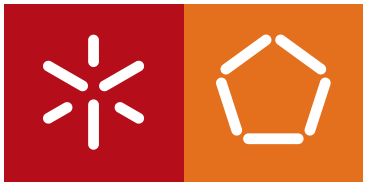
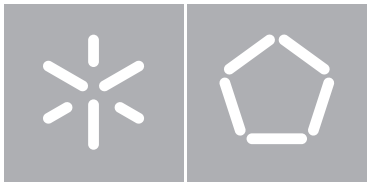


Universidade do Minho
Escola de Engenharia

André Martins Pereira

Efficient processing of ATLAS events analysis in platforms with accelerator devices

Fevereiro de 2013



Universidade do Minho
Escola de Engenharia
Departamento de Informática

André Martins Pereira

Efficient processing of ATLAS events analysis in platforms with accelerator devices

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor Alberto Proença
Professor António Onofre

Fevereiro de 2013

Resumo

A maior parte das tarefas de análise de dados de eventos no projeto ATLAS requerem grandes capacidades de acesso a dados e processamento, em que a performance de algumas das tarefas são limitadas pela capacidade de I/O e outras pela capacidade de computação.

Esta dissertação irá focar-se principalmente nos problemas limitados computacionalmente nas últimas fases de análise dos dados do detector do ATLAS (as calibrações), complementando uma dissertação paralela que irá lidar com as tarefas limitadas pelo I/O.

O principal objectivo deste trabalho será desenhar, implementar, validar e avaliar uma tarefa de análise mais robusta e melhorada, que envolve aperfeiçoar a performance da reconstrução kinemática de eventos dentro da framework usada para a análise de dados no ATLAS, a ser executada em plataformas de computação heterogénea, baseadas em CPUs multicore acoplados a placas PCI-E com dispositivos many-core, tais como o Intel Xeon Phi e/ou os dispositivos GPU NVidia Fermi/Kepler.

Uma aplicação de análise será usada como caso de estudo, desenvolvida pelo grupo LIP, para melhorar a reconstrução kinemática, bem como restruturar e paralelizar outras regiões críticas desta análise.

Uma framework experimental, GAMA, será usada para automatizar (i) a distribuição de carga pelos recursos disponíveis e (ii) a gestão transparente de dados através do ambiente de memória física distribuída, entre a memória partilhada do CPU multicore e da memória dos dispositivos many-core. A eficiência e usabilidade do GAMA será avaliada e comparada com outras frameworks concurrentes.

Abstract

Most event data analysis tasks in the ATLAS project require both intensive data access and processing, where some tasks are typically I/O bound while others are compute bound.

This dissertation work will mainly focus on compute bound issues at the latest stages of the ATLAS detector data analysis (the calibrations), complementing a parallel dissertation work that addresses the I/O bound issues.

The main goal of the work is to design, implement, validate and evaluate an improved and more robust data analysis task which involves tuning the performance of the kinematical reconstruction of events within the framework used for data analysis in ATLAS, to run on heterogeneous computing platforms based on multi-core CPU devices coupled to PCI-E boards with many-core devices, such as the Intel Xeon Phi and/or the NVidia Fermi/Kepler GPU devices.

As a case study, an analysis application will be used, developed by the LIP research group at University of Minho, to tune the kinematical reconstruction, as well as restructure and parallelize other critical areas of this analysis.

An experimental framework, GAMA, will be used to automate (i) the workload distribution among the available resources and (ii) the transparent data management across the physical distributed memory environment between the shared multi-core memory and the many-core device memory. The GAMA efficiency and usability will be assessed and compared against a concurrent framework.

Contents

Glossary

Event head-on collision between two particles at the LHC

LHC Large Hardron Collider particle accelerator

ATLAS project Experiment being conducted at the LHC with an associated particle detector

LIP Laboratório de Instrumentação e Física Experimental de Partículas, Portuguese research group working in the ATLAS project

CERN European Organization for Nuclear Research, which results from a collaboration from many countries to test HEP theories

HEP High Energy Physics

Analysis Application developed to process the data gathered by the ATLAS detector and test a specific HEP theory

Accelerator device Specialized processing unit connected to the system by a PCI-Express interface

CPU Central Processing Unit, which may contain one or more cores (multicore)

GPU Graphics Processing Unit

GPGPU General Purpose Graphics Processing Unit, recent designation to scientific computing oriented GPUs

DSP Digital Signal Processor

MIC Many Integrated Core, accelerator device architecture developed by Intel, also known as Xeon Phi

Homogeneous system Classic computer system, which contain one or more similar multicore CPUs

Heterogeneous system Computer system, which contains a multicore CPU and one or more accelerator devices

SIMD Single Instruction Multiple Data, describes a parallel processing architecture where a single instruction is applied to a large set of data simultaneously

SIMT Single Instruction Multiple Threads, describes the processing architecture that NVidia uses, very similar to SIMD, where a thread is responsable for a subset of the data to process

SM/SMX Streaming Multiprocessor, SIMT/SIMD processing unit available in NVidia GPUs

Kernel Parallel portion of an application code designed to run on a CUDA capable GPU

Host CPU in a heterogeneous system, using the CUDA designation

CUDA Compute Unified Device Architecture, a parallel computing platform for GPUs

OpenMP Open Multi-Processing, an API for shared memory multiprocessing

OpenACC Open Accelerator, an API to offload code from a host CPU to an attached accelerator

GAMA GPU and Multicore Aware, an API for shared memory multiprocessing in platforms with a host CPU and an attached CUDA enabled accelerators

List of Figures

1 Introduction

The Large Hardron Collider (LHC) [1] is a high-energy particle accelerator, located in the underground in both sides of the border between Switzerland and France, built by the European Organization for Nuclear Research (CERN) [2]. It results from a cooperation between many countries, involving thousands of scientists around the world. The LHC is used to conduct experiments to validate high-energy physics (HEP) theories. Proving the existence of the Higgs boson is currently one of the most popular.

At the LHC, an experiment usually refers to a head-on collision of particles (which is considered an event), where detectors gather data related to the collision. There are different detectors with different purposes according to the experiments that they were built for, but these usually capture information related to the particles resultant from the head-on collision, such as their mass, momentum and energy. There are six detectors spread along the LHC, where millions of particle collisions occur each second, generating massive amounts of data to process [3].

The information gathered passes through a set of computational tiers, in which the data is refined and scattered among the distributed computational resources, until it is ready to be used in simulations by the research groups, where it is analyzed [4].

