Society.

[ARPS12] Albano Alves, José Rufino, António Pina, and Luís Paulo Santos. dopencl - supporting distributed heterogeneous computing in hpc clusters. *10th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar2012)*, 2012.

[Bar11a] Blaise Barney. Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`, Agosto 2011.

[Bar11b] Blaise Barney. Message passing interface (mpi). `https://computing.llnl.gov/tutorials/mpi/`, Agosto 2011.

[Bar12] Blaise Barney. Posix threads programming. `https://computing.llnl.gov/tutorials/pthreads/`, November 2012.

[Bri12] Todd Brian. Putting multicore processing in context: Part one. http://www.eetimes.com/design/embedded/4006507/Putting-multicore-processing-in-context-Part-One, Setembro 2012.

[BS11] Amnon Barak and Amnon Shiloh. The virtual opencl (vcl) cluster platform. *Abstract in Proc. Intel European Research & Innovation Conf., pp. 196, Leixlip*, October 2011.

[Cen12] AMD Developer Central. Amd core math library (acml). http://developer.amd.com/tools/cpu/acml/pages/default.aspx, Outubro 2012.

[DPnS+10] J. Duato, A. Peña, F. Silla, R. Mayo, and E.S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing and Simulation (HPCS)*, pages 224– 231, July 2010.

[fACR11] Caltech Center for Advanced Computing Research. mpich2 vs open mpi. https://www.cacr.caltech.edu/pipermail/danse-dev/2008-July/000388.html, Setembro 2011.

[fHPSC12] Laboratory for High Performance Scientific Computing. gvirtus general-purpose virtualization service. http://code.google.com/p/gvirtus/, August 2012.

[For94] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[For12] Basic Linear Algebra Subprograms Technical (BLAST) Forum. Basic linear algebra subprograms technical (blast) forum standard. http://www.netlib.org/blas/index.html, Outubro 2012.

[FZRL08] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared. *Grid Computing Environments Workshop, 2008. GCE '08*, November 2008.

[GHK+11] Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[GLA+09] William Gropp, Ewing Lusk, David Ashton, Pavan Balaji, Darius Buntinas, Ralph Butler, Anthony Chan, David Goodell, Jayesh Krishna, Guillaume Mercier, Rob Ross, Rajeev Thakur, and Brian Toonen. Mpich2 user's guide. November 2009.

[GML+11] Francisco Giunta, Raffaele Montella, Giuliano Laccetti, Florin Isaila, and Francisco Javier García Blas. A gpu accelerated high performance cloud computing infrastructure for grid computing based virtual environmental laboratory. *Advances in Grid Computing*, 2011.

[GNU12] GNU. Gsl - gnu scientific library. `http://www.gnu.org/software/gsl/`, Outubro 2012.

[Gog11] Brice Goglin. High-performance message-passing over generic ethernet hardware with open-mx. *Parallel Comput.*, 37(2):85–100, February 2011.

[Gro11] Khronos OpenCL Working Group. *The OpenCL Specification*. 15th edition, 2011.

[Gro12a] William Gropp. Mpi and hybrid programming models - presentation. `http://www.cs.illinois.edu/homes/wgropp/`, August 2012.

[Gro12b] Khronos Group. Khronos group connecting software to silicon. `http://www.khronos.org/`, August 2012.

[Gro12c] Khronos Group. Opencl - the open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl`, August 2012.

[Gro12d] Rocks Group. Rocks: Open-source toolkit for real and virtual clusters. `http://www.rocksclusters.org/`, August 2012.

[IBM] IBM. Message passing library. IBM AIX Parallel Environment Parallel Programming Subroutine Reference.

[Inc12] Trek Inc. Introduction to bsd sockets. http://wiki.treck.com/Introduction_to_BSD_Sockets, November 2012.

[Ker12] Kernel.org. Linux programmer's manual mmap(2). http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html, Outubro 2012.

[KSG12] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. dopencl: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems. In *IPDPS Workshops*, pages 174–186, 2012.

[Lab11a] Sandia National Laboratories. More chip cores can mean slower supercomputing. https://share.sandia.gov/news/resources/news_releases/more-chip-cores-can-mean-slower-supercomputing-sandia-simulation-shows/, Setembro 2011.

[Lab11b] Oak Ridge National Laboratory. Pvm parallel virtual machine. http://www.csm.ornl.gov/pvm/, Setembro 2011.

