Abstract

Heterogeneous computing combines general purpose CPUs with accelerators to efficiently execute both: sequential control-intensive and data parallel phases of applications. Existing programming models for heterogeneous computing rely on programmers to explicitly manage the different memories in the system and manage data transfers between the CPU system memory and accelerator memory.

To facilitate the programming of accelerator-based systems, has created a library called GMAC. This model basically abstracts the developer from the peculiarities of each particular system and proposes a simple API to replace the host code for devices management for a more generic one. Programming with GMAC, the programmer does not need to manage data transfers between different devices. This action is done internally by GMAC.

In this project we improve GMAC with new capabilities and also improve already existing functions. First, we have developed a utility that divide the shared data space into similar blocks in length. Secondly, we have developed a memory coherence protocol to maintain all shared data coherent along all the devices. Also, we have checked that all the functions, already implemented, work with this new system. And finally, we performed a set of tests to check that every function in the system work properly.
**Resum**

La computació heterogènia combina les CPUs de propòsit general amb acceleradors per executar de manera eficient tant: la part seqüencial per al control-intensiu com les fases paral·leles de les dades de les aplicacions. Els Models de programació existents per a la computació heterogènia depenen dels programadors per gestionar de forma explícita les diferents memòries que hi ha al sistema i gestionar les transferències de dades entre la memòria del sistema de la CPU i de la memòria de l’accelerador.

Per facilitar la programació de sistemes basades en acceleradors, s’ha creat una llibreria anomenada GMAC. Aquest model bàsicament abstreu el desenvolupador de les peculiaritats de cada sistema en particular i proposa una API senzilla per reemplaçar el codi del host per a la gestió de dispositius per un de més genèric. Programar amb GMAC, el programador no té cap necessitat per gestionar les transferències de dades entre els diferents dispositius. Aquesta acció es porta a terme internament per GMAC.

En aquest projecte millorrem GMAC amb noves capacitats i millorar les funcions ja existents. En primer lloc, hem desenvolupat una utilitat que divideix l’espai de dades compartit en blocs similars en longitud. En segon lloc, hem desenvolupat un protocol de coherència de memòria per mantenir totes les dades compartides coherents al llarg de tots els dispositius. A més, hem comprovat que totes les funcions, ja implementades, funcionen amb aquest nou sistema. I, finalment, es va realitzar una sèrie de proves per comprovar que totes les funcions en el sistema funcionen correctament.
Resumen

La computación heterogénea combina las CPUs de propósito general con aceleradores para ejecutar de manera eficiente tanto: la parte secuencial para el control-intensivo como las fases paralelas de los datos de las aplicaciones. Los Modelos de programación existentes para la computación heterogénea dependen de los programadores para gestionar de forma explícita las diferentes memorias que ha en el sistema i gestionar las transferencias de datos entre la memoria del sistema de la CPU y de la memoria del acelerador.

Para facilitar la programación de sistemas basadas en aceleradores, se ha creado una librería llamada GMAC. Este modelo básicamente abstrae al desarrollador de las peculiaridades de cada sistema en particular y propone una API sencilla para reemplazar el código del host para la gestión de dispositivos por uno más genérico. Programar con GMAC, el programador no tiene ninguna necesidad para gestionar las transferencias de datos entre los diferentes dispositivos. Esta acción se lleva a cabo internamente por GMAC.

En este proyecto mejoramos GMAC con nuevas capacidades y mejorar las funciones ya existentes. En primer lugar, hemos desarrollado una utilidad que divide el espacio de datos compartido en bloques similares en longitud. En segundo lugar, hemos desarrollado un protocolo de coherencia de memoria para mantener todos los datos compartidos coherentes a lo largo de todos los dispositivos. Además, hemos comprobado que todas las funciones, ya implementadas, funcionan con este nuevo sistema. Y, por último, se realizó una serie de pruebas para comprobar que todas las funciones en el sistema funcionan correctamente.
Acknowledgements

This project has benefited greatly from the support of many people, some of whom I would sincerely to thank.

I would like to thank my supervisor Javier Cabezas for his helpful suggestions, his help, corrections and constructive feedback during all the phases of this project. Without his help, this project wouldn’t have finished.

I would also like to thank my co-supervisors Agustín Fernández and Nacho Navarro for their guidance in this project and discussions on the topic.

In third place, I would like to thank my friends and colleagues at the Barcelona School of Informatics. Thanks to all of you who have always been my side during this experience.

Finally, but first in my heart, I would like to thank my parents. They are due my deep gratitude for their continued moral support and encouragement throughout my studies.
Contents

Abstract ii

Resum iii

Resumen iv

Acknowledgements v

Table of Contents vi

1 Introduction 1
   1.1 Introduction .................................................. 1
   1.2 Project Overview ............................................. 2
   1.3 Goals ........................................................... 2
   1.4 Technical Competencies ....................................... 2

2 Analysis 4
   2.1 State of the Art ............................................... 4
     2.1.1 Technology overview ..................................... 4
     2.1.2 Related work .............................................. 6
   2.2 Stakeholders .................................................. 6
   2.3 Risks ........................................................... 7

3 Heterogeneous Computing 8
   3.1 Introduction .................................................. 8
     3.1.1 GPU Architecture ......................................... 10
     3.1.2 CPU-GPU Connection ....................................... 11
     3.1.3 Parallel Programming Languages ......................... 12
   3.2 CUDA ............................................................ 13
     3.2.1 Introduction ............................................... 13
     3.2.2 Data Parallelism Model ................................... 13
     3.2.3 CUDA Program Structure ................................. 14
     3.2.4 Device Memories and Data Transfer ...................... 16
     3.2.5 Kernel Functions .......................................... 19
     3.2.6 CUDA Threads .............................................. 22
     3.2.7 CUDA vs. OpenCL .......................................... 22

4 GMAC 24
   4.1 Introduction .................................................. 24
   4.2 Overall Design ................................................ 25
   4.3 Memory Model .................................................. 25
   4.4 Execution Model ............................................... 26
# Contents

4.5 GMAC code example ........................................ 26

5 Design and Implementation .................................... 29
  5.1 Shared Address Space ........................................ 29
  5.2 Memory Coherence Protocol .................................. 32
  5.3 Improvements ............................................... 34

6 Testing .......................................................... 35
  6.1 Test Environment ............................................ 35
  6.2 Tests .......................................................... 35
    6.2.1 Unit Tests ................................................ 37

7 Management ....................................................... 40
  7.1 Planning ....................................................... 40
    7.1.1 Task Description ......................................... 40
    7.1.2 Gantt Diagram ............................................. 41
  7.2 Budget ......................................................... 42
    7.2.1 Human Resources .......................................... 43
    7.2.2 Material Resources ....................................... 44
    7.2.3 Total Cost ................................................ 44
    7.2.4 Viability ................................................... 45
  7.3 Methodology .................................................. 45

7.4 Laws and Regulations ......................................... 46

7.5 Sustainability and Social Impact ............................. 47
  7.5.1 Social impact .............................................. 47
  7.5.2 Environment impact ........................................ 47
  7.5.3 Economic impact ......................................... 48

8 Conclusions ....................................................... 49

9 Future Work ...................................................... 51

Glossary ............................................................ 52

Bibliography .......................................................... 54

A GMAC API ......................................................... 56
  A.1 GMAC API ...................................................... 56
  A.2 DSM manager interface ...................................... 60

B Matrix Multiplication ............................................... 65
  B.1 C++ Source Code .............................................. 65
  B.2 CUDA Source Code .......................................... 67
  B.3 GMAC Source Code .......................................... 71

List of Figures ...................................................... 74

List of Tables ....................................................... 75

List of Code Samples ................................................ 76
Chapter 1

Introduction

1.1 Introduction

Traditionally, the use of GPUs (Graphics Processing Units) was limited to computing graphics or image processing. In recent years, nevertheless, has begun to use, such processors, for computing applications where traditionally general-purpose CPUs (Central Processing Units) are used. Currently, though graphics processors are designed primarily for generation of 3D graphics, fulfil certain characteristics that make them very attractive for computing.

Heterogeneous computing combines general purpose CPUs with accelerators to efficiently execute both: sequential control-intensive and data parallel phases of applications. Nowadays, heterogeneous system architecture utilize multiple processor types usually on the same silicon die, to give the best of both worlds: GPUs processing, apart from its well known 3D graphics rendering capabilities, can also perform mathematically intensive computations on very large data sets, while CPUs can run the operating system and perform traditional serial tasks.

There are various existing programming models for heterogeneous computing. The two most important are: CUDA (Compute Unified Device Architecture), owned by Nvidia and therefore is only executable on Nvidia GPUs; and for the other hand, OpenCL (Open Computing Language), which is an open standard and developed by a group called khronos.

All of the existing programming models rely on programmers to explicitly manage the different memories in the system and manage data transfers between the CPU system memory and accelerator memory.

To facilitate the programming of the GPUs, it has created a library called GMAC (Global Memory ACcelerators). GMAC is a user-level library that provides a programming model for that issue. This model basically abstracts the developer from the peculiarities of each particular system and proposes a simple API (Application Programming Interface) to replace the host code for devices management for a more generic one. The programmer does not need to manage any memory or data transfers between different devices. This action is done internally by the library.

In this project our work consists to improve GMAC with new capabilities and also improve already existing functions. First, we have developed a utility that divide the shared data
space into similar blocks. All these blocks has the same length and they are fixed. Secondly, we have developed a memory coherence protocol to maintain all shared data coherent along all the devices address space. Also, we have checked that all the functions, already implemented, work with this new system. And finally, we performed a set of tests to check that every function in the system work properly.

1.2 Project Overview

The project is part of a project to develop a library that serves as a run-time, called GMAC to simplify the programmability of applications in heterogeneous architectures. One of the GMAC goals is to abstract the programmer of the type of API that hardware accelerator uses, trying to simplify the two major programming platforms of accelerators, CUDA and OpenCL.

The project will consist in the development of the necessary support for memory management. First off, we will implement the shared address space between the accelerator (GPU) and CPU. And lastly, we will implement the memory coherence protocol. In Chapter 5, we explain most deeply what these terms mean.

1.3 Goals

The main goal of this project is to develop of the necessary support for memory management. Therefore, the concrete goals for the project are as follows:

- The first objective is a personal goal and consists to acquire new knowledge about heterogeneous systems and how to deal with them.
- Provide GMAC library with new functionalities.
- Improve the library that allow users, engineers, and other stuff to use and program their applications with it.

1.4 Technical Competencies

Bellow follows a commentary on the particular competencies that were listed at the start of the project:

- **CEC2.1**: Analyse, evaluate, select and configure hardware platforms for the development and execution of computer applications and services. [Competent]

  This competence has been used during the first phases of the project, with the analysis of the existing architectures and the whole software they use to work properly.

- **CEC2.2**: Program taking into account the hardware architecture, using assembly language as well as high-level programming languages. [In depth]
1 Introduction

All programming has been made, in C++ language, thinking about how GPUs and CPUs can interact and share information with each other. For that reason, we had to know how memory works.

diamond CEC2.3: Develop and analyse software for systems based on microprocessors and its interfaces with users and other devices. [In depth]

All work has been done considering the hardware Accelerators. we have also made tests to evaluate the work done. With this work we improve the GMAC library with improvements to make it easy for the programmers or users develop their application.

diamond CEC2.4: To design and implement system and communications software. [Competent]

The core of this project is to develop of the necessary support for information sharing between CPUs and GPUs. Thus, we developed our system to deal with this.
Analysis

In this chapter we will discuss the analysis that has been done of the project.

2.1 State of the Art

In order to start with the project, we must analyse the state of the art. First, we will talk about existing technologies, architectures and products related to out project. And finally, we will talk about previous work done in the field.

2.1.1 Technology overview

Heterogeneous computing combines general-purpose CPUs with accelerators to efficiently execute both sequential control intensive and data parallel phases of applications. Existing programming models for heterogeneous computing rely on programmers to explicitly manage data transfers between the CPU system memory and accelerator memory.

Maximizing multi thread throughput and minimizing single-thread latency are two design goals that impose very different and often conflicting requirements on processor design. For example, the Intel Xeon E7 family [9] processors consist of six processor cores each of which is an high frequency out-of-order, multi instruction issue processor with a sophisticated branch prediction mechanism to achieve short single thread execution latency. This is in contras to the NVIDIA Tesla GT200 GPU [12] design that achieves high multi thread throughput with many cores, each of which is a moderate frequency, multi threaded, in-order processor that shares its control unit and instruction cache with seven other cores. For control intensive code, the Intel Xeon design can easily outperform the NVIDIA Tesla. For massively data parallel applications, the NVIDIA Tesla design can easily achieve higher performance than the Intel Xeon.

Data parallel code has the property that multiple instances of the code can be executed concurrently on different data. Data parallelism exists in many applications such as weather prediction, financial analysis, medical imaging and physics simulation. Most of these applications also have control intensive phases that are often interleaved between data parallel phases. Hence, general-purpose CPUs and accelerators can be combined to form heterogeneous parallel computing systems that efficiently execute all application phases [19].
General-purpose CPUs and accelerators can be coupled in many different ways. Fine-grained accelerators are usually attached as functional units inside the processor pipeline. In the Cell BE chip, the synergistic processing units, L2 cache controller, the memory interface controller, and the bus interface controller are connected through an Element Interconnect Bus [10]. The Intel Graphics Media Accelerator is integrated inside the graphics and memory controller hub that manages the flow of information between the processor, the system memory interface, the graphics interface, and the I/O controller [8]. AMD Fusion chips will integrate CPU, memory controller and GPU into a single chip. A common characteristic among Cell BE, Intel Graphics Media Accelerator, and AMD Fusion is that general-purpose GPUs and accelerators share access to system memory. In these systems, the system memory controller deals with memory requests coming from both general-purpose CPUs and accelerators.

Accelerators and general-purpose CPUs impose very different requirements on the system memory controller. General-purpose CPUs are designed to minimize the instruction latency and typically implement some form of strong memory consistency. Accelerators are designed to maximize data throughput and implement weak forms of memory consistency. Memory controllers for general-purpose CPUs tend to implement narrow memory buses (e.g. 192 bits for the Intel Core i7) compared to data parallel accelerators (e.g. 512 bits for the NVIDIA GTX280) to minimize the memory access time. Relaxed consistency models implemented by accelerators allow memory controllers to serve several requests in a single memory access. Strong consistency models required by general-purpose CPUs do not offer the same freedom to rearrange accesses to system memory. Memory access scheduling in the memory controller has different requirements for general-purpose CPUs and accelerators (i.e., latency vs throughput).