ATLAS [5] is one of the main experiments being conducted at the LHC. The Laboratório de Instrumentação e Física Experimental de Partículas (LIP) [6] is one of the research groups involved in supporting and analyzing the data of this experiment. LIP continuously performs analysis on the data gathered by the ATLAS detector, competing against other research groups, within the ATLAS project, to obtain and publish relevant results.

The focus of this dissertation will be on tuning and parallelizing the kinematical reconstruction of events, using as case study a specific analysis application, the `ttH_dilep`, developed by LIP. The kinematical reconstruction is widely used within the group, but the work will not be restricted to only the kinematical reconstruction. Other application specific tasks will be analyzed to improve their performance in order to (i) get the maximum efficiency from the tuned kinematical reconstruction, and (ii) improve the overall performance of the analysis.

The tuning of both the kinematical reconstruction and overall improvement of the application performance will address heterogeneous architectures, which combine conventional multicore processors with accelerator devices in the same system. Efficient porting from sequential execution to these heterogeneous environments places a series of challenges, such as different architectural and programming paradigms, and load balancing of the tasks and data between CPU and accelerators. The main efforts will be towards obtaining an optimized implementation of the kinematical reconstruction possible for the heterogeneous platforms, specifically to run the kinematical reconstruction, and possibly other tasks, on accelerator devices. These platforms will be presented in section 3, with a special focus on the accelerator devices.

There are several development frameworks that aim to ease the programmer burden of these challenges. Usually, they attempt to create a level of abstraction between the architectural details of the heterogeneous environments, and consequently the programing paradigm, and the programming environment. The GAMA and OpenACC frameworks will be tested in the context of this problem, and the implementations using these frameworks will be compared to the optimized implementations mentioned previously, in terms of performance, usability and development time.

2 Contextualization

The LHC accelerates two particle beams in opposite directions, causing them to collide at the particle detectors. From this head-on collision between two particles results a limited chain reaction of decaying particles, where the detector records the characteristics of most of the final particles. A schematic representation of the head-on collision, and respective particle decay, is presented in figure 2.1, and it is known as the `ttbar` system. The detected particles are the bottom quarks (which are detected as a jet of particles) and leptons (electron and muon), while the neutrinos do not react with the detector and, therefore, are not detected. To reconstruct the collision, the characteristics of these neutrinos must be determined. Since this system obeys a set of properties, related to the calibrated model expected from the collision, it is possible to analytically determine the neutrinos characteristics and reconstruct the event (kinematical reconstruction), and then estimate the degree of certainty associated with the computed reconstruction.

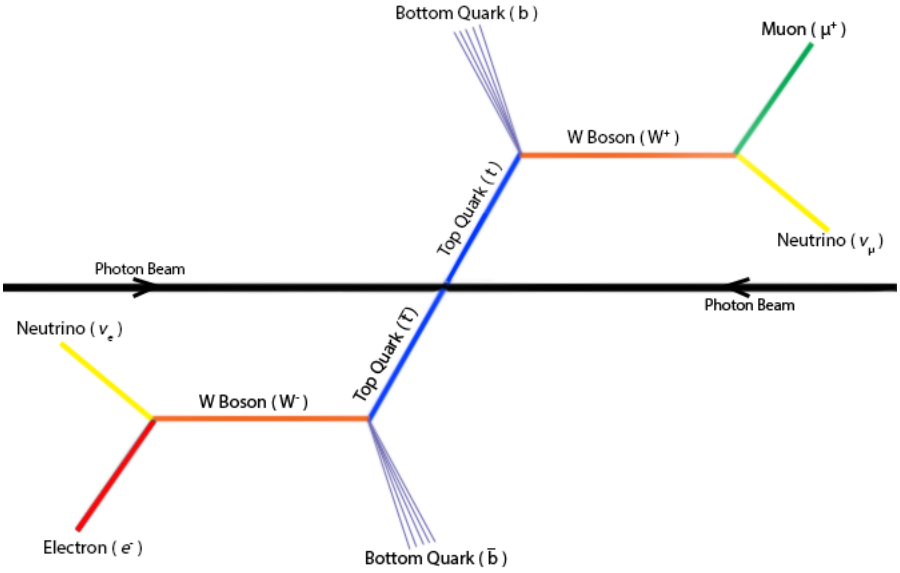


Figure 2.1: Schematic representation of the `ttbar` system.

The amount of bottom quark jets and leptons detected can vary between events. However, it is needed at least 2 jets and 2 leptons are needed to reconstruct the `ttbar` system, as represented in the figure 2.1, but their amount can reach up to 14. Some of the jets/leptons may not belong to the `ttbar` system (as other reactions occur during the collision), so there is a need to choose the ones that most accurately reconstruct the system. Note that the quality of the reconstruction has a strong impact the quality of the research being conducted by the LIP group.

By performing the kinematical reconstruction to each combination of all the bottom quark jets and leptons, two by two, and computing the probability associated with the respective reconstruction, it is possible to chose only the combination that results on the most accurate reconstruction.

Another factor that can affect the accuracy of the reconstruction is the experimental resolution associated with the ATLAS detector. The detected values for the particles (bottom quark jets and leptons) are not fully accurate: the measurements made by ATLAS can have a 2% fluctuation to the real values. Since these particles are used in the kinematical reconstruction, its accuracy can be impaired. To improve the quality of these reconstructions, the experimental resolution must be compensated. This can be achieved by varying the values of the bottom quark jets and leptons characteristics, such as the mass

or momentum, and use them in the kinematical reconstruction. However, this cannot be performed only once: the search space must be covered a certain amount of times to get higher probability of finding a great reconstruction. This means running the kinematical reconstruction as many times as possible, per event, with different variations of the original inputs (jet/lepton combination).

The execution time of the analysis is critical due to the large amounts of data (events) that must be processed. Since for each event it is necessary to reconstruct all the bottom quark jets and leptons combinations, and for each combination a variation is applied several times, the number of kinematical reconstructions per event can quickly rise, increasing the overall time to process an event. A must be achieved balance between the required quality of the reconstruction, directly related to the number of times that the kinematical reconstruction is performed, and the time that takes to process an event.

The relevance of the kinematical reconstruction (dilep) is even greater in the `ttH_dilep` analysis. This analysis aims to reconstruct the Higgs boson based on the two jets that decay from it. Figure 2.2 schematically represents the `ttbar` system with the Higgs boson decay and respective jets. However, the jets that decay from the Higgs boson cannot be differentiated from the other similar jets. After performing the `ttbar` system reconstruction, i.e., the kinematical reconstruction, and considering the jets used in its best reconstruction, the application uses the remaining jets to reconstruct the Higgs boson. If an event `ttbar` system is not properly reconstructed, the Higgs boson reconstruction will not be accurate. The best final reconstruction is the one with the higher combined probabilities of the best kinematical reconstruction and the respective best Higgs boson reconstruction.