[Mat12a] MathWorks. Matlab distributed computing server, perform matlab and simulink computations on clusters, clouds, and grids. http://www.mathworks.com/products/distriben/index.html, Outubro 2012.

[Mat12b] MathWorks. Matlab the language of technical computing. http://www.mathworks.com/products/matlab/, Outubro 2012.

[Mat12c] MathWorks. Parallel computing toolbox, perform parallel computations on multicore computers, gpus, and computer clusters. http://www.mathworks.com/products/parallel-computing/, Outubro 2012.

[MCG+11] R. Montella, G. Coviello, G. Giunta, G. Laccetti, F. Isaila, and J. Garcia Blas. A general-purpose virtualization service for hpc on cloud computing: an application to gpus. In *9th International Conference on Parallel Processing and Applied Mathematics*, September 2011.

[MGM+11] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[Mic12] Microsoft. Directx developer center. `http://msdn.microsoft.com/pt-BR/DirectX`, August 2012.

[MPI11] MPICH2. Mpich2. `http://www.mcs.anl.gov/research/projects/mpich2/`, Setembro 2011.

[Mye12] Judith Myerson. Cloud computing versus grid computing. `http://www.ibm.com/developerworks/web/library/wa-cloudgrid/`, November 2012.

[Nvi12] Nvidia. Introducing cuda 5. `http://www.nvidia.com/object/cuda_home_new.html`, August 2012.

[oV12] University of Virginia. Rodinia:accelerating compute-intensive applications with accelerators. `https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page`, November 2012.

[Pac11] P. Pacheco. *An Introduction to Parallel Programming*. An Introduction to Parallel Programming. Elsevier Science, 2011.

[Pro12a] Condor Project. Condor high throughput computing. `http://www.cs.wisc.edu/condor/`, August 2012.

[Pro12b] Ganglia Project. Ganglia monitoring system. `http://ganglia.sourceforge.net/`, August 2012.

[Rup12] Karl Rupp. Viennacl. http://viennacl.sourceforge.net/, November 2012.

[Sac09] Mathew Sacks. Creating virtual clusters with rocks, in the rocks. *Linux Pro Magazine*, 2009.

[Sha06] Amar Shan. Heterogeneous processing: a strategy for augmenting moore's law. http://www.linuxjournal.com/article/8368, January 2006.

[Sou12] Sourceforge. Automatically tuned linear algebra software (atlas). http://math-atlas.sourceforge.net/, Outubro 2012.

[Sta11] StackOverflow. Mpich vs openmpi. http://stackoverflow.com/questions/2427399/mpich-vs-openmpi, Setembro 2011.

[Sul12] Aater Suleman. Parallel programming: Amdahl's law or gustafson's law. http://www.futurechips.org/thoughts-for-researchers/parallel-programming-amdahls-law-gustafsons-law.html, Setembro 2012.

[Sun12] Sun. Sun grid engine. http://gridengine.sunsource.net/, August 2012.

[Tea11] Open MPI Team. Open mpi: Open source high performance computing. http://www.open-mpi.org/, Setembro 2011.

[Tec12] Michigan Tech. What is shared memory? http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/shm/what-is-shm.html, November 2012.

[vA08] Sebastian von Alfthan. Introduction hybrid programming - presentation, August 2008.

[Wik12a] Wikipedia. C99. http://en.wikipedia.org/wiki/C99, August 2012.

[Wik12b] Wikipedia. Cell (microprocessor). http://en.wikipedia.org/wiki/Cell_(microprocessor), August 2012.

[Wik12c] Wikipedia. Compute unified device architecture. `http://pt.wikipedia.org/wiki/Compute_Unified_Device_Architecture`, August 2012.

[Wik12d] Wikipedia. Digital signal processor. `http://en.wikipedia.org/wiki/Digital_signal_processor`, August 2012.

[Wik12e] Wikipedia. Field-programmable gate array. `http://en.wikipedia.org/wiki/FPGA`, August 2012.

[Wik12f] Wikipedia. Heterogeneous computing. `http://en.wikipedia.org/wiki/Heterogeneous_computing`, August 2012.

[Wik12g] Wikipedia. Shared memory. `http://en.wikipedia.org/wiki/Shared_memory`, November 2012.

[Wik12h] Wikipedia. Symetric multiprocessing. `http://en.wikipedia.org/wiki/Symmetric_multiprocessing`, August 2012.

Appendix A

Source Code

A.1 Preliminary Experiments

See `https://beta.estig.ipb.pt/~perform/Preliminary_Experiments`.