Programming models for current heterogeneous parallel systems, such as CUDA [3] and OpenCL [25], present different memories in the system as distinct memory spaces to the programmer. Applications explicitly request memory from a given memory space and perform data transfers between different memory spaces.

Such programming models ensure that data structures reside in the memory of the processor (CPU or accelerator), that performs subsequent computations. These models also imply that programmers must explicitly request memory on different processors and, thus, a data structure is referenced by two different memory addresses, a virtual address in system memory and a physical address in the accelerator memory. Programmers must explicitly manage memory coherence before executing kernels on the accelerator. This approach also prevents parameters from being passed by reference to accelerator kernels [5] and computationally critical methods to return pointers to the output data structures instead of returning the whole output data structure, which would save bandwidth whenever the code at CPU only requires accessing a small portion of the returned data structure. These approaches harm portability because they expose data transfer details of the underlying hardware. Offering a programming interface that requires a single allocation call and removes the need for explicit data transfers would increase programmability and portability of heterogeneous systems.
2 Analysis

2.1.2 Related work

Most programming models proposed for massively parallel systems deal with data distribution and kernel scheduling on clusters of computers. Global Arrays [15] provide semantics to divide and access arrays on a distributed memory system. In a data-centric programming model, the accelerator memory hosts all data required by accelerator kernels and, therefore, no data distribution is required if only one accelerator is used for each kernel execution. Global Arrays are compatible with a data centric programming model and might be used if the execution of a kernel is distributed among several accelerators. ASSIST [27] decomposes programs into components that communicate through data streams. ASSIST requires the programmer to declare modules and connect them using streams. This data dependence information is used by the ASSIST run-time to schedule the execution of modules on different processors. A data centric programming model also requires the programmer to assign data structures to computational intensive kernels.

Software development kits for commercially available accelerators such as the Cell Runtime Management Library [7], NVIDIA CUDA [17][16] or OpenCL [18], require programmers to explicitly move data between system memory and accelerator memory prior to performing any calculation using these data structures on the accelerator. The library GMAC removes the need for explicit data transfers, thus easing application development, and uses CUDA or OpenCL to interact with GPU accelerators. OmpSs [4][1] is a programming model where programmers identify tasks and their input and output parameters through source code annotations. The OmpSs run-time exploits task-level parallelism by executing independent tasks concurrently. OmpSs differs from the data centric programming model in that OmpSs identifies input and output parameters whose value is only known at the method call time, instead of data structures. Hence, the OmpSs does not allow data to be eagerly transferred to or from accelerators.

2.2 Stakeholders

There are several actors interested in the development of this project. Specifically, we have detected three main actors that directly involved with the project:

- **Author**

  The author of this project is the most important because its aim is to finish the job in time established planning to finish his degree. This will motivate him to invest the necessary time to carry out the objectives.

- **Supervisor and co-supervisors**

  The supervisor and co-supervisors will have a library after the project reaches its end, which can develop a GPU based program more easily, without taking into account either data transfer or sharing data, nor the consistency between all the memories the system.
2 Analysis

- **Potential customers**

  We can distinguish two types of prospects: those who want to use our library in their applications and those who want to use it in research groups to develop other methods or other libraries. Customers who could use the library might be physicists, biologists, mathematicians, or any other discipline that are not necessarily computer engineers that must know how to deal with memory management of the various devices involved in the system. Also, might be any type of customer who want to develop applications in CUDA or OpenCL, and still does not want to move data from one place to another and making copies of data.

2.3 Risks

During the development of this project we must take into account the risks involved in the process in order to detect their appearance as soon as possible. Some of the most evident risks are as follows:

- **Many different graphics cards.** Currently on the market there are a variety of graphics cards and each one is different from another. To support all of them is a bit difficult. Given that almost every 6 months there is a new graphics card on the market.

- **Competition.** The leading companies in the field (Nvidia, AMD), who are manufacturing graphics cards, or any other institution can find a quick and easy way to carry out this purpose, even this project has not been finished yet.

- **Time.** The project will not be ready in time if any deviation occurs or arises any difficulty. To avoid this, we have made regular meetings during the course of the project.
Heterogeneous Computing

3.1 Introduction

Microprocessors based on a single CPU, such as those in the Intel Pentium family and the AMD Opteron family, drove rapid performance increases and cost reductions in computer applications for more than two decades. These microprocessors brought GFLOPS (Giga FLoating-point Operations Per Second) to the desktop and hundreds of GFLOPS to cluster servers. This relentless drive of performance improvement has allowed application software to provide more functionality and generate more useful results. The users, in turn, demand even more improvements once they become accustomed to these improvements, creating a positive cycle for the computer industry.

During the drive, most software developers have relied on the advances in hardware to increase the speed of their applications under the hood: the same software simply runs faster as each new generation of processors is introduced. This drive, however, has slowed since few years ago due to energy consumption and heat dissipation issues that have limited the increase of the clock frequency and the level of productive activities that can be performed in each clock period within a single CPU. Virtually all microprocessor vendors have switched to models where multiple processing units, referred to as processor cores, are used in each chip to increase the processing power.

Traditionally, the vast majority of software applications are written as sequential programs. The execution of these programs can be understood by a human sequentially stepping through the code. A sequential program will only run on one of the processor cores, which will not become significantly faster than those in use today.

The semiconductor industry has settled on two main trajectories for designing microprocessor. The multi-core trajectory seeks to maintain the execution speed of sequential programs while moving into multiple cores. The multi-cores began as two-core processors, with the number of cores approximately doubling with each semiconductor process generation (e.g. Intel Core i7 microprocessor). In contrast, the many-core trajectory focuses more on the execution throughput of parallel applications. The many-cores began as a large number of much smaller cores, and, once again, the number of cores doubles with each generation (e.g. NVIDIA GeForce GTX280). Many-core processors, especially the GPUs, have led the race of floating-point performance since few years ago. This phenomenon is illustrated in the Figure 3.1. While the performance improvement of the general-purpose microprocessors (CPUs) has slowed significantly, the GPUs have continued to improve
3 Heterogeneous Computing

relentlessly.

Traditionally, the use of GPUs was limited to computing graphics or image processing [11]. In recent years, nevertheless, has begun to use, such processors, for computing applications where traditionally general-purpose CPUs are used. Currently, though graphics processors are designed primarily for generation of 3D graphics, fulfill certain characteristics that make them very attractive for computing, for example in science or simulation. Between these features, we can highlight a high level of parallelism and optimization for floating point calculations. The Figure 3.2 illustrate the results of a benchmark made between GPUs and CPUs that shows how GPU performance is greater than CPU performance while increasing the size of the problem. With a higher performance value, indicating that a problem of a certain size will be solved more quickly.

Heterogeneous computing refers to systems that use more than one kind of processor. These are multi-core systems that gain performance not just by adding cores, but also by incorporating specialized processing capabilities to handle particular tasks. Heterogeneous System Architecture utilize multiple processor types (typically CPUs and GPUs), usually on the same silicon die, to give the best of both worlds: GPU processing, apart from its well known 3D graphics rendering capabilities, can also perform mathematically intensive computations on very large data sets, while CPUs can run the operating system and perform traditional serial tasks.

The drive to improve performance and the continuing constraints on power and scalability in multi-core CPU development have led semiconductor, software and systems designers increasingly to look to the vector processing capabilities of GPUs. Vector processors like those in advanced GPUs have up to thousands of individual compute cores, which can operate simultaneously. This makes GPUs ideally suited for computing tasks that deal with a combination of very large data sets and intensive numerical computation.
3 Heterogeneous Computing

Figure 3.2: Performance benchmark between GPUs and CPUs

3.1.1 GPU Architecture

One might ask why there is such a large performance gap between GPUs and general-purpose CPUs. The answer lies in the differences design philosophies between the two type of processors, as illustrated in Figure 3.3. The design of the CPU is designed for the sequential code performance. It makes use of sophisticated control logic to allow instructions from a single thread of execution to execute in parallel or even out of their sequential order while maintaining the appearance of sequential execution. More importantly, large cache memories are provided to reduce the instruction and data latencies of large complex applications. Neither control logic nor cache memories contribute to the peak calculation speed.

In contrast, the design philosophy of the GPUs is shaped by the fast growing video game industry, which exerts tremendous economic pressure for the ability to perform a massively number of floating-point calculations per video frame in games. This demand, motivates the GPU vendors to look for ways to maximize the chip area and power budget dedicate to floating-point calculations. The prevailing solution, to date, is to optimize the execution throughput of massive number of threads. The hardware takes advantage of a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses, thus minimizing the control logic required for each execution thread. Small cache memories are provided to help control the bandwidth requirements of these applications that multiple threads that access the same memory data do not need to all go to the main memory. As a result, much more chip area is dedicated to the floating-point calculations.

Memory bandwidth is another important issue. Graphics chips (GPUs) have been op-
3 Heterogeneous Computing

Figure 3.3: CPUs and GPUs: different design philosophy

Operating approximately 10 times the bandwidth of available CPU chips. This is because of frame buffer requirements and the relaxed memory model (the way various system software, applications, and I/O devices expect their memory accesses to work). General-purpose processors have to satisfy requirements from operating system, applications, I/O devices that make memory bandwidth more difficult to increase. In contrast, with simpler memory models, the GPU designers can easily achieve more higher memory bandwidth.

3.1.2 CPU-GPU Connection

The GPU executes independently from the CPU but is controlled by the CPU. Application programs running on the CPU use graphics API, runtime, and driver software components to communicate with the GPU. Most of the communication involves placing commands or data in memory buffers and transmitting them to the GPU. Graphical data that are accessed frequently (Such as vertices, textures, output images) by the GPU are often placed in a high-bandwidth memory attached directly to the GPU, with the CPU being used to set the initial state of these objects. Even with the dedicated GPU memory, the CPU sends a great deal of data to the GPU on behalf of the application. Modern PCs use the PCIe bus [23] to connect the CPU and the GPU. PCIe is a scalable bus, divided into serial, bidirectional lanes as illustrated in Figure 3.4.

Entertainment consoles or other dedicated devices use their own interconnect strategy. In the case of consoles, this may offer considerably higher bandwidth than is available through PCIe. In some implementations, the GPU may be integrated into the memory controller chip and CPU may share the same memory rather than using dedicated memory for the GPU. These integrated GPUs are a popular low-cost alternative to add-in cards. These options provide interesting cost and performance tradeoffs and also affect some of the processing strategies used by the application developer. However, the basic GPU architecture concepts remain unaffected.
3.1.3 Parallel Programming Languages

As GPUs designed for running graphics applications, most of the development for programming languages and APIs has been targeted at writing graphics applications. This makes non-graphics programming more challenging as the programmer must deal with idioms from the graphics APIs and languages, such as drawing triangles to create a set of domain points and trigger fragment processing across that domain. Shading programs must be written to process the domain points, using texture mapping operations to read data associated with each domain point and writing the computed result as a color value.

To simplify this programming task and hide the underlying graphics idioms, several programming languages and routines have been created. These range from systems from graphics vendors that expose low-level details of the underlying graphics hardware implementation (CUDA [3], OpenCL [25]) to research and commercial higher-level languages and systems intended to simplify development of data parallel programs. The ones that are the most widely used are MPI (Message Passing Interface) [14] and OpenMP (Open Multi-Processing) [2]. MPI for scalable cluster computing, is a model where computing nodes in a cluster do not share memory. All data sharing and interaction must be done through explicit message passing. MPI has been successful in the high-performance scientific computing domain. Applications written in MPI have been known to run successfully on cluster computing systems with more than 100,000 nodes. The amount of effort required to port an application into MPI, however, can be extremely high due to lack of shared memory across computing nodes. OpenMP for shared-memory multiprocessor systems, supports shared memory, so it offers the same advantage as other graphic language. However, it has not been able to scale beyond a couple hundred computing nodes due to thread management overheads and cache coherence hardware requirements.

Aspects of CUDA are similar to both MPI and OpenMP in that the programmer manages the parallel code constructs, although OpenMP compiler do more of the automation in managing parallel execution. Several ongoing research efforts aim to adding more automation of parallelism management and performance optimization to the CUDA tool chain. OpenCL is similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery. OpenCL is a standardized programming model in that applications developed in it can run...
3 Heterogeneous Computing

without modification on all processors that support the language extensions and API.

In many cases, the language combines the parts of the code that execute on the CPU and the parts that execute on the GPU in a single program. This differs from many of the graphics APIs (such as OpenGL (Open Graphics Library) [22]) that deliberately make the boundary between the CPU and GPU explicit.

One advantage of higher level languages is that they preserve high-level information that can potentially be used by the underlying runtime to manage execution and memory coherence. In contrast, lower level systems leave that largely up to the programmer, requiring the programmer to learn various architectural details to approach peak efficiencies. Low-level systems may allow programmers to achieve better performance at the cost of portability, but they also may allow access to newer, more efficient processing constructs that are not currently available in the graphics APIs.

3.2 CUDA

3.2.1 Introduction

CUDA is a set of tools (created by nVIDIA) allowing to encode programs for nVIDIA GPUs.

To a CUDA programmer, the computing system consists of a host, which is a traditional CPU, such as an Intel architecture microprocessor in personal computers today, and one or more devices, which are massively parallel processors equipped with a large number of arithmetic execution units. In modern software applications, program sections often exhibit a rich amount of data parallelism, a property allowing many arithmetic operations to be safely performed on program data structures in a simultaneous manner. The CUDA devices accelerate the execution of these applications by picking a large amount of data parallelism.

3.2.2 Data Parallelism Model

Data parallelism is a form of parallelization of computing across multiple processors in parallel computing environments. Data parallelism focuses on distributing the data across different parallel computing nodes. It contrasts to task parallelism as another form of parallelism.

In a multiprocessor system executing a single set of instructions, data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code.