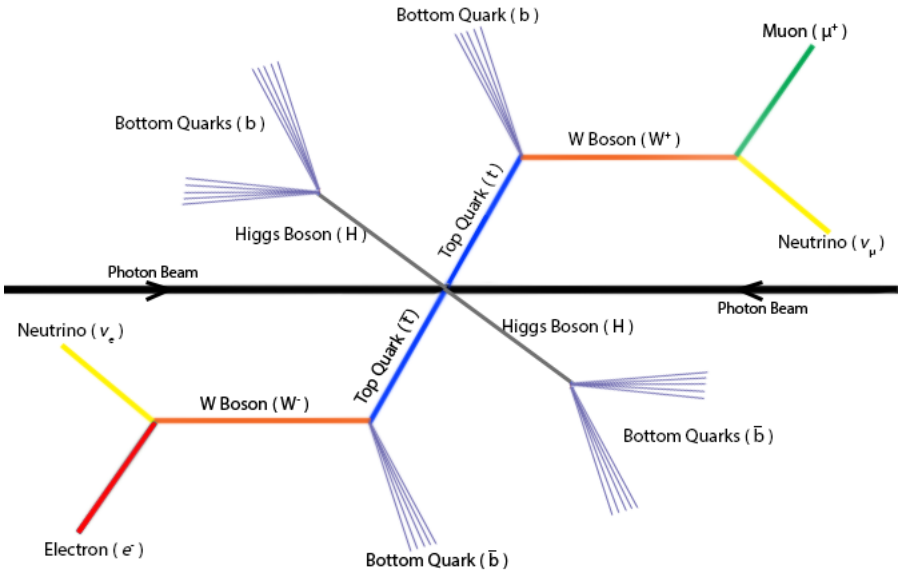


Figure 2.2: Schematic representation of the `ttbar` system with the Higgs boson decay.

By increasing the performance of the kinematical reconstruction it is possible to compute more reconstructions per event, leading to better and more accurate results. However, it is not possible to narrow the scope of this dissertation work only to the reconstruction; to get the most efficiency from it, it is necessary to consider other tasks to improve, such as the jet combination, variance appliance and Higgs reconstruction, and eventually re-design the workflow of this section of the application. The LIP research group needs to improve the performance of both kinematical reconstruction and the overall `ttH_dilep` analysis, to improve the overall results quality of the ATLAS project, giving them an advantage over the other teams.

3 State of the Art

Most of today's programmers produce code and design applications using sequential programming paradigms. The application behaviour is designed and tested only for sequential execution, where the only parallelism is performed by the compiler at the instruction level, or by the hardware through the use of superscalarity and out-of-order execution. A few years ago a transition from single core very fast CPUs to slightly slower multicore CPUs started to happen. Since it was not possible to make CPUs faster by increasing their clock frequency due to heat dissipation problems, manufacturers opted to include many CPUs (now designated as cores) in the same chip, interconnected and sharing the same memory space. Now, instead of producing faster CPUs, manufacturers focused on increasing their processing throughput. Unfortunately, these newer CPUs (now including many cores) need a different programming paradigm to get the most performance possible when designing an application; programmers training did not accompany this transition to parallel programming paradigms that take advantage of this increase in processing throughput.

Programming for multicore environments require some knowledge of the underlying architectural concepts. Shared memory, cache coherence and consistency and data races are architectural aspects that the programmer did not have to face in sequential execution environments. Now, when designing an application, all these aspects must be taken into account, not only to ensure efficient use of the computational resources, but also the correctness of the application.

Heterogeneous computer architectures are becoming increasingly popular. They combine the flexibility of multicore CPUs with the specific capabilities of many-core accelerator devices, connected by PCI-Express interfaces. However, most computational algorithms and applications are designed with the specific characteristics of CPUs in mind. Even multithreaded applications cannot be easily ported to these devices expecting high performance. To optimize the code for these specific devices it is necessary to deeply understand the architectural principles behind their design.

These devices are usually made from small processing units, designed to achieve the most performance possible on specific problem domains, as opposed to conventional CPUs. They are oriented for massive data parallelism processing (SIMD architectures), offloading the CPU from such data intensive operations. Several many-core accelerator devices are available, ranging from the general purpose GPUs, to the Intel Many Integrated Core line, currently known as Intel Xeon Phi [7], and Digital Signal Processors (DSP) [8]. An heterogeneous platform may have one or more accelerator devices of the same or different types.

Many libraries and development frameworks were already developed for these new heterogeneous platforms. They range from frameworks to abstract the inherent complexity of these systems, such as OpenACC [9] or GAMA [10], to specialized high performance libraries for some specific scientific domains, such as CuBLAS [11].

A more in-depth analysis of these two groups of state of the art technology (hardware and software) will be presented in the next sections.

3.1 Hardware Environment

Different accelerator devices opt to use distinct approaches to solve their domain specific problems, leading to small, but important, architectural differences. If these details are not taken into account, it may not be possible to make efficient code, underusing the specialized resources of these devices.

The Single Instruction Multiple Data (SIMD) parallelism model is common ground for most accelerator devices architectures. It is designed to get the most throughput when processing information by applying the same instruction, in parallel, to large sets of independent data. SIMD processors originated in 1970, designated as vector supercomputers, and was popularized by Cray. However, these supercomputers were too specialized for most common processing needs. The current CPUs are including some of the characteristics of vector processors, such as the creation of SSE, and later AVX, instructions from Intel. These extensions to their x86 instruction set, and new special registers capable of holding up to 256 bits (AVX), are extremelly useful for processing highly data parallel algorithms. Current accelerator devices adopted this model in their architectures, as it fits to the needs of the specific problems that they were developed for, such as the GPUs.

Consider GPUs and image processing as an example to justify the use of the SIMD model. Each pixel that is rendered is independent from all other pixels on the image. Their computation result from the same instructions but on different independent data, thus making their processing embarrassingly parallel. For achieving maximum performance, one important characteristic of the code, common to most accelerator device architectures, is that it needs to explore the most parallelism possible between the data to be processed, also known as data parallelism. Other device specific properties, with interest for the programmer, will be discussed later.