A.2 Parallel Approaches

See `https://beta.estig.ipb.pt/~perform/Prallel_Approaches`.

Appendix B

OpenCL Details

B.1 OpenCL Terminology

Knowing the OpenCL terminology helps to understand the definition of the terms used to characterize OpenCL functioning, its standard and specification, alongside its architecture. Following are the main terms used throughout this document.

Buffer: a memory object that stores a linear collection of bytes. Buffer objects are accessible using a pointer in a kernel executing on a device. Buffer objects can be manipulated by the host using OpenCL API calls. A buffer object encapsulates the following information:

- Size in bytes.
- Properties that describe usage information and which region to allocate from.
- Buffer data.

Command-queue: an object that holds commands that will be executed on a specific device. The command-queue is created on a specific device in a context. The commands

are collected in-order, but their execution can be performed in in-order or out-of-order way.

Context: the environment within which the kernels execute, and the domain in which synchronization and memory management is defined. The context includes a set of devices, the memory accessible to those devices, the corresponding memory properties and one or more command-queues used to schedule execution of a kernel(s) or operations on memory objects.

Device: a device is a collection of compute units. A command-queue is used to queue commands to a device. Examples of commands include executing kernels, or reading and writing memory objects. OpenCL devices typically correspond to a GPU, a multi-core CPU, and other processors.

Event: an event object encapsulates the status of an operation such as a command. It can be used to synchronize operations in a context.

Image: a memory object that stores a two-dimensional or three-dimensional structured array. Image data can only be accessed with read and write functions. The read functions use a sampler.

Kernel: a *kernel* is a funtion declared in a program and executed on an OpenCL device. A kernel is identified by the `__kernel` qualifier applied to any function defined in a program.

Platform: it comprises the host plus a collection of devices managed by the OpenCL framework that allow an application to share resources and execute kernels on devices in the platform.

Program: an OpenCL program consists of a set of kernels. Programs may also contain auxiliary functions called by the `__kernel` functions and constant data.

Sampler: an object that describes how to sample an image when the image is read in the kernel. The image read functions take a sampler as an argument. The sampler

specifies the image addressing-mode, i.e., how out-of-range image coordinates are handled, the filter mode, and whether the input image coordinate is a normalized or unnormalized value.

B.1.1 OpenCL Standard

Developing a programming standard that satisfies a range of necessities and requirements is not an easy task. Nevertheless the Khronos consortium made it happen, addressing these concerns with OpenCL. They developed an API that is general enough to run on significantly different architectures while being capable of adapting to each hardware platform and still obtaining high performance.

If a developer correctly follows the OpenCL *specification* (see section B.1.2), any program previously designed for one vendor will execute on another's hardware. This way, OpenCL, creates portable, vendor and device independent programs.
Talking about the standard it makes sense to refer that, OpenCL is a C with a C++ wrapper API, defined in terms of the C language API. As what happens on CUDA, there are third-party bindings for other languages, such as Java, Python, and .NET. The code that executes on an OpenCL device, called kernel, is written in the OpenCL C language. OpenCL C, as mentioned previously, is a restricted and adapted version of C99, that has appropriate extensions specifically for executing data-parallel code on a variety of heterogeneous devices. OpenCL C was the language used throughout the development of this project.

B.1.2 OpenCL Specification

The OpenCL specification is set in four parts, termed models. Below follows a shortened overview of each of these models.

Platform Model

It defines a high-level representation of any heterogeneous platform used with OpenCL. This model consists of a single *host* (a processor coordinating execution) connected to one or more OpenCL *devices* (processors capable of executing OpenCL C code). A device is divided into *compute units* (CUs), being these further divided into one or more *processing elements* (PEs). It is on the device where streams of instructions (namely *kernels*) execute. The PEs within a compute unit execute a single stream of these instructions as SIMD (Single Instruction, Multiple Data) units or as SPMD (Single Program, Multiple Data) units. This model is shown in Figure B.1.