Data parallelism refers to the program property whereby many arithmetic operations can be safely performed on the data structures in a simultaneous manner.
To give an illustration the concept of data parallelism, we give a matrix-matrix multiplication example in Figure 3.5. In this example, each element of the product matrix $C$ is generated by performing a dot product between a row of input matrix $A$ and a column of input matrix $B$. The highlighted element of matrix $C$ is generated by taking the dot product of the highlighted row of matrix $A$ and the highlighted column of matrix $B$. Note that the dot product operations for computing different matrix $C$ elements can be simultaneously performed. That is, none of these dot products will affect the results of each other. Therefore, matrix multiplication of large dimensions can have very large amount of data parallelism. For example, a $1000 \times 1000$ matrix multiplication has $1,000,000$ independent dot product. By executing many dot products in parallel, a CUDA device can significantly accelerate the execution of the matrix multiplication over a traditional host CPU. Nevertheless, the data parallelism in real applications is not always as simple as that in the matrix multiplication example.

\[ C = A \cdot B \]

**Figure 3.5:** Matrix-matrix multiplication example

### 3.2.3 CUDA Program Structure

A CUDA program consists of one or more phases that are executed on either the host, CPU, or a device such as a GPU. The phases that show little data parallelism are implemented in host code. The phases that show rich amount of data parallelism are implemented in the device code. A CUDA program is a unified source code comprising both host and device code. The `nvcc` (NVIDIA C compiler) separates the two code type during the compilation process. The host code is straight ANSI C code: it is further compiled with the host’s standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labelling data-parallel functions, called *kernels*, and their associated data structures. The device code is typically further compiled by the `nvcc` and executed on a GPU device. In situations where no device is available or the kernel is more appropriately executed on a CPU, one can also
choose to execute kernels on a CPU using the emulation features in CUDA SDK (Software Development Kit) or the MCUDA tool [24].

The kernel functions typically generate a large number of threads to exploit data parallelism. In a problem with a lot of data, the number of threads that will be created will be very large. Would generate more than 1,000,000 threads when it is invoked. It is worth noting that CUDA threads are of much lighter weight than the CPU threads. CUDA programmers can assume that these threads take very few cycles to generate and schedule due to efficient hardware support. This is in contrast with the CPU threads that typically require thousands of clock cycles to generate and schedule. For example, in the matrix-matrix multiplication, the entire matrix multiplication computation can be implemented as a kernel where each thread is used to compute one element of output matrix. In this example, the number of threads used by the kernel is a function of the matrix dimension. For a 1000 x 1000 matrix-matrix multiplication, the kernel that uses one thread to compute one output element would generate 1,000,000 threads.

The Figure 3.6 illustrate the execution of a typical CUDA program. The execution starts with host (CPU) execution. When a kernel function is invoked, the execution is moved to a device (CUDA), where a large number of threads are generated to take advantage of abundant data parallelism. All the threads that are generated by a kernel during an invocation are collectively called a grid (array of threads). When all threads of a kernel complete their execution, the corresponding grid terminates, and the execution continues on the host until another kernel is launched.

It is worthwhile to introduce a code example that concretely illustrates the CUDA program structure. Listing 3.1 shows a simple main function skeleton. The main program first allocates the variables, which will work, in the host memory and then performs I/O to read the values. Then, it allocates memory for each variable in the device and copy data from the host memory to the device memory. At this point, the program is able to perform kernel launch (invoke kernel functions to do calculations in the GPU). Once the kernel execution is finished (after completing the calculations in the device), the main function read the data from the device memory to the host memory and finally, free up all the allocated data.
int main()
{
    // Part 1:
    // Allocate and initialize x, y
    // Perform I/O to read the input values x, y

    // Part 2:
    // Allocate device memory for d_x, d_y
    // Copy data from host to device

    // Part 3:
    // Launch the kernel to perform the calculation in the device

    // Part 4:
    // Copy data from device to host

    // Part 5:
    // Free data, x, y, d_x, d_y

    return 0;
}

Listing 3.1: A simple CUDA program structure

As we can see, part 3 performs a kernel launch. We will explain how to do this operation as well as we will give a full example in the Section 3.2.5 (both the host code and the device code).

3.2.4 Device Memories and Data Transfer

In CUDA, the host and devices have separate memory spaces. This reflects the reality that devices are typically hardware cards that come with their own memory GDRAM (Graphical Dynamic Random Access Memory). In order to execute a kernel on a device, the programmer needs to allocate memory on the device and transfer pertinent data from the host memory to the allocated device memory. Similarly, after device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed. The CUDA runtime system provides API functions to perform these activities on behalf of the programmer.

Figure 3.7 shows an overview of the CUDA device memory model for programmers to reason about the allocation, movement, and usage of the various memory types of a device. At the bottom of the figure, we see: global memory, constant memory and texture memory. These are the memories that the host code can transfer data to and from the device, as illustrated by the bidirectional arrows between these memories and the host. Registers, shared memory and local memory are on-chip memories. Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner.

- Global memory

  Global memory resides in device memory, usually is the larger memory available
3 Heterogeneous Computing

Figure 3.7: CUDA device memory model

in the device. When a warp\(^1\) executes an instruction that accesses global memory board requests from around the warp in one or many transactions depending on the size of the memory accesses. Access time of the memory is high, so we need a very high number of accesses to mask latency. Is a read-write memory.

- **Constant memory**

  Constant memory supports short-latency, high-bandwidth, read-only access by the device code when all threads simultaneously access the same location. This is where constants and kernel arguments are stored.

- **Texture memory**

  This memory advantage memory used in the graphics pipeline for computational uses. Specifically, texture memory is cache optimized for 2D spatial access pattern.

- **Local memory**

  Does not physically exist. It is an abstraction to the local scope of a thread. Actually put in global memory by the compiler. Used for whatever does not fit into registers.

- **Shared memory**

\(^1\)A warp in CUDA is a group of 32 threads, which is the minimum size of the data processed in SIMD by a CUDA multiprocessor
Shared memory resides in the chip, therefore has a higher bandwidth and lower latency than the local and global memories. To achieve a higher bandwidth, the memory is divided into banks. Any request for reading/writing of \( n \) addresses that fall in \( n \) different banks can still be served simultaneously. As a result, we have that the optimal bandwidth is \( n \) times the bandwidth of a single bank. But if you have two requests to the same bank, is a conflict that must be serialized.

Shared memory is allocated to thread blocks. All threads in a block can access variables in the shared memory locations allocated to the block. Shared memory is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work.

**Registers**

Registers are allocated to individual threads. Each thread can only access its own registers. A kernel function typically uses registers to hold frequently accessed variables that are private to each thread.

By declaring a CUDA variable in one of the CUDA memory types, the programmer dictates the visibility and access speed of the variable. *Table 3.1* presents the CUDA syntax for declaring program variables into the various types of device memory. Each such declaration also gives its declared CUDA variable a scope and lifetime.

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic variables other than arrays</td>
<td>Register</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td>Automatic array variables</td>
<td>Local</td>
<td>Thread</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int var</td>
<td>Shared</td>
<td>Block</td>
<td>Kernel</td>
</tr>
<tr>
<td><strong>device</strong> int var</td>
<td>Global</td>
<td>Grid</td>
<td>Application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int var</td>
<td>Constant</td>
<td>Grid</td>
<td>Application</td>
</tr>
</tbody>
</table>

*Table 3.1: CUDA variable qualifiers type*

Scope identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of all grids. If the scope of a variable is a single thread, a private version of the variable will be created for every thread. Each thread can only access its private version of the variable. For example, if a kernel declares a variable whose scope is a thread and it is launched with 1 million threads, then 1 million versions of the variable will be created so each thread initializes and uses its own version of the variable.

Lifetime specifies the portion of the program’s execution duration when the variable is available for use: either within a kernel’s invocation or throughout the entire application. If a variable’s lifetime is within a kernel invocation, it must be declared within the kernel function body and will be available for use only by the kernel’s code. If the kernel is invoked several times, the contents of the variable are not maintained across these invocations. Each invocation must initialize the variable in order to use them. On the other side, if a variable’s lifetime is throughout the entire application, it must be declared outside of any
function body. The contents of the variable are maintained throughout the execution of the application and are available to all kernels.

### 3.2.5 Kernel Functions

In CUDA, a kernel function specifies the code to be executed by all threads during a parallel phase. Because all of these threads execute the same code, CUDA programming is an instance of the well-known single-program, multiple-data (SPMD) parallel programming style, a popular programming style for massively parallel computing systems.

Listing 3.2 shows a simple kernel function. The syntax is ANSI C with some notable extensions. First, there is a CUDA specific keyword `__global__` in front of the declaration of the function name. This keyword indicates that the function is a kernel and that it can be called from a host functions to generate a grid of threads on a device.

```c
__global__ void vecadd ( int N, float *x, float *y, float *c)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[idx] = x[idx] + y[idx];
}
```

**Listing 3.2:** A simple CUDA kernel function

In general, CUDA extends C function declarations with three qualifier keywords. The meanings of these keywords are summarized in Table 3.2. The `__global__` keyword indicates that the function being declared is a CUDA kernel function. The function will be executed on the device and can only be called from the host to generate a grid of threads on a device. We will show the host code syntax for calling a kernel function later. Moreover, there are two other keywords that can be used in front of a function declaration.

<table>
<thead>
<tr>
<th>Declaration</th>
<th>Executed on</th>
<th>Only callable from</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float devFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void kernel()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float hosFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

**Table 3.2:** CUDA extensions to C functional declaration.

Following with the table, The `__device__` keyword indicates that the function being declared is a CUDA device function. A device function executes on a CUDA device and can only be called from a kernel function or another device function. Device functions can have neither recursive function calls nor indirect function calls through pointers in them. The `__host__` keyword indicates that the function being declared is a CUDA host function. A host function is simply a traditional C function that executes on the host and can only be called from another host function. By default, all functions in a CUDA program are host functions if they do not have any of the CUDA keywords in their declaration.

When a kernel is invoked, it is executed as grid of parallel threads. In Figure xx, the launch of kernel 1 creates grid 0 (As shown in Figure 3.8). Each CUDA thread grid
3 Heterogeneous Computing

typically is comprised of thousands to millions of GPU threads per kernel invocation. Creating enough threads to fully utilize the hardware often requires a large amount of data parallelism. Threads in a grid are organized into a two-level hierarchy. At the top level, each grid consists of one or more thread blocks. All blocks in a grid have the same number of threads. Each block has a unique two-dimensional coordinate given by the CUDA specific keywords `blockIdx.x` and `blockIdx.y`. All thread blocks must have the same number of threads organized in the same way.

![CUDA thread organization](image)

**Figure 3.8:** CUDA thread organization

Each thread block is, in turn, organized as a three-dimensional array of threads with a total size of up to 512 threads. The coordinates of threads in a block are uniquely defined by three thread indices given by the CUDA specific keywords: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`.

When the host code invokes a kernel, it sets the grid and thread block dimensions via execution configuration parameters. This is illustrated in Listing 3.3: cuda-kernel-invoke. To call the kernel function, we must specify two parameters. The first, `dimBlock`, is for describing the configuration of blocks. The second, `dimGrid`, describes the configuration of the grid. And finally, the final line of code invokes the kernel. The special syntax between the name of the kernel function and the traditional C parameters of the function surrounded by `<<<` and `>>>` is a CUDA extension to ANSI C. It provides the dimensions of the grid in terms of number of blocks and the dimensions of the blocks in terms of number of threads.

```c
unsigned int nw = (N + 255) / 256;

// Configuration
dim3 dimBlock(256, 256, 1);
dim3 dimGrid(nw, nw, 1);

// Invoke kernel
vecadd<<<dimGrid, dimBlock>>>(N, d_x, d_y, d_c);
```

**Listing 3.3:** A CUDA kernel invocation
Heterogeneous Computing

A complete example of code is shown in Listing 3.4. This piece of code illustrates all the concepts we have been explained in this section.

```c
__global__ void vecadd(int N, float *x, float *y, float *c)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[idx] = x[idx] + y[idx];
}

int main()
{
    // Part 1: Allocate x, y, c
    unsigned int length = N * sizeof(float);
    float* h_x = (float*) malloc(length);
    float* h_y = (float*) malloc(length);
    float* h_c = (float*) malloc(length);

    // Part 2: Allocate device memory for d_x, d_y, d_c
    float* d_x, d_y;
    cudaMalloc((void**)&d_x, length);
    cudaMalloc((void**)&d_y, length);
    cudaMalloc((void**)&d_c, length);

    // Part 3: initialize x, y
    read(x); read(y);

    // Part 4: Transfer data to device
    cudaMemcpy(d_x, h_x, length, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, h_y, length, cudaMemcpyHostToDevice);

    // Part 5: Launch kernel to do calculation in the device
    // Configuration
    unsigned int nw = (N + 255) / 256;
    dim3 dimBlock(256, 256, 1);
    dim3 dimGrid(nw, nw, 1);

    // Invoke kernel
    vecadd<<<dimGrid, dimBlock>>>(N, d_x, d_y, d_c);

    // Part 6: Transfer data to host
    cudaMemcpy(h_c, d_c, length, cudaMemcpyDeviceToHost);

    // Part 7: Use results
    useful(h_c);

    // Part 8: Free data, x, y, c host
    free(h_x); free(h_y); free(h_c);

    // Part 9: Free data d_x, d_y, d_c device
    cudaFree(d_x);
    cudaFree(d_y);
    cudaFree(d_c);
}
```
3 Heterogeneous Computing

```c
    cudaFree(d_x); cudaFree(d_y); cudaFree(d_c);
    return 0;
}
```

Listing 3.4: A complete CUDA application

### 3.2.6 CUDA Threads

As seen so far, CUDA threads are organized hierarchically to facilitate the programming of applications that automatically scale depending on the number of processors or cores. The basic unit of this hierarchy is the thread, which as explained, that just running the kernel source coordinating with other threads. Also seen how these threads have a unique identifier to distinguish themselves from each other and to identify the appropriate portion of the data to process. These threads are organized into a two-level hierarchy using unique coordinates: `blockIdx` (for block index) and `threadIdx` (for thread index), assigned to them by the CUDA runtime system.

When a thread executes the kernel function, references to the `blockIdx` and `threadIdx` variables return the coordinates of that thread. Additional built-in variables, `gridDim` and `blockDim`, provide the dimension of the grid and the dimension of each block respectively.

In general, a grid is organized as a 2D array of blocks. Each block is organized into a 3D array of threads. The exact organization of a grid is determined by the execution configuration provided at kernel launch.

CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function, `__syncthreads()`. When a kernel function calls `__syncthreads()`, the thread that executes the function call will be held at the calling location until every thread in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase. Barrier synchronization is a simple and popular method of coordinating parallel activities.

Once a grid is launched, its blocks are assigned to streaming multiprocessors in arbitrary order, resulting in transparent scalability of CUDA applications. The transparent scalability comes with the limitation that threads in different blocks cannot synchronize with each other. The only safe way for threads in different blocks to synchronize with each other is to terminate the kernel and start a new kernel for the activities after the synchronization point.

### 3.2.7 CUDA vs. OpenCL

OpenCL is a standardized, cross-platform, parallel-computing API based on the C language. It is designed to enable the development of portable parallel applications for systems with heterogeneous computing devices.