Load balancing is always a challenge when programming for parallel environments. Even when using only multicore CPUs, it is important to manage how much load each core is processing, so that every core is working most of the time. If the workload is badly distributed, there will be cores stalled waiting for others to complete, wasting the available computational resources. However, it also depends on the nature of the problem; regular problems are easier to balance than irregular problems, which usually require a dynamic load balancing strategy at runtime, since the execution time of the parallel tasks is not predictable. The same concept also aplies to the dataset: it has to be properly distributed among the available resources in such a way that the CPU and devices take the roughly the same time to process.

Heterogeneous architectures open the possibility of running parallel tasks on both CPU and accelerators simultaneously. However, due to their technical differences, the same task can take different amount of time to complete depending in where it is executed. This creates another layer of complexity when dealing with the workload balance. Now, while managing the workload distribution inside the CPU (between its cores), and also inside the accelerator device chip (with a mechanism similar to the CPU), it is also relevant to manage the distribution between CPU and accelerator device. It is important to have a good control over the load balancing, specially in these hybrid systems, in such a way that neither of the processing units (CPU and accelerator devices) becomes stalled waiting for the other to complete, and thus not wasting any computational resources. Also, the load balancing must be done with the least communication possible between CPU and accelerator, since the time it takes to complete is usually high.

As of 2012, over 50 of the TOP500's list [12] are powered by GPUs, which indicates an exponential growth in usage when compared to previous years. The Intel Xeon Phi is also becoming popular, being the accelerator device of choice in 7 clusters of the TOP500.

3.1.1 The GPU as Computing Accelerator

There are several accelerator devices currently arriving, or already, on the market. The first and most common are General Purpose Graphics Processing Units (GPGPUs). Recently, GPGPU makers allowed drivers to execute code that is not produced for image rendering. However, there are specific hardware details that were designed only for image rendering purposes, which limit the utilization of these devices for certain types of algorithms. One example was the use of only single precision float point arithmetic in the early GPGPUs design.

This type of devices are specialized for massive data parallelism, where the same instruction is simultaneously applied to large amounts of data. One example of a problem domain that can take advantage of these characteristics is the multiplication of matrices, which is very common in scientific applications. As GPGPUs evolved, the support for specific scientific demands was added, such as support for double precision float point arithmetic and compliance to all IEEE float point arithmetic rules.

Recently, NVidia [13] launched a line of GPUs designed for scientific computation rather than image processing [14]. This category of devices, known as the Tesla, has more GDDR RAM, processing units and a slight different design suitable for use in cluster computational nodes (in terms of size and cooling). The chip has suffered some changes too, increasing the cache and the amount of processing units. In this dissertation two different NVidia GPUs will be used, the NVidia Tesla C2070 (Fermi architecture [15]) and the new NVidia Tesla based on the GK110 chip (Kepler architecture [16]).

The NVidia GPUs architecture has two main components: computing units (Streaming Multiprocessors, known as SM) and the memory hierarchy (global external memory, GDDR5 RAM, and 2-level cache and shared memory block). Each SM contains a set of CUDA cores, which are processing units that perform both integer and float point arithmetic (additions, multiplications and divisions). These SMs also have some specialized processing units for only square roots, sins and cosines, as well as a warp scheduler (warps will be explained later) to match CUDA threads to CUDA cores, load and store units, register files and the L1 cache/shared memory.

NVidia considers that a parallel task is constituted by a set of CUDA threads, which will execute the same instructions (conditional jumps are a special case that will be explained next) but on different data. This set of instructions is considered a CUDA kernel, in which the programmer defines the behavior of the CUDA threads. A simple way to visualize this concept is by considering the example of multiplying a scalar with a matrix. In this case, a single thread will handle the multiplication of the scalar by an element of the matrix, and it is needed to use as many CUDA threads as matrix elements.

The CUDA threads are organized in a hierarchy. A block is a set of CUDA threads that is matched by the global scheduler to run on a specific SM. A grid is a set of blocks, representing the whole parallel task. Considering the scalar-matrix multiplication example, each CUDA thread calculates the value of an element of the matrix, and they are organized in blocks, which can represent all the calculations for a single line of the matrix. The grid holds all the blocks responsible for calculating all the new values of the matrix. Note that both the block and the grid have a limited size.

A warp is a set of CUDA threads (usually the same amount as the number of the CUDA cores available in a SM), scheduled by the SM scheduler to run on its SM at a given time. A warp can only be constituted by CUDA threads from the same block.

When programming these devices, conditional jumps must be avoided at all costs if they lead to divergence of the CUDA threads within the same warp. Within an SM it is not possible to have 2 threads executing different instructions at the same time. So, if there is a divergence between the threads within the warp, the two conditional branches will be executed one after the other, doubling the warp execution time. The high latency of accessing the GPU memory os another important detail to take into account. To hide this latency, and avoid the GPU of becoming stalled, the strategy behind the NVidia architectures is to provide the GPU with the most number of CUDA threads possible. This allows the schedulers to keep a scoreboard of which warps are ready to execute and which are waiting for data to load, hopefully never starving the CUDA cores.

Since the GPU is connected by PCI-Express interface, the bandwidth for communications between CPU and GPU is restricted to only 12 GB/s (6 GB in each direction of the communication). Memory transfers between the CPU and GPU must be minimal as it greatly restricts the performance.

Architecture specific details, relevant to the programmer, of both Fermi and Kepler will be presented in the next subsections.

NVidia Fermi Architecture

The relevant architectural details of this architecture, specifically for the Tesla C2070, are explained in this section. The Fermi architecture is schematized in figure 3.1.

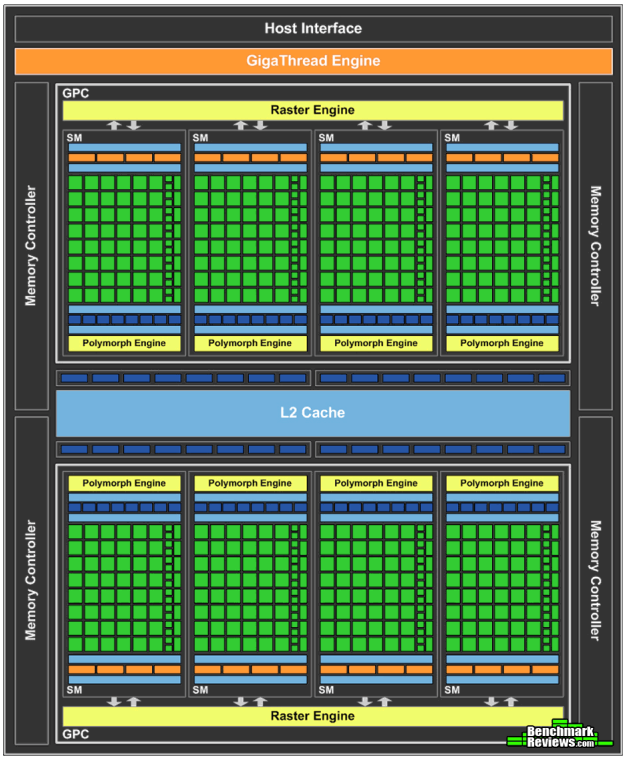


Figure 3.1: Schematic representation of the NVidia Fermi architecure.