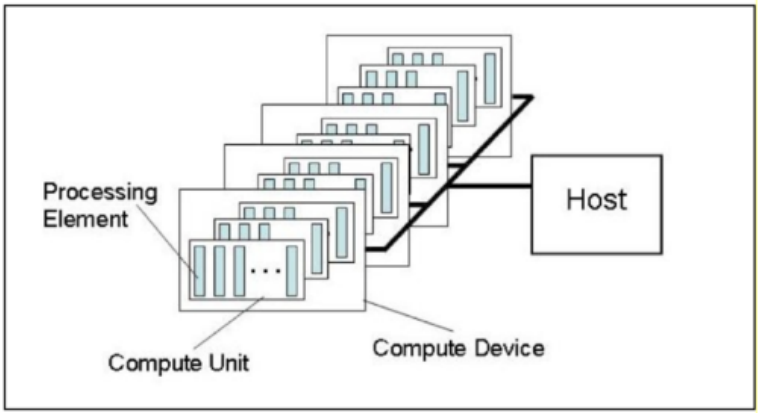

Figure B.1: OpenCL platform

Execution Model

This model defines how the OpenCL environment is configured on the host and how *kernels* are executed on the devices. Execution of an OpenCL program occurs in two parts, namely a collection of one or more *kernels* that execute on the devices and a *host program* that executes on the host. The host program defines the context for the kernels and manages their execution, providing mechanisms for host-device interaction,

and defining a concurrency model used for kernel execution on devices.

When a kernel is submited for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This instance of an executing kernel is called a **work-item** and is identified by its "coordinates" in the index space, which provides a global ID for the work-item. When a kernel is submitted for execution, by its respective primitive, a collection of work-items is created, each of which uses the same sequence of instructions defined by the kernel. While the sequence of instructions is the same, and each work-item executes the same code, the behaviour of each work-item can vary because of branch statements within the code or data selected through the global ID.

Work-items are organized into **work-groups**. Work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned with a unique work-group ID having the same dimensionality as the index space used for the work-items. Also, work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. This is very important in understanding the concurrency in OpenCL; OpenCL only assures that the work-items within a work-group execute concurrently and share processor resources on the device. Hence, one can never assume that work-groups or kernel invocations execute concurrently.

In OpenCL the supported index space, mentioned above, is called **NDrange**. NDRange is an N-dimensional index space, where, currently, N can be 1, 2 or 3. This index space is defined by an integer array of length N specifying the extent of the index space in each dimension starting at an offset of zero, by default. Each work-item's global ID and local ID are N-dimensional tuples. See Figure B.2 for a better understanding on the NDRange concept [Gro11, MGM+11].

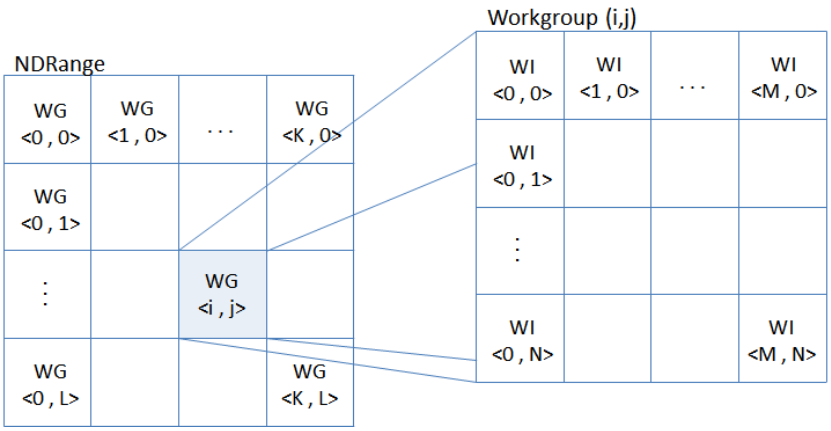

Figure B.2: Representation of the NDRange where Work-items (WI) are grouped in Work-groups (WG).

Memory Model

This model defines the abstract memory hierarchy used by the kernels, regardless of the actual primary memory architecture. It defines four distinct memory regions (see Figure B.3 for a visual representation), being them:

- **Global Memory** – All work-items in all work-groups have permission for read /write access on this memory region. In global memory, work-items can read from or write to any memory object element. Reads and writes to global memory can be cached depending on the capabilities of the device. Whenever data is transferred from the host to the device, the data will reside in global memory. When data is to be transferred back from the device to the host it must also reside in global memory. In order to specify that the data resides in global memory, the keyword `__global` is added to a pointer declaration;

- **Constant Memory** – During the execution of a kernel, this memory region of global memory remains constant. The host allocates and initializes memory objects placed into constant memory. Work-items have read-only access to these objects. Data is mapped to constant memory by using the `__constant` keyword;

- **Local Memory** – This memory region is local to a work-group, as such, accesses may have much shorter latency and much high bandwidth than global memory. Local memory can be used to allocate variables that are shared by all work-items in that work-group. This memory region may be implemented as dedicated regions of memory on the OpenCL device, or alternatively, the local memory region may be mapped onto sections of the global memory. The keyword to specify this region is `__local`;

- **Private Memory** – This memory region is private to a work-item. Variables defined in one work-item's private memory are not visible to other work-items. Local variables and nonpointer kernel arguments are private by default.