OpenCL has a more complex platform and device management model than CUDA that
Heterogeneous Computing

reflects its support for multi-platform and multi-vendor portability. Whereas the OpenCL standard is designed to support code portability across devices produced by different vendors, such portability does not come free. OpenCL programs must be prepared to deal with much greater hardware diversity and thus will exhibit more complexity. Also, many OpenCL features are optional and may not be supported on all devices, so a portable OpenCL code must avoid using these optional features. Some of these optional features, though, allow applications to achieve significantly more performance in devices that support them.

OpenCL employs a data parallelism model that has direct correspondence with the CUDA data parallelism model. An OpenCL program consists of two parts: kernels that execute on one or more OpenCL devices and a host program that manages the execution of kernels. Figure 3.9 summarizes the mapping of OpenCL data parallelism concepts to their CUDA equivalents.

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Device (CPU, GPU, etc)</td>
</tr>
<tr>
<td>Multiprocessor</td>
<td>Compute Unit or CU</td>
</tr>
<tr>
<td>Scalar or CUDA core</td>
<td>Processing Element or PE</td>
</tr>
<tr>
<td>Global or Device Memory</td>
<td>Global Memory</td>
</tr>
<tr>
<td>Shared Memory (per block)</td>
<td>Local Memory (per workgroup)</td>
</tr>
<tr>
<td>Local Memory (registers)</td>
<td>Private Memory</td>
</tr>
<tr>
<td>Thread Block</td>
<td>WorkGroup</td>
</tr>
<tr>
<td>Thread</td>
<td>WorkItem</td>
</tr>
<tr>
<td>Warp</td>
<td>-</td>
</tr>
<tr>
<td>Grid</td>
<td>NDRange</td>
</tr>
</tbody>
</table>

Figure 3.9: Mapping between OpenCL and CUDA concepts

Like CUDA, the way to submit work for parallel execution in OpenCL is for the host program to launch kernel functions. OpenCL kernels have identical basic structure as CUDA kernels. All OpenCL kernel function declarations start with a _kernel keyword, which is equivalent to the _global keyword in CUDA.
4.1 Introduction

Programming models like CUDA and OpenCL present different memories to the programmer, which is responsible for memory arrangement in a given space and perform explicit data transfers between these different spaces. These programming models suppose that data structures that reside in the memory are closest to the computation unit (either the processor of the CPU or the accelerator) to perform computations more efficiently. Programmers are, furthermore, responsible for managing the consistency of memory. This approach prevents kernels has parameters passed by reference and instead of returning a pointer (to save bandwidth) have to return all the resulting structure.

Therefore, providing the programmer an interface that requires an unique allocation (an unique address space) and eliminate the need for explicit transfer, would increase the programmability and portability of applications for heterogeneous systems. Bearing that in mind, was born GMAC library.

GMAC is a user-level library that provides a programming model that attempts to facilitate the programming of accelerator-based systems, such as systems with GPUs. This model basically abstracts the developer from the peculiarities of each particular system and proposes a simple API to replace the host code for devices management for a more generic one. This allows, for example, the execution of CUDA or OpenCL code on the same host with the same code (no need to change the code).

Some important features that the model give us are:

- Unification of the various system memories (asymmetric distributed shared memory [6]).
- Compatibility with CUDA and OpenCL.
- Information about system topology (PCIe interconnection, sockets number, memory access bandwidth, etc.).
- System hardware abstraction and its peculiarities.
- Transparently exploits the system hardware capacities.
Increment the portability between different systems.

### 4.2 Overall Design

**GMAC** is composed of several layers that give each one a certain level of abstraction of the system. Most important that are shown in [Figure 4.1](#) are:

- **HAL** (Hardware Abstraction layer). Implements a set of operations to obtain, store and execute kernel in heterogeneous architectures. Thanks to this layer, the programmer has a hardware abstraction layer.

- **DSM** (Distributed Shared Memory). Implements a memory manager coherence between different address spaces with operations Acquire/Release. It abstracts us for making explicit data copies between different address spaces.

- **ULAS** (Unified Logical Address Space). Provides a single logical address space for all the memories.

- **System-level programming model**. This can be any programming language.

- **Backend**. Such as **CUDA** or **OpenCL**. The backends are a layer that is under HAL, which implement the abstraction layer.

### 4.3 Memory Model

**GMAC** memory model is based on the use of multiple virtual address spaces to define which objects can be accessed from devices (as illustrated in [Figure 4.2](#)). Given that each device has its own physical address space (local memory), it allows the creation of virtual address space for each device. In addition, **GMAC** allows the programmer to create data objects that can be mapped in one or multiple existent virtual address spaces. Thus, the programmer can easily define which objects are visible to each device. All objects can be
accessed by any of the system devices and GMAC transparently handles remote access or data replication.

![GMAC memory model](image)

**Figure 4.2:** GMAC memory model

When an object is mapped in a virtual address space, you get a view. This view allows you to define certain properties about data visibility, such as the ability to read or write, or the behaviour of the coherence protocol.

Needless to say that GMAC also provides optimizations for memory access in a transparent manner and the programmer does not have to worry about, for example, to make transferences to the device using ”Pinned” memory or exploit certain characteristics of a particular GPU model.

### 4.4 Execution Model

GMAC allows the creation of different execution context for each virtual device. These contexts can be created from a virtual address space, allowing, creating multiple independent execution contexts. This allows you to run different kernels on one GPU independently and therefore its execution can be parallelized.

### 4.5 GMAC code example

Before giving an example, in Figure 4.3 the two models (CUDA and GMAC) are shown and what steps to follow to create an application.

If we compare them, CUDA has to allocate memory for both the CPU and GPU. It also has to do data transfers between the CPU and GPU and vice versa when the job finished. At the end of the program, you need to free the memory for both the CPU as the GPU. In contrast, GMAC only needs one memory allocation. No need to do memory transfers, manually by the developer, but GMAC is responsible internally in a timely manner. And at the end of the program, you just need to free the memory that has been reserved. As noted in the figure, the kernel is the same for both CUDA and GMAC and will not change. Thus, programming is simplified significantly using GMAC.
It is worthwhile to introduce a code example that concretely illustrate a GMAC program structure. Listing 4.1 show an easy GMAC code to illustrate what we explained in this chapter.

As we can see in the code, there is no need to perform memory copy action between the host and the device manually. Also we don’t need to create a duplicated references to a variables, with only one reference to a variable is enough.

With only call (*gmacMalloc()*), GMAC is capable to handle all these actions internally; perform memory copy when needed, create only one pointer.

```c
int main()
{
    // Part 1: Allocate x, y, c
    unsigned int length = N * sizeof(float);
    float* x, y, c;
    gmacMalloc((void **)&x, length);
    gmacMalloc((void **)&y, length);
    gmacMalloc((void **)&c, length);

    // Part 2: initialize x, y
    read(x); read(y);

    // Part 3: Launch kernel to do calculation in the device
    // Configuration
    unsigned int nw = (N + 255) / 256;
    dim3 dimBlock(256, 256, 1);
    dim3 dimGrid(nw, nw, 1);

    // Invoke kernel
    vecadd<<<dimGrid, dimBlock>>>(N, x, y, c);

    // Part 4: Use results
    useful(c);

    // Part 5: Free data, x, y, c
    gmacFree(x); gmacFree(y); gmacFree(c);
}
```
Comparing the code shown above with the one shown in the last chapter (Listing 3.4), we can see that the code written in GMAC is much simpler than code written in CUDA. In the above code, the relevant parts that it save are: memory allocation (If memory is reserved on both devices, duplicate pointers are created) and data transfer to and from the device.
Design and Implementation

After defining what improvements should be added to the library, has begun implementing them.

First of all we have seen that we need to implement a memory coherence protocol (explained in Section 5.2). But before implement this protocol, we have to implement the virtual space that devices share (explained in Section 5.1).

5.1 Shared Address Space

GMAC builds a shared address space between the CPUs and GPUs. When an application requests memory (via \texttt{gmacMalloc()}), accelerator memory is allocated on the accelerator, returning a memory address that can be used only by the accelerator. Then, the library request the operating system to allocate system memory over the same range of virtual memory addresses. To carry out this, is done by an operating system call, which accepts a virtual address and maps it to an allocated range of system memory. At this point, two identical memory addresses ranges have been allocated, one in the GPU memory and the other in the CPU memory. Hence, a single pointer can be returned to the application to be used by both code (CPU and GPU). Therefore, with a simple memory request call (\texttt{gmacMalloc()}), GMAC can give to the programmer a single reference to the memory instead of two references.

Until now, the memory range created is an unique block. Internally it’s treated as an unique block in which all the operations are performed in it. This can cause problems when dealing with memory coherence (explained in the next section). For example, when a device writes in the start of the block, is necessary to invalidate the whole range in other devices and send them, if necessary, the new data. Even if another device only reads and writes in a part at the end of the block. This phenomenon is called false sharing. Two or more devices share the same memory range, but no other part overlap each other. But the protocols when managing the memory consistency, the whole range is treated as if it was just one.

The solution to this problem, is to divide the range in blocks where every block has the same length. We have decided that the block length should be the page size of the system memory. By doing this, we had to review the entire DSM layer, adapt all the existing functions to the new block style as well as create new functions to perform this change.
We illustrate an example in Figure 5.1. As can be seen, in this example, we have three devices named ptr1, ptr2, ptr3 (these devices can be either CPUs or GPUs). Each device has its own memory (the blue range for ptr1, green range for ptr2 and orange range for ptr3) and then when two devices want to create a shared space, the responsible function, creates a mapping between these two memory spaces (in the picture looks purple). The created mapping, is divided into parts of equal size and fixed length, called blocks. It may happen that a third device wants to create a shared space with another device that is already mapped with another. Then, a second mapping is created with these two devices. These blocks can be created in different mappings. In our example, we created two mappings, the m1: between ptr1 and ptr2; and m2: between ptr2 and ptr3. Therefore, the b1 and b4 block are only in m1 mapping, while blocks b2 and b3 are in m1 and m2 mapping.

To keep all this information, we had to create data structures. We have created a structure in the mapping, to know what blocks has the mapping and the information associated with them. We have also created a structure on the block to know what mappings are assigned to it.

Within the DSM layer, functions that are responsible for creating the mapping between the different memory spaces and to decouple these spaces are: link and unlink.

- Function link. This function receives the following parameters, creates a mapping between the spaces of the two devices and returns a code corresponding to the operation result. This result can be classified in two categories: correct or incorrect. But within the result of incorrect, there are several codes depending on the error. The error might be one of those shown in Listing 5.3.

In Listing 5.1, we show the prototype for the link function. Each parameter is explained in the code.

```c
error link(
    hal::ptr dst, // Pointer of the first device
    hal::ptr src, // Pointer of the second device
) {
    // Implementation...
}
```
5 Design and Implementation

```cpp
size_t count, // The length of the memory range that we want
    // reserve. From this length blocks are created
GmacProtection protDst, // What type of protection will have
    // the memory range for the first device. Can be either Read or write
GmacProtection protSrc, // What type of protection will have
    // the memory range for the second device. Can be either Read or write
int flags = mapping_flags::MAP_DEFAULT // In this release are
    // not used
)
```

Listing 5.1: Prototype of link function

- **Function unlink.** This function receives the following parameters, undoes the mapping between two devices returns a code corresponding to the operation result. This result can be classified in two categories: correct or incorrect. But within the result of incorrect, there are several codes depending on the error. As mentioned earlier, the error might be one of those shown in Listing 5.3.

In Listing 5.2, we show the prototype for the `unlink` function. Each parameter is explained in the code.

```cpp
error unlink(
    hal::ptr mapping, // Pointer of the device which we want to
          // undoes the mapping
    size_t count // The length of the memory range that we want to
         // undoes
)
```

Listing 5.2: Prototype of unlink function

As mentioned, this class, has all the error type that we handle in our layer. If an error code doesn’t exist in this class, simple add it to it.

```cpp
#ifndef GMAC_DSM_ERROR_H_
#define GMAC_DSM_ERROR_H_
namespace __impl { namespace dsm {
enum class error {
    DSM_SUCCESS = 2000,
    DSM_ERROR_INVALID_ALIGNMENT = 2001,
    DSM_ERROR_INVALID_PTR = 2002,
    DSM_ERROR_INVALID_VALUE = 2003,
    DSM_ERROR_INVALID_PROT = 2004,
    DSM_ERROR_ownership = 2005,
    DSM_ERROR_PROTOCOL = 2006,
    DSM_ERROR_HAL = 2999
};
}
#endif
```

Listing 5.3: Error code
5.2 Memory Coherence Protocol

To keep all the system information consistent and consistent with all devices on the system, the library must have a memory coherence protocol. We have chosen the MSI protocol because the purpose for which we want, it fits our needs perfectly.

The GMAC coherence protocol is defined from the CPU perspective. All booking and data transfers are managed by the CPU. The GPU (or accelerator) don’t perform any memory consistency or coherence actions. In our system, this protocol works at a block level (as mentioned in the last section).

MSI stands for: Modified-Shared-Invalid. The protocol maintains the following invariant: each block of memory is always in exactly one of the following states:

- **I (Invalid)**. Means that the block is only in GPU (or accelerator) memory and must be transferred back if the CPU reads this block after the accelerator kernel returns.

- **M (Modified)**. Means that the CPU has an updated copy of the block and this block must be transferred back to the accelerator when the accelerator kernel is called.

- **S (Shared)**. Means that both the CPU and the GPU have the same version of the data so the block does not need to be transferred before the next method invocation on the accelerator.

To maintain this invariant the MSI protocol forces state transitions as dictated by the following state machine (Figure 5.2), which shows the state of the memory block with respect to a single device. All edges are labelled with the activity that causes the transition. Any value after the / represents an action place on the bus.

![Figure 5.2: MSI protocol](image)

The local device is capable of performing the following actions:
5 Design and Implementation

- **R (Read).** Attempting to read the data in the block.
- **W (Write).** Attempting to write data in the block.

These actions can result in the following bus actions, which in turn can cause state transitions in other devices.

- **trans.** Provide a copy of the block to another device. Because your copy is more recent than the the copy in it memory.
- **r_acquiereR.** Request a copy of the block to another devices.
- **B.** Broadcast, send a request or provide an information to all the devices
- **inv.** Send a request to some device to invalidate it block.

Within the DSM layer, functions that are responsible for managing the consistency are: acquire and release.

- **Function acquire.** This function receives the following parameters, protects and lock the memory blocks for the device that called and acts according to the MSI protocol.

  In Listing 5.4, we show the prototype for the **acquire** function. Each parameter is explained in the code.

```c
error acquire(
    hal::ptr mapping , // Address of the device that calls the function. It is the address that is mapped to some other device
    size_t count , // The length of the memory range that we want bring
    GmacProtection prot // What protection will have the block. Can be either Read or write
)
```

**Listing 5.4:** Prototype of acquire function

- **Function release.** This function receives the following parameters and leave the block. Unlock it so another device can lock it.