In the Tesla C2070, each SM has 32 CUDA cores, with 14 SM per chip, making a total of 448 CUDA cores. Theoretically, it is possible to have 448 CUDA threads running at the same time. In each SM there are 4 Special Functional Units (SFU) to process special operations such as square roots and trignometric arithmetic.

Memory wise, these devices have a slightly different memory hierarchy than the CPUs, but still with the faster and smaller memory closer to the CUDA cores. Each CUDA thread can have up to 63 registers, but when large amounts of threads are used this amount diminishes, which can, in some cases, lead to register spilling (when there is not enough registers to hold the variables values and they must be stored in local memory, which is the external RAM).

Within a SM there is a block of configurable 64 KB memory. In this architecture it is possible to use it as 16 KB for L1 cache and 48 KB for shared memory, or vice-versa (only shared between threads of the same block). The best configuration is dependent of the specific characteristics of each algorithm, and usually requires some preliminary tests to evaluate which configuration obtains the best performance. Shared memory can also be used to hold common resources to the threads, even if they are read-only, avoiding accesses to the slower global memory.

The L2 cache is slower but larger, with the size of 768 KB. It is shared among all SMs, opposed to the L1 cache. The global memory is the last level of on device memory. The Tesla C2070 has a total of 6

GB GDDR5 RAM, with a bandwidth of 192.4 GB/s.

One important detail for efficient memory usage is to perform coalesced memory accesses. Since the load units get memory in blocks of 128 bits, it is possible to reduce the amount of loads by guaranteeing that all threads that need to load data, preferably if it is continuous on the address space (such as contiguous elements of an array), do it at the same time. This allows the memory controller to find the best grouping of thread loads and consolidates them in fewer memory accesses [15].

Finally, on the Fermi architecture it is only possible to run one kernel at a time on the GPU.

NVidia Kepler Architecture

The Kepler and Fermi architectures have many similarities so only the relevant differentiating aspects will be presented.

The Streaming Multiprocessor present in the Fermi architecture was changed to hold more, but smaller, CUDA cores (now 192), working at half the speed of the previous CUDA cores, since the shader clock was removed. The individual performance of the CUDA cores is lower but the overall performance of the GPU has increased. These new Streaming Multiprocessors are now known as SMX and it is possible to have up to 2880 CUDA cores in only one chip. The schematic representation of the Kepler architecture is presented in figure 3.2.

The maximum amount of registers per CUDA thread was increased from 63 to 255. A new read-only cache of 48 KB was added at the same hierarchy level of the L1 cache. The size of the L1/shared memory block is the same as Fermi, but adds a new configuration of 32/32 KB for each type. The L2 cache size has increased to 1536 KB, and its hit bandwidth is 73% larger than on Fermi.

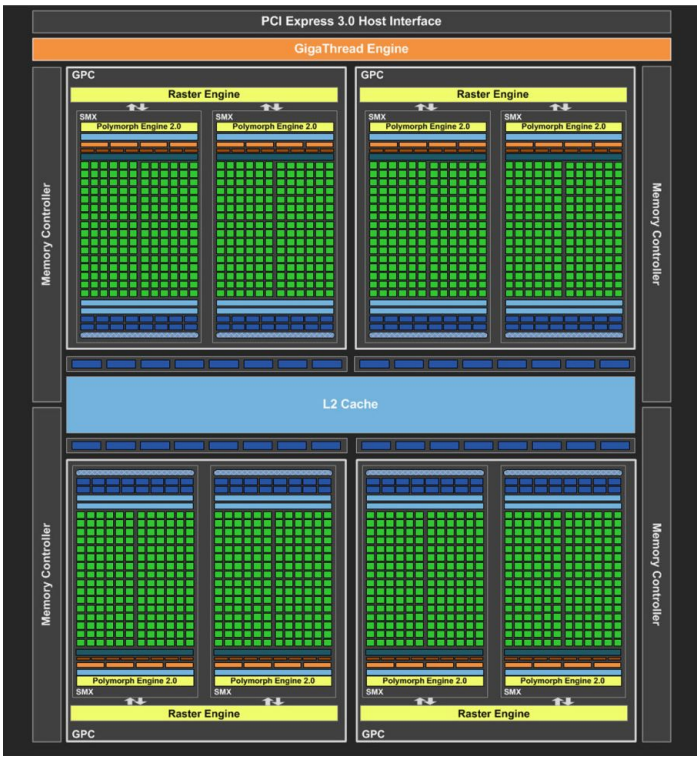


Figure 3.2: Schematic representation of the NVidia Kepler architecture.

Programming-wise, a set of new important features has been added to this architecture. One of them

is the Dynamic Parallelism. Now it is possible to CUDA threads spawn other threads, without it being explicitly required by the host (CPU). It allows for improvements in irregular problems, such as Monte Carlo ray tracing. Another feature is the Hyper-Q, which allows multiple cores of the same CPU to use and spawn kernels on the same GPU. Also, it is now possible to run several different kernels in the same GPU at the same time, where they will be scheduled to different SMX. Finally, a new shuffle instruction has been added to the instruction set. By using this instruction, CUDA threads can now read values directly from each other, within the same warp, without the need of using shared memory.

The problem of coalesced memory accesses is still present. However, while Fermi load units get blocks of 128 bits, load units on the Kepler architecture are capable of getting blocks of 256 bits [16].

3.1.2 The Intel Many Integrated Core architecture

The Intel Many Integrated Core (MIC architecture), currently known as Intel Xeon Phi, has a different conceptual design than the Nvidia GPUs. A chip can have up to 61 multithread cores, with 4 threads per core, and it is more focused on vectorization [17].