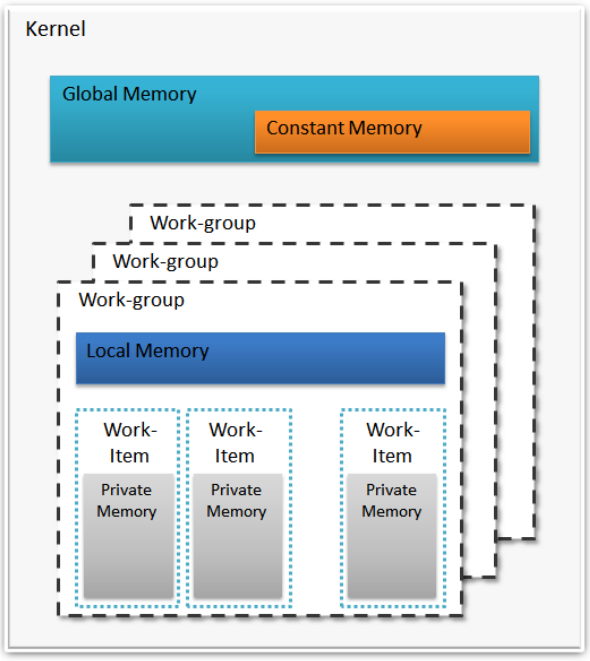

Figure B.3: Memory Model representation.

Programming Model

Specifies how the concurrency model is mapped to the physical hardware: processing elements, memory regions, and the host. Despite the primary model, that drives the design of OpenCL, being the *data parallel* programming model, the OpenCL execution model supports also *task parallel* programming model, and, also, a hybrid of these two models: tasks that contain data parallelism.

B.1.3 Framework

The OpenCL framework enables OpenCL applications to use a host and one or more devices as a single heterogeneous parallel computer system. The framework is divided into the following components:

- **OpenCL Platform Layer:** defines functions used by the host program to find OpenCL devices and their respective capabilities, as well as to create the context for the OpenCL application.
- **OpenCL Runtime:** allows the host program to manipulate contexts to create command-queues and other operations that occur at runtime, for example, the functions that submit commands to the command queue come from this component.
- **OpenCL Compiler:** it creates program executables that contain OpenCL kernels. The compiler supports a subset of the ISO C99 language with extensions for parallelism.

B.2 OpenCL API Supported Data Types

In the following table B.1, are presented, amongst others (i.e. built-in vector data types, reserved data types, other built-in data types), some of OpenCL supported built-in scalar

data types and the syntax alterations made by the API.

Table B.1: Built-in Scalar Data Types.

OpenCL Type	**API Type**	**Description**
bool	–	true (1) or false (0)
char	cl_char	8-bit signed
unsigned char, uchar	cl_uchar	8-bit unsigned
short	cl_short	16-bit signed
unsigned short, ushort	cl_ushort	16-bit unsigned
int	cl_int	32-bit signed
unsigned int, uint	cl_uint	32-bit unsigned
long	cl_long	64-bit signed
unsigned long, ulong	cl_ulong	64-bit unsigned
float	cl_float	32-bit float
half	cl_half	16-bit float (for storage only)
size_t	–	32- or 64-bit unsigned integer
ptrdiff_t	–	32- or 64-bit signed integer
intptr_t	–	signed integer
uintptr_t	–	unsigned integer
void	–	void

Appendix C

Scientific Contributions

C.1 Published Paper

Abstract. Clusters that combine heterogeneous compute device architectures, coupled with novel programming models, have created a true alternative to traditional (homogeneous) cluster computing, allowing to leverage the performance of parallel applications. In this paper we introduce clOpenCL, a platform that supports the simple deployment and efficient running of OpenCL-based parallel applications that may span several cluster nodes, expanding the original single-node OpenCL model. clOpenCL is deployed through user level services, thus allowing OpenCL applications from different users to share the same cluster nodes and their compute devices. Data exchanges between distributed clOpenCL components rely on Open-MX, a high-performance communication library. We also present extensive experimental data and key conditions that must be addressed when exploiting clOpenCL with real applications.

The complete paper can be consulted at `http://pm.bsc.es/heteropar12/papers/30_dOpenCL.pdf`.