  In Listing 5.5, we show the prototype for the **release** function. Each parameter is explained in the code.

```c
error release(
    hal::ptr mapping , // Address of the device that calls the function. It is the address that is mapped to some other device
    size_t count , // The length of the memory range that we want unlock
)
```

**Listing 5.5:** Prototype of release function
5 Design and Implementation

5.3 Improvements

The improvement we have made, are related to the memory coherence protocol.

Improve upon original MSI protocol by detecting CPU read and write access to objects in invalid state. These accesses are detected using the CPU hardware memory protection mechanisms (accessible using the `mprotect()` system call) to trigger a page fault exception, which causes a page fault handler to be executed. The code inside the page fault handler implements the state transition diagram shown in Figure 5.3.

![Figure 5.3: MSI protocol improvement](image)

Memory protection hardware is configured to trigger a page fault on any access (read or write) to shared data structures in invalid state. Whenever a data structure in invalid state is accessed by the CPU, the object is transferred from accelerator memory to system memory, and the data structure state is updated to shared, on a read access, or to modified on a write access.

On a kernel invocation, all shared data structures are invalidated and those in the modified state are transferred from system memory to accelerator memory. On kernel return no data transfer is done and all shared data objects remain in invalid state. This approach produces important performance gains with respect to original protocol in applications where the code executed on the accelerator is part of an iterative computation and the code executed on the CPU after the accelerator invocation only updates some of the data structures used or produced by the code executed on the accelerator.
Testing

Testing has been done exhaustively in this project. Unit testing has been done for every new function we made and for every other functions, already, in the library to be certain that the library still works.

6.1 Test Environment

To perform both work and tests, it has been used the Asterix server from BSC (Barcelona Supercomputer Centre). This server is described in Table 6.1.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>4x Nvidia Tesla C2070</td>
</tr>
<tr>
<td>CPU</td>
<td>2x Intel Xeon E5620</td>
</tr>
<tr>
<td>Memory</td>
<td>24GB DDR3/1066</td>
</tr>
</tbody>
</table>

Table 6.1: Test equipment hardware

6.2 Tests

To carry out with the tests, it has been used GoolgeTest. GoogleTest is a framework to write C++ unit testing based on the xUnit architecture [13]. It supports automatic test discovery, a rich set of assertions, user-defined assertions and more features. The library allow unit testing of C sources as well as C++ with minimal source modification. GoogleTest is a debugger for computer programs and code because at the first sign of a failure, the debugger is automatically invoked [21].

xUnit is the collective name for several unit testing frameworks. All xUnit frameworks share the following basic component architecture, with some varied implementation details.

- **Test runner.** Is an executable program that runs tests implemented using an xUnit framework and reports the test results.

- **Test case.** Is the most elemental class. All unit tests are inherited from here.
6 Testing

- **Test fixtures.** Also known as a test context is the set of preconditions or state needed to run a test. The developer should set up a known good state before the tests, and return to the original state after the tests.

- **Test suites.** Is a set of tests that all share the same fixture.

- **Test execution.** Is the execution of an individual unit test.

- **Test result formatter.** Produces results in one or more output formats.

- **Assertions.** Is a function or macro that verifies the behaviour (or the state) of the unit under test.

In the **Listing 6.2**, we illustrate a simple example of GoogleTest.

Fist, lets us consider the prototype for a simple cubic function shown in **Listing 6.1**

```cpp
double cubic(const double);
```