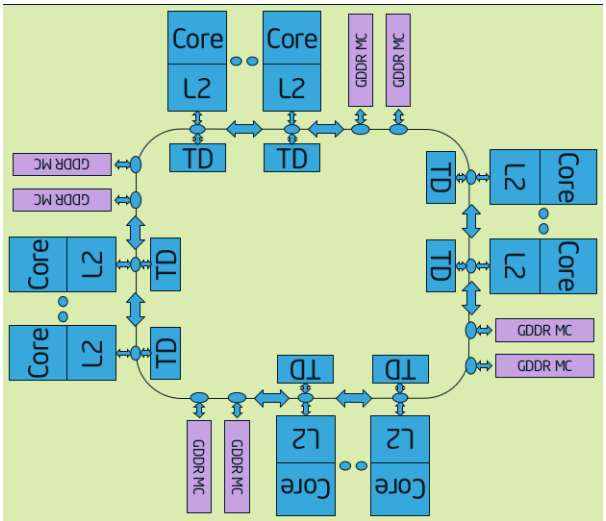


Figure 3.3: Schematic representation of the Intel MIC architecture.

It has 32 512 bit wide vector registers per core, with the capacity of holding 16 single precision float point values. The L2 cache size is 512 KB per core and the chip comes with 6 to 8 GB of GDDR5 RAM, providing up to 320 GB/s of throughput. It was designed for memory bound problems, as opposed to GPUs (Fermi only has a bandwidth of 192.4 GB/s), but Intel will also launch a different version of the chip tuned for compute bound problems.

Unlike conventional CPUs, the MIC cores do not share any cache, therefore cache consistency and coherence is not assured. If needed, data must be explicitly passed between cores, as in a distributed memory system. The cores are connected in a ring network, as represented in figure 3.3.

The MIC uses the same instruction set as conventional x86 CPUs. This allows to easily port current libraries to run on this device. Furthermore, Intel has already announced that a tuned MPI library will be available for this device.

3.2 Development Frameworks

Application development for homogeneous systems with multicore CPUs has been around for some time. There are some libraries that attempt to abstract the programmer from specific architectural and implementation details, providing an easy API as close as possible to current sequential programming paradigms.

Developing applications for heterogeneous systems, with both CPU and accelerator devices, poses a series of new challenges due to the change of programming paradigm. However, some frameworks attempt to abstract the inherent complexity of these platforms.

Frameworks that attempt to ease the programmer's job, while providing scalable and flexible solutions will be presented through the next subsections. Other frameworks that will not be used are not presented.

3.2.1 OpenMP

For shared memory systems, where there is one or more multicore CPUs sharing the same memory address space, one of the most popular libraries for task parallelization is OpenMP [18]. This API is designed for multi-platform shared memory parallel programming in C, C++ and Fortran, on all available CPU architectures. It is portable and scalable, aiming to provide a simple and flexible interface for developing parallel applications, even for the most inexperienced programmers.

While being simple to use, OpenMP allows experienced users to fine-tune the code, providing various task schedulers, as well as instructions for controlling more efficiently the shared memory accesses and parallel execution of the tasks.

3.2.2 OpenACC

OpenACC [9] is a framework for heterogeneous platforms with accelerator devices. It is designed to simplify the programing paradigm for CPU/GPU systems by abstracting the memory management, kernel creation and GPU management. Like OpenMP, it is designed for C, C++ and Fortran, but allowing the parallel task to run on both CPU and GPU at the same time.

While it was originally designed only for CPU/GPU systems, they are currently working on the support for the new Intel Xeon Phi [19]. Also, they are working alongside with the members of OpenMP to create a new specification supporting accelerator devices in future OpenMP releases [20].

3.2.3 GAMA

The GAMA framework [10] has the same purpose of OpenACC, of providing the tools to help building efficient and scalable applications for heterogeneous platforms, but opts for a different strategy. It aims to create an abstraction layer between the architectural details of heterogeneous platforms and the programmer, aiding the development of portable and scalable parallel applications. However, unlike OpenACC, its main focus is on obtaining the best performance possible, rather than abstracting the architecture from the programmer. The programmer to still needs to have some knowledge of each different architecture, and it is necessary to instruct the framework about how tasks should be divided, in order to fit the requirements of the different devices.

The framework frees the programmer from managing the workload distribution (apart from the dataset division), memory usage and data transfers between the available devices. However, it is possible for the programmer to tune these specific details, if he is confortable enough with the framework.

GAMA assumes a hierarchy composed of multiple devices (both CPUs and GPUs, in its terminology), where each device has access to a private address space (shared within that device), and a distributed memory system between devices. To abstract this distributed memory model, the framework offers a global address space. However, since the communication between different devices is expensive, GAMA uses a relaxed memory consistency model, where the programmer can use a synchronization primitve to enforce memory consistency.

3.2.4 Debugging

Debugging applications in shared memory systems is a complex task, as the errors are usually harder to replicate than on sequential applications. Bugs can happen due to deadlocks, unexpected changes to the shared memory, data inconsistency and incoherence. While there are some tools to efficiently debug sequential applications, such as the GNU Debugger [21], they lack on the support for multithreaded applications. Unfortunately, there are no debuggers that can efficiently be used to debug a parallel application.

The effort necessary to debug these applications, without the use of any third-party tools, is directly related to the programmers experience and knowledge of working with shared memory systems. However, even the most experienced will face complex obstacles when debugging for more than 4 threads, as the application behavior is much harder to control.

Nvidia offers a tool for debugging CUDA kernels on their GPUs, which is based on the GNU Debugger [22]. It is useful when used to find bugs in the kernels, but only in the same way that a sequential application is debugged. Also, when using more than 2-4 CUDA threads it does not help the programmer at all, considering that CUDA kernels can reach to the thousands of threads.

4 Case study: the `ttH_dilep` analysis application

The computing resources related to all CERN projects are organized in a tier hierarchy. The first is the Tier-0 computing clusters at CERN, which from there the data is distributed to the 10 Tier-1 data centers, spread among different countries. These Tier-1 data centers are used for central processing, reconstruction of events data and Monte Carlo simulations. Tier-2 sites are dedicated to further processing and reconstruction of data and Monte Carlo events, while Tier-3 sites are used to perform data analysis and simulation [3].