**Listing 6.1:** Prototype of the cubic function

Next, **Listing 6.2** creates a test hierarchy named **CubicTest** and then adds two unit tests, **PositiveNs** and **ZeroAndNegativeNs**, to that hierarchy. **TEST()** arguments go from general to specific. The first argument is the name of the test case, and the second argument is the test’s name within the test case. Both names must be valid C++ identifiers. A test’s full name consists of its containing test case and its individual name. Tests from different test cases can have the same individual name.

```cpp
#include "gtest/gtest.h"

TEST(CubicTest, PositiveNs) {
  EXPECT_EQ (3375 , cubic (15) );
  EXPECT_EQ (2406.104 , cubic (13.4) );
  EXPECT_EQ (125 , cubic (5) );
  EXPECT_EQ (63 , cubic (4) ); // This will give an error
}

TEST(CubicTest, ZeroAndNegativeNs) {
  ASSERT_EQ (0.0 , cubic (0.0) );
  ASSERT_EQ (-27 , cubic (-3.0) );
}

int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

**Listing 6.2:** GoogleTest example

The `::testing::InitGoogleTest` method initializes the framework and must be called before `RUN_ALL_TESTS` and must be called only once in the code because multiple calls
to it conflict with some of the advanced features of the framework. \textit{RUN\_ALL\_TESTS} automatically detects and runs all the tests defined using the \textit{TEST} macro.

\section*{6.2.1 Unit Tests}

\textbf{From DSM layer.} In this layer, we test the next:

1. \textbf{Block creation}. Create a new block with a specific size.

2. \textbf{Mapping creation}. Create a new mapping.

3. \textbf{Prepend block}. Append blocks to mapping at the start of the mapping (Start of the structure).

4. \textbf{Append block}. Append blocks to mapping at the end of the mapping (End of the structure).

5. \textbf{Append Mapping}. Append mapping to another mapping. This action moves the blocks from one mapping to another mapping and clear up the first mapping.

6. \textbf{Split}. Split mapping into more ranges. This action will split the mapping making from one more than one.

7. \textbf{Resize}. Resize mapping. Might resize the mapping either from the start or from the end. Might make it larger or shorter as needed.

8. \textbf{Mappings in range}. get mapping in a specific range. Mappings when are created, they are inserted in a group manager. With this operation, we test that we can retrieve a specific set of mappings. We might get from the start of the group until some length or from some offset until some length.

9. \textbf{Insert blocks}. Check if we might insert blocks from the interface of the layer.

10. \textbf{Insert mappings}. Check if we might insert mappings from the interface of the layer.

11. \textbf{Merge mappings}. This operation will merge two or more mapping in one. The blocks from one mapping are transferred to another mapping and then the unnecessary mapping is deleted.

12. \textbf{Link}. Will test the creation of mapping between two different virtual spaces from different devices.

13. \textbf{Unlink}. Will test the operation of unlinking two different virtual spaces from different devices.

The output of all these tests is shown in Figure 6.1 and Figure 6.2.
Figure 6.1: Test output from DSM layer part 1

Figure 6.2: Test output from DSM layer part 2
6 Testing

From HAL layer. In this layer, we test the next:

1. **Acquire/Release**. Test MSI protocol.

2. **Memset**. This action, first perform the link between two devices (E.g. between CPU and GPU) and then initializes a region (E.g. a vector) in shared memory. So, when one device access to this region, will copy the data to its memory and if the other device, also access this region, will copy the data from one memory to another memory. Basically, with this operation we test the MSI protocol.

The output of all these tests is shown in Figure 6.3

![Figure 6.3: Test output from HAL layer](image-url)
Chapter 7

Management

In this section we will tackle with management for the project as a whole. We will introduce the final planning, the final budget, and we will briefly talk about the methodology followed, laws that affect the project and sustainability and social compromise.

7.1 Planning

In order to talk about planning we must first introduce the task involved in the project. Then, we can proceed to introduce the planning itself, from its initial version to its final version.

7.1.1 Task Description

Initially, we considered a set of phases that included all the tasks we then foresaw would be required to complete our project.

There are five main task involved in the project. Next, we will proceed to describe each one:

- **Previous Study**

  This first phase consists in the analysis of the work environment and become familiar with the tools with which to develop the project, both software and hardware. In this phase the resources are also tested, already implemented, which will be used during the project, such as open source libraries or reusable code.

- **Research**

  In this point we will do an analysis of the requirements to be achieved in the project and the best solutions are valued to meet the objectives set. Also, a part of this phase is to experiment with the library and acquire new knowledge.

- **Development**

  The goal of this phase is to develop the various functionalities that are part of the final library. The two functionalities are: shared address space and memory
coherence protocol.

- **Testing**

In this phase, that corresponds to the end of the methodology, will require verify that the entire system meets the project requirements. This work will be carried out with unit tests and later with tests together.

- **Documentation**

This phase consists in preparing the project report. In that document will include the context of the project, as has been its development and its result. more over, will include the viability studies and the planning. And lastly, will include some appendix with some real examples.

### 7.1.2 Gantt Diagram

**Initial planning**

Initially, we considered a set of phases that include all the tasks we then foresaw would be required to complete our project. This division into phases (or main tasks) included the system deployment as well as system testing and report writing (described above).

In *Figure 7.1* shows the Gantt diagram that was done at the beginning of the project. At this point we only considered the five main tasks of the project, as necessary to complete the *research* task to determine the different subtasks to be performed.

![Gantt diagram at the start of the project](image)
Final planning

The planning has suffered several changes since the initial planning, until it has reached its final state. This includes phase additions as well as date rearrangements, which we will now see.

First of, we added some new phases. All of them are in the development task. The first two, are the features to be developed during the project: address space memory and coherence memory protocol. And the last one, is all the improvements that will be made in the project.

Finally, we rearrange the date of some tasks. The first one is the development task. As we have now added two subtasks, the time for the entire task has been increased. Also because we have decided to include the improvement phase, which is expected to improve some system functionalities. And the last one is the testing task. This task has increased due to new tasks and improvements we included in the project.

In Figure 7.2 shows the Gantt diagram that contains all the changes mentioned above.

![Figure 7.2: Gantt diagram at the end of the project](image)

We consider our working day to be 4 hours long, since the author of this project will not be able to work full time on the project. With this idea, the Table 7.1 shows a comparison of hours between the initial and final planning:

7.2 Budget

Below are listed which have been the costs of the project. As we will see, the budget is divided in human resources and material resources, and then these two topics are added to generate the final cost. We also present a brief viability analysis.
### 7.2.1 Human Resources

The main cost in this project is given by the human resources. We can distinguish between three types of roles: project manager, developer and technician. In order to calculate the human resources costs, we need an estimated salary for these roles. Table 7.2 shows a summary of the estimated salaries for the different roles we have in our project, taken approximately from the Official Catalan College of Informatics Engineers\(^1\) [20].

<table>
<thead>
<tr>
<th>Resource</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project manager</td>
<td>50 €/h</td>
</tr>
<tr>
<td>Developer</td>
<td>45 €/h</td>
</tr>
<tr>
<td>Technician</td>
<td>25 €/h</td>
</tr>
</tbody>
</table>

Table 7.2: Estimated cost for each role type of human resources

With this data and with planning data, now we can estimate the cost of the task previously described in the project planning. Table 7.3 shows a summary for the human resources.

As we can see, there has been a slight increase in the project budget, which was initially estimated at 20,000 €, with 520 days of work. This is due to the extra days added to the task *development* and *testing*, which have increased the total amount of working hours dedicated to the project.

\(^1\)Col·legi Oficial d’Enginyeria Informàtica de Catalunya. [http://enginyeriainformatica.cat](http://enginyeriainformatica.cat)
### Table 7.3: Total cost for human resources related to each task work

<table>
<thead>
<tr>
<th>Task</th>
<th>Resource type</th>
<th>Duration(h)</th>
<th>Cost(€)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Previous Study</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ Developer</td>
<td>Developer</td>
<td>20</td>
<td>900</td>
</tr>
<tr>
<td>Mercurial</td>
<td>Developer</td>
<td>20</td>
<td>900</td>
</tr>
<tr>
<td>Software/Hardware</td>
<td>Developer</td>
<td>20</td>
<td>900</td>
</tr>
<tr>
<td><strong>Research</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project analysis</td>
<td>Manager</td>
<td>40</td>
<td>2,000</td>
</tr>
<tr>
<td>Research</td>
<td>Developer</td>
<td>80</td>
<td>3,600</td>
</tr>
<tr>
<td><strong>Development</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared address space</td>
<td>Developer</td>
<td>80</td>
<td>3,600</td>
</tr>
<tr>
<td>Memory coherence protocol</td>
<td>Developer</td>
<td>70</td>
<td>3,150</td>
</tr>
<tr>
<td>Improvements</td>
<td>Developer</td>
<td>45</td>
<td>2,025</td>
</tr>
<tr>
<td><strong>Testing</strong></td>
<td>Technician</td>
<td>120</td>
<td>3,000</td>
</tr>
<tr>
<td><strong>Documentation</strong></td>
<td>Technician</td>
<td>108</td>
<td>2,700</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td></td>
<td>603</td>
<td>22,775</td>
</tr>
</tbody>
</table>

#### 7.2.2 Material Resources

The development of this project also entails a set of material resources detailed below. In order to compute the material costs, has determined the total cost of the equipment used in the project and from the time of amortization of each equipment (See Table 7.4).

The equipment amortization was computed as follows: we must take into account the official amortization coefficient stated by the Spanish government, currently fixed to 26% per year [26]. We must take into account a whole year equivalent to the length or this project, with a standard day’s work of 8 hours. Since our working days consist of 4 hours, we must compute the amortization for 75.4 days (603h / 8h) in a year.

In this project we have 2 different equipments: a personal equipment and a test equipment (Described in Chapter 6). Therefore, the material cost remains as follows:

- Personal equipment: $800€ \times 0.26 \times 75.4/365 = 42.97€$
- Test equipment: $10,843€ \times 0.26 \times 75.4/365 = 582.37€$

In reference to software, there is no cost. We have selected development tools that are free of charge and open source.

#### 7.2.3 Total Cost

Bearing in mind the previous calculations, we can see that the total project cost will be

$$22,775 + 625.34 = 23,400.34 \quad €$$
As can be expected, the increase in human resources costs is reflected in the total cost for the project, which has also slightly increased from the initial estimated 20,539 €. The Table 7.5 shows a summary for the whole project.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Cost (£)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>22,775</td>
</tr>
<tr>
<td>Hardware</td>
<td>625.34</td>
</tr>
<tr>
<td>Software</td>
<td>0</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>23,400.34</strong></td>
</tr>
</tbody>
</table>

Table 7.5: Cost summary

### 7.2.4 Viability

The development of this project has been done within the context of a thesis, and therefore did not consider the search for funding resources. With this idea in mind, the project itself can be continued after this thesis by some institution who is interested in the library. In this case, they should consider the viability in the future. Nevertheless, the maintenance of the library will not require special resources that are not already available, and therefore will not represent an increase in economic resources.

### 7.3 Methodology

As agreed at the beginning of the project, we have followed an iterative evolution. The functionalities to be developed were split into several small changes. Each small change expanded with design and implementation and was accompanied by the necessary tests to validate the functionality developed. Figure 7.3 shows a methodology diagram for the project.

In order to ensure that the correct way was followed, we established periodic meetings with the supervisors of the project. More concretely, we met every week to comment work done and to discuss the improvements. This methodology worked, and allowed us to move forward while maintaining regular discussions on possible improvements to do.

About the code generated, has been synchronized with a version control repository. The
version control used is mercurial and the repository can be accessed online \(^2\) to facilitate code exploration, navigation and analysis.

Finally, we covered the platform code with unit tests. This ensured that the code was properly tested.

### 7.4 Laws and Regulations

The project has been developed thinking of it as a library, to help you carry out programming code and there are no external users involved. Therefore, the laws involved are: usage, modification and distribution that can be do with the code which we worked.

The laws that affect our project are:

- GMAC uses a BSD (Berkeley Software Distribution) 3-clause license, that allow to the users totally freedom to use, modify, share, distribute, and improve the library imposing minimal restrictions. The restrictions are: to retain the copyright notice and neither the names of the copyright holder nor names of its contributors may be used to endorse or promote products.

- The compiler GCC (GNU Compiler Compilation) has a GPL (General Public License) 3+ license. This license, gives a permission to copy, distribute and/or modify the software under the terms of the license.

- The control version Mercurial has a GPL v2+ license. Its designed to take away your freedom to so share and change it.

- The GoogleTest is released under the BSD 3-clause license. That allow to the users to use, modify, share, distribute and improve it as mentioned above.

That means that, once we finished the project, the library will still under the BSD 3-clause license, and can be shared and used by other users under open source label.

\(^2\)Bitbucket. \url{https://bitbucket.org/}

![Figure 7.3: Project methodology diagram](image)
7 Management

7.5 Sustainability and Social Impact

7.5.1 Social impact

This project has a goal to improve the programmability of the heterogeneous architecture with accelerators, such as, Nvidia devices, AMD devices or Intel Xeon Phi. If we take a look at the top 500 list (Picture 7.4), we can see that four of the top then supercomputers use some type of accelerator.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National Super Computer</td>
<td>Tianhe-2 (MilkyWay-2) - TH-FIV- FEI Cluster, Intel Xeon E5-2692 12C 2.50Ghz, TH Express-2, Intel Xeon Phi 31S1P NUDT</td>
<td>3,120,000</td>
<td>33,862.7</td>
<td>54,902.4</td>
<td>17,809</td>
</tr>
<tr>
<td>2</td>
<td>DOE/SCOE/Ascend Ridge</td>
<td>Titan - Cray XE6, Opteron 6274 16C 2.26Ghz, Cray Gemini Interconnect, Nvidia K20x</td>
<td>968,640</td>
<td>17,590.0</td>
<td>27,112.5</td>
<td>8,209</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NAS/Sequoia</td>
<td>Sequoia - BlueGeneQ, Power QIC 16C 1.60 Gz, Custom IBM</td>
<td>1,572,664</td>
<td>17,173.2</td>
<td>20,152.7</td>
<td>7,950</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science (AICS)</td>
<td>K computer, SPARC 8x 64x, Fujitsu Interconnect</td>
<td>705,024</td>
<td>10,510.0</td>
<td>11,200.4</td>
<td>12,660</td>
</tr>
<tr>
<td>5</td>
<td>DOE/SCOE/Argonne National</td>
<td>Mira - BlueGeneQ, Power QIC 16C 1.80Ghz, Custom IBM</td>
<td>786,432</td>
<td>6,586.6</td>
<td>10,066.3</td>
<td>3,945</td>
</tr>
<tr>
<td>6</td>
<td>Swiss National Supercomputer Centre (CSCS)</td>
<td>Piz Daint - Cray XC30, Xeon E5-2670 8C 2.50GHz, Arka Interconnect, Nvidia K20x</td>
<td>115,984</td>
<td>2671.0</td>
<td>7,718.9</td>
<td>2,325</td>
</tr>
<tr>
<td>7</td>
<td>Texas Advanced Computing</td>
<td>Stampede - PowerEdge C8220, Xeon E5-2680 8C 2.70GHz, Infiniband FDR, Intel Xeon Phi 714P</td>
<td>462,462</td>
<td>5,168.1</td>
<td>8,920.1</td>
<td>4,210</td>
</tr>
<tr>
<td>8</td>
<td>Forschungszentrum Juelich</td>
<td>JUEQUEEN - BlueGeneQ, Power QIC 16C 1.60GHz, Custom Interconnect</td>
<td>458,752</td>
<td>5,008.9</td>
<td>5,672.0</td>
<td>2,301</td>
</tr>
<tr>
<td>9</td>
<td>DOE/NAS/Sequoia</td>
<td>Vulcan - BlueGeneQ, Power QIC 16C 1.60GHz, Custom Interconnect IBM</td>
<td>393,216</td>
<td>4,293.3</td>
<td>5,033.3</td>
<td>1,972</td>
</tr>
<tr>
<td>10</td>
<td>Leibniz Recherchezentrum</td>
<td>SuperMUC - DataPlex DX360M4, Xeon E5-2660 8C 2.70GHz, Infiniband FDR IBM</td>
<td>147,456</td>
<td>2,897.0</td>
<td>3,183.1</td>
<td>3,423</td>
</tr>
</tbody>
</table>

Figure 7.4: Top 500 list (November 2013)

Therefore, we can say that an improvement in the programmability is not a wasted effort. For that reason, even though a small improvement is welcome. Also we can see that the trend in HPC (High-Performance Computing) is to use every time more and more accelerators to increase the machines performance. Moreover, the aim of this library is to reduce developer hours in the applications programming by letting to the library to deal with the hardware aspects so that the developer can focus in the development. These developers, sometimes, may not are computer scientist, but biologist, mathematician, etc. And therefore any kind of help is useful.

7.5.2 Environment impact

In computing, performance per watt is a measure of the efficiency of an architecture. It measures the speed of computation per watt of energy consumed.
GPUs have continued to increase the use of energy while the CPUs are designed thinking in performance per watt. But in the scientific field, GPUs may be more efficient than CPUs. This is because they have a great potential and a great parallelism which they can achieve.

If we take a look at the green 500 (Figure 7.5), we can see that the top ten most efficient supercomputers include an accelerator.

<table>
<thead>
<tr>
<th>Green500 Rank</th>
<th>MFLOPS/W</th>
<th>Site*</th>
<th>Computer*</th>
<th>Total Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4,503.17</td>
<td>GSIC Center, Tokyo Institute of Technology</td>
<td>TSUBAME-KFC - LX 1U:4GPU/164Re:1G Cluster, Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, NVIDIA K20x</td>
<td>27.78</td>
</tr>
<tr>
<td>2</td>
<td>3,631.86</td>
<td>Cambridge University</td>
<td>Wilkes - Dell T920 Cluster, Intel Xeon E5-2650v2 6C 2.600GHz, Infiniband FDR, NVIDIA K20</td>
<td>52.62</td>
</tr>
<tr>
<td>3</td>
<td>3,517.04</td>
<td>Center for Computational Sciences, University of Tsukuba</td>
<td>HA-PACS TCA - Cray X6-2700 V8 6C 2.660GHz, Infiniband QDR, NVIDIA K20</td>
<td>73.77</td>
</tr>
<tr>
<td>4</td>
<td>3,185.91</td>
<td>Swiss National Supercomputing Centre (CSCS)</td>
<td>Piz Daint - Cray XC30, Xeon E5-2670 6C 2.600GHz, Arlis Interconnect, NVIDIA K20x</td>
<td>1,753.66</td>
</tr>
<tr>
<td>5</td>
<td>3,130.95</td>
<td>ROMEO-HPC Center - Champagne-Ardenne</td>
<td>romee - Bull R421-E3 Cluster, Intel Xeon E5-2650v2 8C 2.600GHz, Infiniband FDR, NVIDIA K20x</td>
<td>81.41</td>
</tr>
<tr>
<td>6</td>
<td>3,068.71</td>
<td>GSIC Center, Tokyo Institute of Technology</td>
<td>TSUBAME 2.5 - Cluster Platform BL2900 G7, Xeon X5670 6C 2.930GHz, Infiniband QDR, NVIDIA K20x</td>
<td>922.54</td>
</tr>
<tr>
<td>7</td>
<td>2,702.16</td>
<td>University of Arizona</td>
<td>2xFlux, Xeon E5-2650v2 8C 2.600GHz, Infiniband, NVIDIA K20x</td>
<td>53.62</td>
</tr>
<tr>
<td>8</td>
<td>2,629.10</td>
<td>Max-Planck-Gesellschaft MPi/IP</td>
<td>2xFlux, Xeon E5-2650v2 10C 2.600GHz, Infiniband, NVIDIA K20x</td>
<td>269.94</td>
</tr>
<tr>
<td>9</td>
<td>2,629.10</td>
<td>Financial Institution</td>
<td>2xFlux, Xeon E5-2650v2 10C 2.600GHz, Infiniband, NVIDIA K20x</td>
<td>55.62</td>
</tr>
<tr>
<td>10</td>
<td>2,358.69</td>
<td>CSIRO</td>
<td>CSIRO GPU Cluster - Nitro G16 3GPU, Xeon E5-2650 8C 2.600GHz, Infiniband FDR, NVIDIA K20x</td>
<td>71.01</td>
</tr>
</tbody>
</table>

Figure 7.5: Green 500 list (November 2013)

Green 500 list is a good indicator of how the supercomputers will be in the future due to energy constraints are a limiting factor in the construction of the new machines. Therefore, as heterogeneous architecture with accelerators appears in the top position, indicates that in the future will be more abundant this type of architecture.

7.5.3 Economic impact

The improvement in the programmability can save time in applications development and facilitate the portability between platforms (CUDA and OpenCL).
Conclusions

If we compare the shape and the features it has scheduled to co-processors, such as GPUs, in a programming language such as CUDA and how to do so using a library as CUDA, we observe a great leap regarding the ease that CUDA brings to programmers.

First off, we have introduced the project topic and have explained the library in Chapter 1. Then, We have analysed the overview of the project and we have specified the functionalities to be included in the project. Then, we described the technical competencies that come into play in this project. Once done this, we continued with Chapter 2 where we have analysed the background of the project, potential risks that may occur during the project and finally, we have described several actors involved in our project. In Chapter 3 and Chapter 4, we have described the basis of our project and heterogeneous computing paradigms, as well as related programming languages and the CUDA library in question. Then we explained the development of our goals in Chapter 5, and rounded up the work done by describing testing in Chapter 6. In Chapter 7, we explained the general management of the project, talking about the planning and budget as well as changes made since the project began. We also discussed the methodology followed and the laws and regulations that affect our project. Finally, we have described the social impact and sustainability. Finally, we analysed the results of our work where we finished in Chapter 8.

A comprehensive list of glossary can be found in Chapter Glossary. And a list of references can be found Chapter Bibliography. Additional documents and examples can be found in Appendix A and Appendix B.

First, thanks to the shared memory range partitioning between different memory spaces of different devices. Now we can share the information at a block level and therefore, the performance will be increased from the previous version.

In the other hand, when implementing the protocol memory, all information shared will be consistent across all memory spaces of different devices.

We consider we have learnt very valuable lessons during the development of this project.

We have learnt to deal with a big project applying knowledge that has been acquired throughout the bachelor degree. Also, we have learnt about many new technologies related to the world of heterogeneous computing and programming GPUs without this project would not have been.
8 Conclusions

we have seen the whole software process working, from analysis to implementation. Along the bachelor degree, we see all the parts in the process, but we are not able to properly combine them into a single process until this project. Therefore, this knowledge may be applied in future projects.

Finally, we have developed a series of improvements that make the library be a little more best to help programmers to carry out their work.
Future Work

During the course of the project there have been various issues, or were beyond the scope of the project or posed a difficult effort to take in a reduced time.

Next, we show a list of possible extensions to be made.

**Asynchronous Transference Between CPU and GPU.** Creating a structure to store blocks in modified state in the host. Instead of going whenever making copies between host and device, now only when the size of this structure reaches a certain threshold, the asynchronous transfer is made. That is, copy all the blocks in that structure to the device.

**Thread safety.** Currently, some functions of the library are sequential. Should implement them using threading and make them thread safety. Thread safety is a concept applicable in the context of multi-thread programming. A piece of code is thread safety if it only manipulates shared data structures in a manner that guarantees safe execution by multiple threads at the same time.

**Study of the performance differences between GMAC and CUDA.** Conducting a comprehensive study to determine the performance of the GMAC code with respect to CUDA code.
**Glossary**

**API** Application Programming Interface. Specifies how some software components should interact with each other. 1, 2, 11–13, 16, 22, 24

**BSD** Berkeley Software Distribution. A family of permissive free software licenses. 46

**CPU** Central Processing Unit. 1–5, 8–11, 13–15, 26, 29, 30, 32, 34, 35, 39, 48, 51

**CUDA** Compute Unified Device Architecture. Is a parallel computing platform and programming model. 1, 2, 5–7, 12–16, 18–20, 22–26, 28, 48, 49, 51, 64

**GCC** GNU Compiler Compilation. Is a compiler system produced by the GNU Project supporting various programming languages. 46

**GDRAM** Graphical Dynamic Random Access Memory. 16

**GFLOPS** Giga (billion) FLoating-point Operations Per Second. Is a measure of computer performance. 8

**GMAC** Global Memory for ACcelerators. Is a user-level library that implements an Asymmetric Distributed Shared Memory model to be used by accelerator languages. 1–3, 6, 24–29, 32, 46, 51, 55, 64

**GPL** GNU General Public License. Is a free, copyleft license for software and other kinds of works. 46

**GPU** Graphics Processing Unit. 1–6, 8–15, 20, 24, 26, 29, 30, 32, 35, 39, 48, 49, 51

**Heterogeneous architecture** Refers to systems that use more than one kind of processor. It utilize multiple processor types (typically CPUs and GPUs). 1, 4, 22, 24, 47–49

**HPC** High-Performance Computing. 47

**Mercurial** Is a free, distributed source control management tool. 43, 44, 46

**MPI** Message Passing Interface. Is a message-passing library interface specification. 12

**NVCC** NVIDIA C compiler. 14
**Glossary**

**OpenCL**  Open Computing Language. Is a framework for writing programs that execute across heterogeneous platforms. 1, 2, 5–7, 12, 22–25, 48

**OpenGL**  Open GL. Is a cross-language API that supports multi-platform for rendering 2D and 3D vector graphics. 13

**OpenMP**  Open Multi-Processing. Is an API that supports multi-platform shared memory multiprocessing programming. 12

**PCIe**  PCI Express (Peripheral Component Interconnect Express). Is a high-speed serial computer expansion bus standard. 11, 24

**SDK**  Software Development Kit. Is typically a set of software development tools that allows for the creation of applications for a certain software package. 15
Bibliography


GMAC API

A.1 GMAC API

Bellow we show the GMAC user programming interface.

```c
/**
 * Returns the number of available accelerators.
 * \return The number of available accelerators
 */
GMAC_API unsigned APICALL gmacGetNumberOfDevices();

/**
 * Returns the ID of the accelerator the current thread is running on
 * \return The number of available accelerators
 */
GMAC_API unsigned APICALL gmacGetCurrentDeviceId();

/**
 * Fills the struct passed by reference with the information of the given
 * accelerator
 * \param acc Id of the accelerator to which request the information
 * \param info Pointer to the structure to be filled
 * \return gmacSuccess on success, an error code otherwise
 */
GMAC_API gmacError_t APICALL gmacGetDeviceInfo(unsigned acc, GmacDeviceInfo *info);

/**
 * Returns the amount of memory available in the given accelerator
 * \param acc The identifier of the accelerator to query
 * \param freeMem A pointer to the variable to store the amount of free
 * memory
 * \return gmacSuccess on success, an error code otherwise
 */
GMAC_API gmacError_t APICALL gmacGetFreeMemory(unsigned acc, size_t *freeMem);

/**
 * Migrates the GPU execution mode of a thread to a concrete accelerator.
 * Valid values are 0 * ... gmacNumberOfAccelerators() - 1.
 */
```
* Currently only works if this is the first gmac call in the thread.
* 
* \param acc index of the preferred accelerator
* 
* \return On success gmacMigrate returns gmacSuccess. Otherwise it returns
* the
* causing error
*/
GMAC_API gmacError_t APICALL gmacMigrate(unsigned acc);

GMAC_API gmacError_t APICALL
gmacCreateAddressSpace(GmacAddressSpaceId *aSpaceId, int accId);

GMAC_API gmacError_t APICALL
gmacDeleteAddressSpace(GmacAddressSpaceId aSpaceId);

GMAC_API gmacError_t APICALL
gmacCreateVirtualDevice(GmacVirtualDeviceId *vDeviceId, GmacAddressSpaceId aSpaceId);

GMAC_API gmacError_t APICALL
gmacDeleteVirtualDevice(GmacVirtualDeviceId vDeviceId);

/**
* Maps a range of CPU memory on the GPU. The memory pointed by cpuPtr must
* NOT have been allocated
* using gmacMalloc or gmacGlobalMalloc, and must not have been mapped
* before. Both, GPU and CPU,
* use the same addresses for this memory.
* \param cpuPtr CPU memory address to be mapped on the GPU
* \param count Number of bytes to be allocated
* \param prot The protection to be used in the mapping (currently unused)
* \return On success gmacMap returns gmacSuccess. Otherwise it returns the
* causing error
*/
GMAC_API gmacError_t APICALL gmacMemoryMap(void *cpuPtr, size_t count,
                                        GmacProtection prot);

/**
* Unmaps a range of CPU memory from the GPU. Both, GPU and CPU,
* use the same addresses for this memory.
* \param cpuPtr memory address to be unmapped from the GPU.
* \param count bytes to be allocated
* \return On success gmacUnmap returns gmacSuccess. Otherwise it returns
* the
* causing error
*/
GMAC_API gmacError_t APICALL gmacMemoryUnmap(void *cpuPtr, size_t count);

/**
* Allocates a range of memory in the GPU and the CPU. Both, GPU and CPU,
* use the same addresses for this memory.
* \param devPtr memory address to store the address for the allocated
* memory
* \param count bytes to be allocated
* \return On success gmacMalloc returns gmacSuccess and stores the address
* of the allocated
* memory in devPtr. Otherwise it returns the causing error
*/
GMAC_API gmacError_t APICALL gmacMalloc(void **devPtr, size_t count);
A GMAC API

/**
 * Gets the GPU address of an allocation performed with gmacMalloc or
 * gmacGlobalMalloc
 * \param cpuPtr memory address at the CPU
 * \return On success gmacPtr returns the GPU address of the allocation
 * pointed
 * by CPU cpuPtr. Otherwise it returns NULL
 */
GMAC_API __gmac_accptr_t APICALL gmacPtr(const void * cpuPtr);

/**
 * Free the memory pointed by cpuPtr. The memory must have been allocated
 * using
 * with gmacMalloc() or gmacGlobalMalloc()
 * \param cpuPtr Memory address to free. This address must have been
 * returned
 * by a previous call to gmacMalloc() or gmacGlobalMalloc()
 * \return On success gmacFree returns gmacSuccess. Otherwise it returns
 * the
 * causing error
 */
GMAC_API gmacError_t APICALL gmacFree(void * cpuPtr);

/**
 * Waits until all previous GPU requests have finished
 * \return On success gmacThreadSynchronize returns gmacSuccess. Otherwise
 * it returns
 * the causing error
 */
GMAC_API gmacError_t APICALL gmacThreadSynchronize();

/**
 * Returns the error code of the last gmac operation performed by the
 * calling thread
 * \return The error code of the last gmac operation performed by the
 * calling thread
 */
GMAC_API gmacError_t APICALL gmacGetLastError();

/**
 * Sets count bytes to c in the memory pointed by ptr
 * \param ptr A pointer to the memory to be set
 * \param c Value to be set
 * \param count Number of bytes to be set
 * \return A pointer to ptr
 */
GMAC_API void * APICALL gmacMemset(void *ptr, int c, size_t count);

/**
 * Copies count bytes from the memory pointed by src to the memory pointed
 * by dst
 * \param dst Pointer to destination memory
 * \param src Pointer to source memory
 * \param count Number of bytes to be copied
 */
A GMAC API

* \( \text{return A pointer to dst} \)

```c
GMAC_API void * APICALL gmacMemcpy (void *dst, const void *src, size_t count);
```

/**
 * Sends the execution mode of the current thread to the thread identified by tid
 * \param tid The identifier of the destination thread
 */
```c
GMAC_API void APICALL gmacSend(THREAD_T tid);
```

/**
 * The current thread receives the execution mode that is sent by another thread using gmacSend or gmacSendReceive
 */
```c
GMAC_API void APICALL gmacReceive();
```

/**
 * Sends the execution mode of the current thread to the thread identified by tid
 * receives the execution mode that is sent by another thread using gmacSend/gmacSendReceive
 * \param tid The identifier of the destination thread
 */
```c
GMAC_API void APICALL gmacSendReceive(THREAD_T tid);
```

/**
 * Copies the execution mode of the current thread to the thread identified by tid
 */
```c
GMAC_API void APICALL gmacCopy(THREAD_T tid);
```

/**
 * Returns a description of the given error
 * \param err An error code
 * \return A string with the description of the error code
 */
```c
static const char *gmacGetErrorString(gmacError_t err);
```
A GMAC API

A.2 DSM manager interface

Below we show the DSM manager interface.

```cpp
/**
 * Default constructor
 *
 * \return manager instance class
 */
manager(size_t BS)

/**
 * Default destructor
 */
virtual ~manager()

/**
 * Check if one mapping with a specific size will fit in the group
 * That means, if the group have enough space to hold a new mapping
 *
 * \param map mapping’s group
 * \param m mapping to check if it’s fit
 *
 * \return bool true if it’s fit, false otherwise
 */
bool mapping_fits(map_mapping &map, mapping_ptr m)

/**
 * Get the mappings from the address space, to avoid extra map
 *
 * \return map_mapping_group mapping’s group
 */
map_mapping_group &get_aspace_mappings(hal::virt::aspace &as)

/**
 * Get all the mappings in a range
 * From the base allocation begin to some size count
 * If we don’t found the base base allocation, return an empty range
 * And if no mappings is affected, return an empty range
 *
 * \param mappings pointer to mapping’s group
 * \param begin base allocation for the address
 * \param count size of elements to return
 *
 * \return range_mapping mappings in the range
 */
template <bool GetAdjacent, typename Ptr>
static
range_mapping get_mappings_in_range(map_mapping_group &mappings, Ptr begin,
                                      size_t count)

/**
 * Check if a range of mapping has a certain protection
 * The parameter all:
```
A GMAC API

- if is set, all the range need to has the protection prot
- if not set, some one need has the protection prot

\param range pointer to mapping's range
\param prot protection to have

\return bool true if the range has the protection, false otherwise

\template <bool All>
static
bool range_has_protection(const range_mapping &range, GmacProtection prot)

/**
 * Insert a mapping in the group
 *
 * param mappings pointer to mapping's group
 * \param m pointer to mapping that will be inserted
 *
 * return error code of the operation
 */
error insert_mapping(map_mapping_group &mappings, mapping_ptr m)

/**
 * Merge mappings into only one
 * If the range is empty, this method will trigger an error
 *
 * param range pointer to mapping's range to be merged
 *
 * return mapping_ptr the new merged mapping
 */
mapping_ptr merge_mappings(range_mapping &range)

/**
 * Replace the whole range of mappings with a single mapping
 *
 * param mappings pointer to mapping's group
 * \param range pointer to mapping's range to be replaced by mNew
 * \param mNew pointer to the new mapping
 *
 * return error code of the operation
 */
error replace_mappings(map_mapping_group &mappings, range_mapping &range, mapping_ptr mNew)

/**
 * Delete all mapping from the group
 *
 * param group pointer to mapping's group
 *
 * return error code of the operation
 */
error delete_mappings(map_mapping_group &mappings)

/**
 * Delete a specific range of mappings from the group
 *
 * param group pointer to mapping's group
 * \param range pointer to mapping's range to delete
 * 
 */
A GMAC API

```c
error delete_mappings(map_mapping_group &group, range_mapping &range)

/**
 * Creates a mapping between the space of two devices
 * 
 * \param dst pointer to destination device
 * \param src pointer to source device
 * \param count size of elements to link
 * \param protDst destination protection that blocks will have
 * \param protSrc destination protection that blocks will have
 * \param flags for future versions
 * 
 * \return error code of the operation
 */
error link(hal::ptr dst, hal::ptr src, size_t count, GmacProtection protDst , GmacProtection protSrc, int flags = mapping_flags::MAP_DEFAULT)

/**
 * Undo the creation of the mapping between two devices
 * 
 * \param mapping pointer to mapping
 * \param count size of elements to undo
 * 
 * \return error code of the operation
 */
error unlink(hal::ptr mapping, size_t count)

/**
 * Print to console a specific range of mappings. For debug purpose
 * 
 * \param range pointer to range mapping
 * 
 */
template <bool Hex, bool PrintBlocks>
static
void range_print(const range_mapping &range)

/**
 * Print to console all the mapping. For debug purpose
 * 
 * \param as pointer to address space
 * 
 */
template <bool Hex, bool PrintBlocks>
void print_all_mappings(hal::virt::aspace &as)

/**
 * Handle page fault caused by an access to a protected region
 * 
 * \param p pointer of the region
 * \param isWrite boolean that indicates if the protection is for write
 * 
 * \return error code of the operation
 */
static
bool handle_fault(hal::ptr p, bool isWrite)
```
A GMAC API

/**
 * Protect a region of memory
 *
 * \param as address space pointer
 *
 * \return error code of the operation
 */
error use_memory_protection(hal::virt::aspace &as)

/**
 * Retrieve to the caller memory a number of blocks and lock them.
 * Apply memory coherence protocol
 *
 * \param mapping pointer to device mapping
 * \param count size of elements to lock and retrieve
 *
 * \return error code of the operation
 */
error acquire(hal::ptr mapping, size_t count, GmacProtection prot)

/**
 * Unlock the blocks that we locked with acquire function.
 * Apply memory coherence protocol
 *
 * \param mapping pointer to device mapping
 * \param count size of elements unlock
 *
 * \return error code of the operation
 */
error release(hal::ptr mapping, size_t count)

/**
 * Copy count bytes from src device to dst device
 *
 * \param dst destination pointer to copy
 * \param src source pointer of the copy
 * \param count size of elements to write
 *
 * \return error code of the operation
 */
error memcpy(hal::ptr dst, hal::ptr src, size_t count)

/**
 * Write count value c from ptr
 *
 * \param ptr source pointer of device
 * \param c value to write
 * \param count size of elements to write
 *
 * \return error code of the operation
 */
error memset(hal::ptr ptr, int c, size_t count)

/**
 * Read count bytes from input device file to destination ptr
 *
 * \param addr destination pointer
 * \param input source device file
 * \param count size of elements to copy
 */
A GMAC API

*\return error code of the operation*

```c
error from_io_device(hal::ptr addr, hal::device_input &input, size_t count)
```

/**
 * Write count bytes from addr to output device file
 *
 * \param output destination device file
 * \param addr source pointer of the copy
 * \param count size of elements to write
 *
 * \return error code of the operation
 */

```c
error to_io_device(hal::device_output &output, hal::const_ptr addr, size_t count)
```

/**
 * Get the block size
 *
 * \return block size
 */

```c
size_t get_BS() const
```
Matrix Multiplication

In this appendix, we show a real example of a program to illustrate how would the code be written in three different ways:

- **C++ source code.** Here we report the code written in C++ original.
- **CUDA source code.** Here we report the code written in CUDA.
- **GMAC source code.** Here we report the code written in GMAC.

The example we have chosen is the matrix-matrix multiplication.

**B.1 C++ Source Code**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Sequential Matrix dot product
void matrixProduct(float *N, float *M, float *P, unsigned long int n) {
    unsigned long int i, j, k;
    float sum;
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            sum = 0.0;
            for (k = 0; k < n; ++k) {
                sum = sum + (N[i*n+k] * M[k*n+j]);
            }
            P[i*n+j] = sum;
        }
    }
}

int main(int argc, char *argv[]) {
    unsigned long int n, i, j;

    if(argc != 2) {
        fprintf(stderr, "Usage: %s N\n\tN: Number of columns and rows of the matrices\n", argv[0]);
        exit(-1);
    }
```
n = atol(argv[1]);

//Allocate memory for the 3 matrices
N = (float *) malloc(sizeof(float)*n*n);
M = (float *) malloc(sizeof(float)*n*n);
P = (float *) malloc(sizeof(float)*n*n);
if ( N == 0 || M == 0 || P == 0) {
    perror("Error at malloc");
    exit(-1);
}

//Read matrix N
for (i =0; i<n*n; ++i) {
    if( fscanf(stdin, "%lf", &N[i]) <0) {
        perror("Error reading matrix N");
        exit(-1);
    }
}