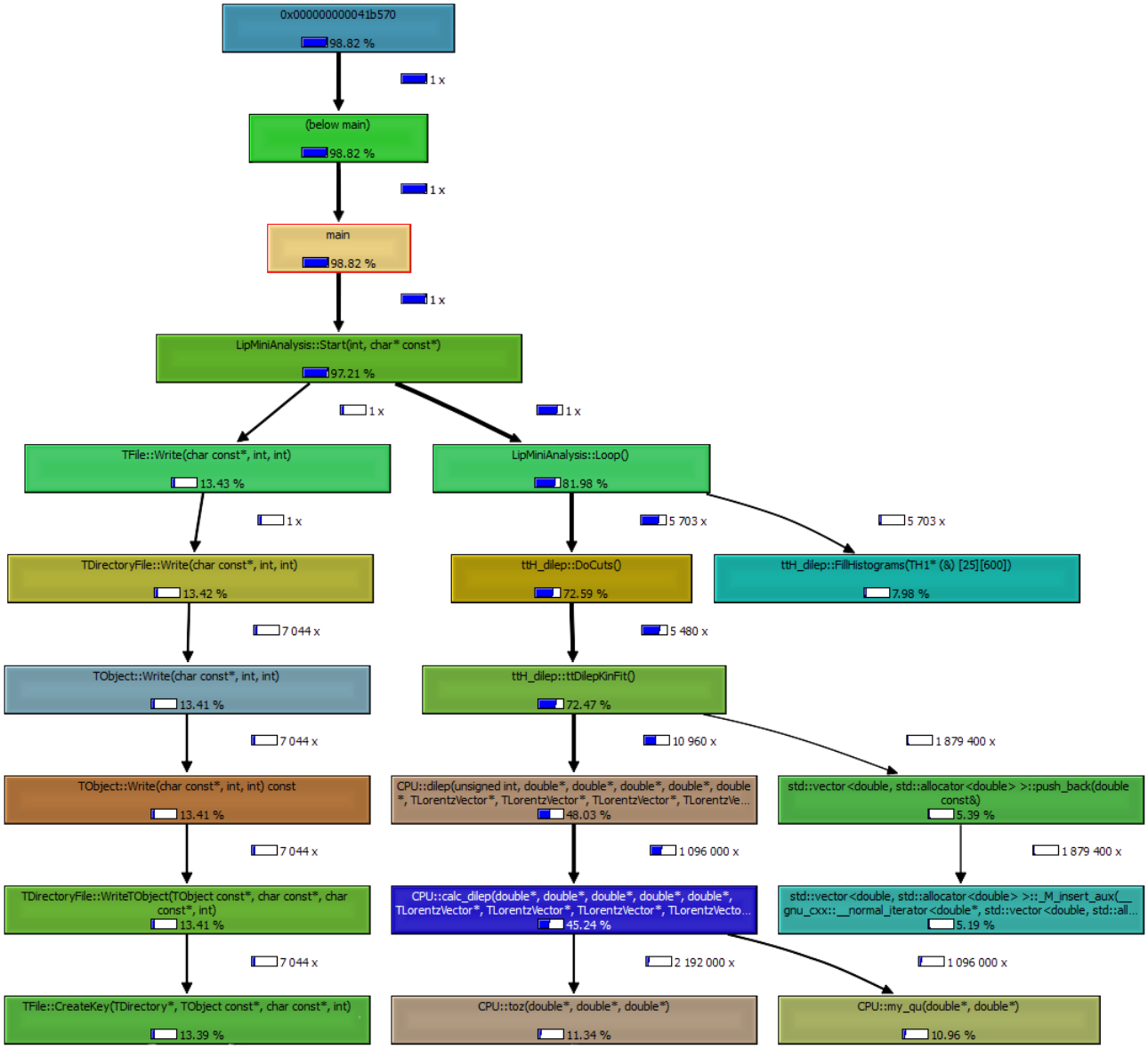


Figure 4.1: Callgraph generated using the Valgrind tool [23] for the `ttH_dilep` analysis with 100 `dilep` executions per event.

It is in this Tier-3 that the `ttH_dilep` analysis application fits. It was developed by the LIP researchers to solve the problem presented in the section 2. The application has two main dependencies: the ROOT framework [24] and the LipMiniAnalysis library.

The ROOT framework is being developed by CERN and provides a set of functionalities needed to handle and analyze large amounts of data. They range from data storage, in the standard formats used by CERN, to histograming, curve fitting minimization and visualization methods. It aims to provide the programmer a set of tools that will ease the development of their analysis code.

The LipMiniAnalysis is a library developed by LIP, containing a set of methods and functionalities useful for the analysis that they conduct with the ATLAS detector data. It is also prepared to read a more refined set of data resultant from the data that arrives at the Tier-3.

As illustrated by the callgraph of the analysis application in figure 4.1, the main flow of the application is controlled by the `Loop` method. This method will apply the DoCuts function to every event to process. The event passes a series of tests and evaluations (cuts). If an event reaches the cut 20, of a total of 21, the `ttDilepKinFit` function is called. It is in this function that the `ttbar` and Higgs reconstructions are performed. In the beginning of the `ttDilepKinFit` method, the available jets are combined two by two, as well as the leptons, as explained in section 2.

The `dilep` function, called within the `ttDilepKinFit` method, analytically determines the neutrinos characteristics for each jet/lepton combination, reconstructing the `ttbar` system. The function is very compute intensive, as it analytically solves a system of 6 complex equations. Based on the equations result, it can produce two to four possible solutions (resulting particles). While it is compute bound, memory transfers of the required data for each time an event is processed can be a limiting factor when using accelerators. It may only become viable if the amount of `dilep` executions within an event is capable of hiding the memory transfer latency. These results are used in the remaining of the `ttDilepKinFit` to estimate the probability of the reconstruction, as well as reconstruct the Higgs boson. The final probability of the reconstruction is determined by combining the probability of the `ttbar` reconstruction with the computed probability of the Higgs reconstruction.

As seen from the callgraph, most of the application execution time is spent in the `ttDilepKinFit` method, so it is the section of the application where most efforts for optimizating the code must be focused. The rest of the functions are mainly auxiliary and I/O methods.

5 Conclusions and Future Work

The workflow of the `ttDilepKinFit` method, explained in section 4, needs to be changed so that it is easier and more effective to parallelize its execution. Currently, the variation is applied for each jet/lepton combination, `dilep` is executed, the results are treated (and the probability of the reconstruction is computed) and the Higgs boson is reconstructed.

The best approach is to create a data set with all the jet/lepton combinations and all the respective variations. Then, execute `dilep` with all the elements on the data set and store all the results with the associated element of the data set. Note that the number of `dilep` executions per event will be equal to the number of jet/lepton combinations times the number of variations per combination. Finally, iterate through all results, reconstruct each Higgs boson and calculate the respective probability for each element of the first data set. Figure 5.1 represents the current and the alternative workflows.

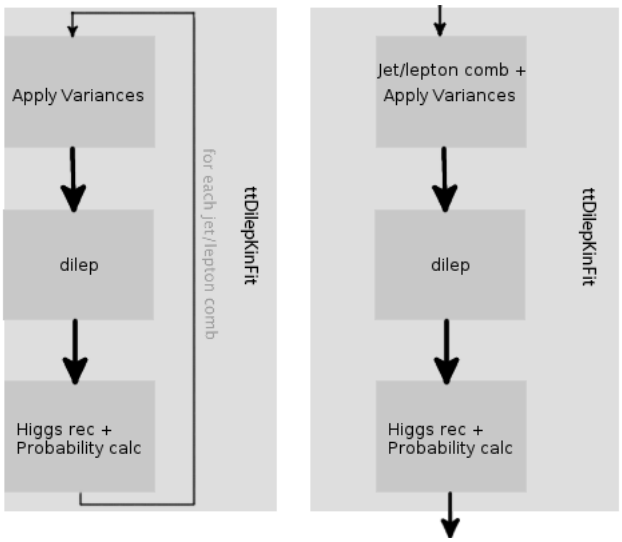


Figure 5.1: Current workflow (left) vs alternative workflow (right) of the `ttDilepKinFit` method.

This approach offers (theoretically) the possibility of having three distinct parallel tasks. The first would be the jet/lepton combination and variance calculations. The second, would be the dilep executions, which are independent, it is only needed to merge the results after. The third would be the final iteration through the results of each dilep execution and respective Higgs reconstruction. However, these three parallel tasks are pipelined with streaming dependencies.

To lower the cost of this dependency, a queue-based approach will be tested. As the first data set is constructed, its elements can be provided to the next parallel task, where `dilep` is executed. As soon as the results from dilep are available they can be passed to the third parallel region and the Higgs boson can be reconstructed. In theory, this will decrease the execution time, relative to a strict implementation, where all the parallel tasks are being executed at the same time after an initial latency.

After this stage, an implementation of the kinematical reconstruction (`dilep`) will be attempted on both GPUs and Xeon Phi. Note that `dilep` is the most time consuming task in the `Loop` method, and it tends to increase even more with the specified number of variances. The efforts will be towards obtaining the optimal hybrid implementation possible (i.e., also using the CPU). The performance will be measured and compared between these devices and identify the bottlenecks.

Finally, an implementation using the OpenACC and GAMA frameworks will be tested, relatively to

the previous optimized implementations, in terms of performance but also considering the development time and usability of these tools.

References

[1] European Organization for Nuclear Research. *The Large Hadron Collider*. 2012. URL: http://public.web.cern.ch/public/en/lhc/lhc-en.html (cit. on p. 5).

[2] European Organization for Nuclear Research. *CERN European Organization for Nuclear Research*. 2012. URL: http://public.web.cern.ch/public/ (cit. on p. 5).

[3] V. Oliveira et al. "Even Bigger Data: Preparing for the LHC/ATLAS Upgrade". In: *6th Iberian Grid Infrastructure Conference* (2012) (cit. on pp. 5, 16).

[4] C. Biscarat G. Brandt G. Duckeck P. van Gemmeren A. Peters RD. Schaffer W. Bhimji and I. Vukotic. *IO performance of ATLAS data formats*. 2010. URL: http://cdsweb.cern.ch/record/1299564/files/ATL-SOFT-SLIDE-2010-381.pdf?version=1 (cit. on p. 5).

[5] European Organization for Nuclear Research. *ATLAS experiment*. 2012. URL: http://atlas.ch/ (cit. on p. 5).

[6] Laboratório de Experimentação e Física Experimental de Partículas. *Laboratório de Experimentação e Física Experimental de Partículas*. 2012. URL: http://www.lip.pt/ (cit. on p. 5).

[7] Intel. *The Intel® Xeon PhiTM Coprocessor 5110P*. Tech. rep. 2012 (cit. on p. 8).

[8] Texas Instruments. *Digital Signal Processors*. 2012. URL: http://www.ti.com/lsds/ti/dsp/overview.page (cit. on p. 8).

[9] OpenACC Corporation. *OpenACC*. 2012. URL: http://www.openacc-standard.org/ (cit. on pp. 8, 14).

[10] João Barbosa. *GAMA framework: Hardware Aware Scheduling in Heterogeneous Environments*. Tech. rep. 2012 (cit. on pp. 8, 14).

[11] NVIDIA Corporation. *CUBLAS*. 2012. URL: https://developer.nvidia.com/cublas (cit. on p. 8).

[12] TOP 500. *November 2012*. 2013. URL: http://top500.org/lists/2012/11/ (cit. on p. 9).

[13] NVIDIA Corporation. *NVIDIA*. 2012. URL: http://www.nvidia.com/page/home.html (cit. on p. 10).

[14] NVIDIA Corporation. *High Performance Computing: Accelerating Science with Tesla GPUs*. 2013. URL: http://www.nvidia.com/object/tesla-supercomputing-solutions.html (cit. on p. 10).

[15] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architectur: Fermi*. Tech. rep. 2009 (cit. on pp. 10, 12).

[16] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architectur: Kepler GK110*. Tech. rep. 2012 (cit. on pp. 10, 13).

[17] Intel. *Intel Many Integrated Core Architecture*. Tech. rep. 2010 (cit. on p. 13).

[18] OpenACC Corporation. *OpenACC: Directives for Accelerators*. 2013. URL: http://openmp.org/wp/ (cit. on p. 14).

[19] HPC Wire. *OpenACC Group Reports Expanding Support for Accelerator Programming Standard*. 2013. URL: http://www.hpcwire.com/hpcwire/2012-06-20/openacc_group_reports_expanding_support_for_accelerator_programming_standard.html (cit. on p. 14).

[20] OpenACC Corporation. *How does the OpenACC API relate to OpenMP API*. 2013. URL: http://www.openacc-standard.org/node/49 (cit. on p. 14).

[21] Free Software Foundation. *GDB: The GNU Project Debugger*. 2013. URL: http://www.gnu.org/software/gdb/ (cit. on p. 15).

[22] NVIDIA. *CUDA-GDB: The NVIDIA CUDA Debugger User Manual*. Tech. rep. 2008 (cit. on p. 15).

[23] Valgrind Developers. *Callgrind: a call-graph generating cache and branch prediction profiler*. 2013. URL: http://valgrind.org/docs/manual/cl-manual.html (cit. on p. 16).

[24] B. Bellenot O. Couet A. Naumann G. Ganis L. Moneta V. Vasilev A. Gheata P. Russo F. Rademakers P. Canal and R. Brun. *ROOT*. 2012. URL: http://root.cern.ch/drupal/ (cit. on p. 16).