//Read matrix M
for (i =0; i<n*n; ++i) {
    if( fscanf(stdin, "%lf", &M[i]) <0) {
        perror("Error reading matrix M");
        exit(-1);
    }
}

//Compute matrix product of N with M and store the result at P
matrixProduct(N, M, P, n);

//Output of P
for (i = 0; i<n; ++i) {
    for (j = 0; j<n; ++j) {
        if( fprintf(stdout, "%e ", P[i*n + j]) <0) {
            perror("Error writing matrix P");
            exit(-1);
        }
    }
    if( fprintf(stdout, "\n") <0) {
        perror("Error writing endline of matrix P");
        exit(-1);
    }
}

//Free memory
free(N);
free(M);
free(P);
return 0;

Listing B.1: C++ matrix-matrix multiplication
B.2 CUDA Source Code

/*
 * This is a implementation of tiling algorithm using Shared Memory
 * Each thread block calculate a sub-matrix (block of TILE_WIDTH x TILE_WIDTH);
 * Now, all thread in a block colaborate to load data from global memory to
 * shared memory. Reuse data and reduce global memory traffic
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TILE_WIDTH 16
#define FLOAT float

__global__ void matrixProdKernel(FLOAT *Nd, FLOAT *Md, FLOAT *Pd, unsigned long int n)
{
    __shared__ FLOAT Nds[TILE_WIDTH][TILE_WIDTH];
    __shared__ FLOAT Mds[TILE_WIDTH][TILE_WIDTH];

    // Thread and block identification
    unsigned int bx = blockIdx.x; unsigned int by = blockIdx.y;
    unsigned int tx = threadIdx.x; unsigned int ty = threadIdx.y;

    // Identify the row and column.
    unsigned int row = by * blockDim.y /* TILE_WIDTH*/ + ty;
    unsigned int col = bx * blockDim.x /* TILE_WIDTH*/ + tx;

    if (row > n || col > n) return;

    FLOAT Pvalue = 0.0;
    unsigned long int k, m;

    for (m = 0; m < (n / TILE_WIDTH); m++)
    {
        // Collaborative loading of Nd and Md to shared memory
        Nds[ty][tx] = Nd[row * n + (m * TILE_WIDTH + tx)];
        Mds[ty][tx] = Md[(m * TILE_WIDTH + ty) * n + col];
        __syncthreads(); // barrier

        for(k = 0; k < TILE_WIDTH; k++)
        { Pvalue += Nds[ty][k] * Mds[k][tx];
        __syncthreads();
        }
    }

    Pd[row * n + col] = Pvalue;
}

// Matrix dot product with CUDA
void matrixProduct(FLOAT *N, FLOAT *M, FLOAT *P, unsigned long int n, unsigned char wp)
B Matrix Multiplication

FLOAT *Nd, *Md, *Pd;
unsigned long int size = sizeof(FLOAT) * n * n;

cudaEvent_t start_event, end_event;
float kernel_time, h2d_time, d2h_time, flops_kernel, flops_hkd;

// Allocate device memory
cudaMalloc((void **) &Nd, size);
cudaMalloc((void **) &Md, size);
cudaMalloc((void **) &Pd, size);

// Create the cuda events
cudaEventCreate(&start_event);
cudaEventCreate(&end_event);

// Copy M and N to allocated device memory
cudaEventRecord(start_event);
cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
cudaEventRecord(end_event);

cudaEventSynchronize(end_event);

cudaEventElapsedTime(&h2d_time, start_event, end_event);

// Setup the execution configuration
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1); // Configuration of blocks.
dim3 dimGrid(n/TILE_WIDTH, n/TILE_WIDTH, 1); // Configuration of grid.
How many blocks are in each grid

// kernel invocations code - to have the device
// to perform the actual matrix multiplication
cudaEventRecord(start_event);
matrixProdKernel<<<dimGrid, dimBlock>>>(Nd, Md, Pd, n);
cudaEventRecord(end_event);

cudaEventSynchronize(end_event);

cudaEventElapsedTime(&d2h_time, start_event, end_event);

flops_kernel = (2e-6 * n * n * n) / kernel_time;
flops_hkd = (2e-6 * n * n * n) / (kernel_time + h2d_time + d2h_time);

if (wp == 1)
{
    fprintf(stdout, "##################################\n");
    fprintf(stdout, "##### Shared memory kernel : ######\n");
    fprintf(stdout, "Kernel Time: %.2f (ms)\n", kernel_time);
    fprintf(stdout, "H2D cpy Time: %.2f (ms)\n", h2d_time);
B Matrix Multiplication

```c
int main(int argc, char *argv[]) {
    unsigned long int n, i, j;
    unsigned char wp;

    if(argc != 3) {
        fprintf(stdout, "Usage: %s N {1,0}\n	N: Number of columns and rows of the matrices\n\t{1,0}: 1- to print counting; 0- to print results\n" , argv[0]);
        exit(-1);
    }

    n = atol(argv[1]);
    wp = atoi(argv[2]);

    // Allocate memory for the 3 matrices
    N = (FLOAT *) malloc(sizeof(FLOAT) * n * n);
    M = (FLOAT *) malloc(sizeof(FLOAT) * n * n);
    P = (FLOAT *) malloc(sizeof(FLOAT) * n * n);

    if (N == 0 || M == 0 || P == 0) {
        perror("Error at malloc");
        exit(-1);
    }

    // Read matrix N
    for(i=0; i<n*n; ++i) {
        if(fscanf(stdin, "%f", &N[i])<0) {
            perror("Error reading matrix N");
            exit(-1);
        }
    }

    // Read matrix M
    for(i=0; i<n*n; ++i) {
        if(fscanf(stdin, "%f", &M[i])<0) {
            perror("Error reading matrix M");
            exit(-1);
        }
    }

    // Compute matrix product of N with M and store the result at P
    matrixProduct(N, M, P, n, wp);

    // Destroy events
    cudaEventDestroy(start_event);
    cudaEventDestroy(end_event);

    // Free memory
    cudaFree(Nd);
    cudaFree(Md);
    cudaFree(Pd);
}
```

if (wp == 0)
{
    // Output of P
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            if (fprintf(stdout, "%e ", P[i*n + j]) < 0) {
                perror("Error writing matrix P");
                exit(-1);
            }
        }
        if (fprintf(stdout, "\n") < 0) {
            perror("Error writing endline of matrix P");
            exit(-1);
        }
    }
    // Free memory
    free(N);
    free(M);
    free(P);
    return 0;
}

Listing B.2: CUDA matrix-matrix multiplication
/ B. Matrix Multiplication

B.3 GMAC Source Code

```
/*
 * This is a implementation of tiling algorithm using Shared Memory
 * Each thread block calculate a sub-matrix (block of TILE_WIDTH x TILE_WIDTH);
 * Now, all thread in a block colaborate to load data from global memory to
 * shared memory. Reuse data and reduce global memory traffic
 * This version uses the gmac library
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gmac/cuda.h>

#define TILE_WIDTH 16
#define FLOAT float

__global__ void matrixProdKernel(FLOAT *Nd, FLOAT *Md, FLOAT *Pd, unsigned long int n)
{
    __shared__ FLOAT Nds[TILE_WIDTH][TILE_WIDTH];
    __shared__ FLOAT Mds[TILE_WIDTH][TILE_WIDTH];

    // Thread and block identification
    unsigned int bx = blockIdx.x; unsigned int by = blockIdx.y;
    unsigned int tx = threadIdx.x; unsigned int ty = threadIdx.y;

    // Identify the row and column.
    unsigned int row = by * blockDim.y /* TILE_WIDTH */ + ty;
    unsigned int col = bx * blockDim.x /* TILE_WIDTH */ + tx;

    if (row > n || col > n) return;

    FLOAT Pvalue = 0.0;
    unsigned long int k, m;

    for (m = 0; m < (n / TILE_WIDTH); m ++)
    {
        // Collaborative loading of Nd and Md to shared memory
        Nds[ty][tx] = Nd[row * n + (m * TILE_WIDTH + tx)];
        Mds[ty][tx] = Md[(m * TILE_WIDTH + ty) * n + col];
        __syncthreads(); // barrier

        for(k = 0; k < TILE_WIDTH; k++)
            Pvalue += Nds[ty][k] * Mds[k][tx];
        __syncthreads();
    }

    Pd[row * n + col] = Pvalue;
}

// Matrix dot product wit CUDA
```

void matrixProduct (FLOAT *N, FLOAT *M, FLOAT *P, unsigned long int n, 
unsigned char wp)
{
    unsigned long int size = sizeof (FLOAT) * n * n;

cudaEvent_t start_event, end_event;
float kernel_time , flops_kernel ;

// Create the cuda events
    cudaEventCreate (& start_event);
    cudaEventCreate (& end_event);

// Setup the execution configuration
    dim3 dimBlock (TILE_WIDTH , TILE_WIDTH , 1); // Configuration of blocks.
    dim3 dimGrid (n / TILE_WIDTH , n / TILE_WIDTH , 1); // Configuration of grid.

// kernel invocations code - to have the device
// to perform the actual matrix multiplication
    cudaEventRecord (start_event);
    matrixProdKernel <<< dimGrid , dimBlock >>>(N , M , P , n);
    cudaEventRecord (end_event);

    cudaEventSynchronize (end_event);
    cudaEventElapsedTime (& kernel_time , start_event , end_event);

    flops_kernel = (2e-6 * n * n * n) / kernel_time;
    if (wp == 1)
    {
        fprintf (stdout , "##################################
        
        ##### Shared memory kernel : ######
        
        Kernel Time : %.2f (ms)
        
        Kernel GFLOPS : %.2f
        
        ##################################
        
        ");
    }

// Destroy events
    cudaEventDestroy (start_event);
    cudaEventDestroy (end_event);
}

int main ( int argc , char * argv[] )
{
    unsigned long int n , i , j;
    unsigned char wp;

    if (argc != 3) 
    {
        fprintf (stdout , "Usage : %s N {1,0}\n\tN: Number of columns and rows
\t{1,0}: 1- to print counting ; 0- to print results\n", argv[0]);
        exit (-1);
    }

    n = atol (argv[1]) ;
    wp = atoi (argv[2]) ;

    // Allocate memory for the 3 matrices
B Matrix Multiplication

```c
if ( N == 0 || M == 0 || P == 0) {
    perror("Error at malloc");
    exit(-1);
}

// Read matrix N
for (i = 0; i < n * n; ++i) {
    if (fscanf(stdin, "%f", &N[i]) < 0) {
        perror("Error reading matrix N");
        exit(-1);
    }
}

// Read matrix M
for (i = 0; i < n * n; ++i) {
    if (fscanf(stdin, "%f", &M[i]) < 0) {
        perror("Error reading matrix N");
        exit(-1);
    }
}

// Compute matrix product of N with M and store the result at P
matrixProduct(N, M, P, n, wp);

if (wp == 0) {
    // Output of P
    for (i = 0; i < n; ++i) {
        for (j = 0; j < n; ++j) {
            if (fprintf(stdout, "%e \n", P[i * n + j]) < 0) {
                perror("Error writing matrix P");
                exit(-1);
            }
        }
    }
}

// Free memory
free(N);
free(M);
free(P);
return 0;
```

Listing B.3: GMAC matrix-matrix multiplication
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Performance evolution gap between GPUs and CPUs</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Performance benchmark between GPUs and CPUs</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>CPUs and GPUs: different design philosophy</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>CPU and GPU interconnection using PCIe with connection bandwidth</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>Matrix-matrix multiplication example</td>
<td>14</td>
</tr>
<tr>
<td>3.6</td>
<td>Execution of a CUDA program</td>
<td>15</td>
</tr>
<tr>
<td>3.7</td>
<td>CUDA device memory model</td>
<td>17</td>
</tr>
<tr>
<td>3.8</td>
<td>CUDA thread organization</td>
<td>20</td>
</tr>
<tr>
<td>3.9</td>
<td>Mapping between OpenCL and CUDA concepts</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>GMAC overall design</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>GMAC memory model</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>Programming step overview</td>
<td>27</td>
</tr>
<tr>
<td>5.1</td>
<td>Shared address space. Mapping and blocking</td>
<td>30</td>
</tr>
<tr>
<td>5.2</td>
<td>MSI protocol</td>
<td>32</td>
</tr>
<tr>
<td>5.3</td>
<td>MSI protocol improvement</td>
<td>34</td>
</tr>
<tr>
<td>6.1</td>
<td>Test output from DSM layer part 1</td>
<td>38</td>
</tr>
<tr>
<td>6.2</td>
<td>Test output from DSM layer part 2</td>
<td>38</td>
</tr>
<tr>
<td>6.3</td>
<td>Test output from HAL layer</td>
<td>39</td>
</tr>
<tr>
<td>7.1</td>
<td>Gantt diagram at the start of the project</td>
<td>41</td>
</tr>
<tr>
<td>7.2</td>
<td>Gantt diagram at the end of the project</td>
<td>42</td>
</tr>
<tr>
<td>7.3</td>
<td>Project methodology diagram</td>
<td>46</td>
</tr>
<tr>
<td>7.4</td>
<td>Top 500 list (November 2013)</td>
<td>47</td>
</tr>
<tr>
<td>7.5</td>
<td>Green 500 list (November 2013)</td>
<td>48</td>
</tr>
</tbody>
</table>
# List of Tables

3.1 CUDA variable qualifiers type .................................................. 18  
3.2 CUDA extensions to C functional declaration ................................ 19  
6.1 Test equipment hardware .......................................................... 35  
7.1 Gantt diagram comparison hours ............................................... 43  
7.2 Estimated cost for each role type of human resources ..................... 43  
7.3 Total cost for human resources related to each task work ................ 44  
7.4 Estimated cost for each type of material resources ....................... 45  
7.5 Cost summary ........................................................................... 45
List of Code Samples

3.1 A simple CUDA program structure ........................................... 15
3.2 A simple CUDA kernel function .................................................. 19
3.3 A CUDA kernel invocation .......................................................... 20
3.4 A complete CUDA application .................................................... 21

4.1 A GMAC application ................................................................. 27

5.1 Prototype of link function ........................................................... 30
5.2 Prototype of unlink function ......................................................... 31
5.3 Error code ................................................................................. 31
5.4 Prototype of acquire function ....................................................... 33
5.5 Prototype of release function ......................................................... 33

6.1 Prototype of the cubic function ...................................................... 36
6.2 GoogleTest example ................................................................. 36

B.1 C++ matrix-matrix multiplication ............................................... 65
B.2 CUDA matrix-matrix multiplication ............................................ 67
B.3 GMAC matrix-matrix multiplication .............................................. 71