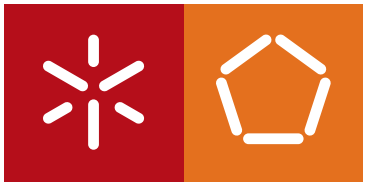
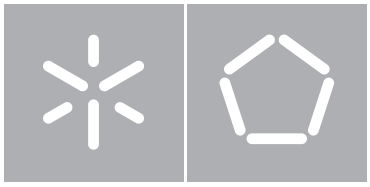


Universidade do Minho
Escola de Engenharia

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

Resumir isto, tuga style

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 computing heterogeneous 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, to tune the kinematical reconstruction, as well as restructure and parallelize other critical areas of this analysis specific code.

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. It will be compared against a similar concurrent framework, OpenACC, in terms of performance, development time and usability.

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

Accelerating 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 common GPUs, as opposed to scientific computing oriented GPUs

DSP Digital Signal Processor

MIC Many Integrated Core, accelerating device developed by Intel®, currently known as Xeon Phi

Homogeneous system Classic computer system, which may contain a multicore CPU

Heterogeneous system Computer system, which may contain a multicore CPU and one or more accelerating devices

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

SM/SMX Streaming Multiprocessor, 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

CUDA formerly Compute Unified Device Architecture. A parallel computing platform for NVIDIA® GPUs

OpenMP Open Multi-Processing

OpenACC A programming standard for parallel computing

GAMA Coisas

List of Figures

1 Introduction

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

At the LHC, an experiment usually consists of a head-on collision of particles (which is considered an event), where detectors gather data about the collision. There are different detectors with different purposes according to the experiments that they were built for, but 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, where the data is refined and scattered to among the many research groups until it is ready to be used in simulations, where it is analyzed [4].

ATLAS [5] is one of the main experiments being conducted at the LHC. One of the research groups involved in support and analysis of the data of this experiment is the Laboratório de Instrumentação e Física Experimental de Partículas (LIP) [6]. LIP continuously performs analysis on the data gathered by the ATLAS detector, competing against other research groups from the same experiment in order to analyze the most data and be the first to publish relevant results.

The focus of this dissertation work 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, which is very important for their research and, consequently, is widely used within the group. Using a case study will lead to analyze and improve the performance of other application specific tasks, in order to (i) get the maximum performance from the tuned kinematical reconstruction, and (ii) improve the overall performance of the analysis.

The tuning of both the kinematical reconstruction and overall application performance will be performed on heterogeneous architectures, which combine traditional all-around multicore processors with accelerating devices in the same system. Porting code that was original designed for sequential execution to these heterogeneous environments faces a series of problems, such as different architectural and programming paradigms, tuning code for specific devices, which requires deep architectural knowledge of the device, and load balancing of the tasks between CPU and accelerators. This will be explained in-depth in section 3.

To ease this burden on the programmers there are several frameworks that try 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 optimized implementations that will also be developed, in terms of performance, usability and development time.

2 Contextualization

The LHC accelerates two particle beams in opposite directions, where they collide at the detectors. From this head-on collision between two particles results a limited chain reaction of decaying particles, where the detector records most of the final particles. The ATLAS detector can record only some of the characteristics of these particles. A schematic representation of the head-on collision is presented in figure 2.1, and it is known as the ttbar system. The particles detected 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. In order to reconstruct the collision, the characteristics of the 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 calculate the degree of certainty associated with the reconstruction.

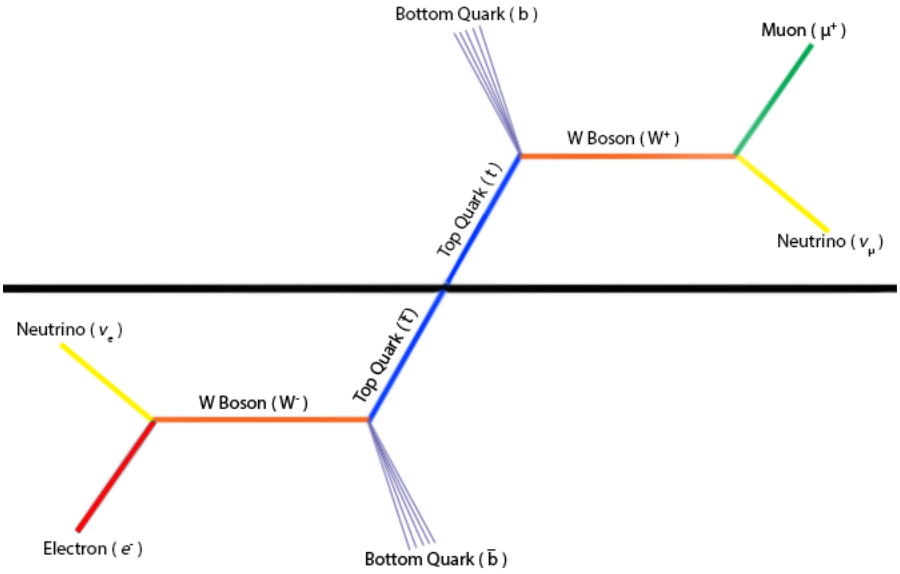


Figure 2.1: Schematic representation of the ttbar system.

During a collision, several particles, bottom quark jets and leptons, are detected. The amount detected can vary from between events, but it is needed at least 2 jets and 2 leptons to reconstruct the ttbar system, as represented in the figure 2.1, but can reach up to 14. However, some of the jets/leptons may not belong to the ttbar system, so it is needed to choose the ones that reconstruct the system with the most accuracy.

By performing the kinematical reconstruction to each combination of all the bottom quark jets and leptons, two by two, and calculating the probability associated with the 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) characteristics are not 100% accurate. In fact, the measurements made by ATLAS can have a 2% inaccuracy to the real values. Since these particles are used in the kinematical reconstruction, its accuracy can be affected. To improve the accuracy 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 in order to get

higher probability of finding a great reconstruction. This means running the kinematical reconstruction as many times as possible, per event, with inputs with different variations.

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

In the ttH_dilep analysis, the importance of the kinematical reconstruction (dilep) is even greater. This analysis takes into account the two jets that result from the Higgs boson decay and also tries to reconstruct it. Figure 2.2 schematically represents the ttbar system with the Higgs boson decay and respective 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 badly reconstructed, the Higgs boson reconstruction will not be accurate. Now, the best final reconstruction is given by the probability of the best kinematical reconstruction and the probability of the respective Higgs boson reconstruction.

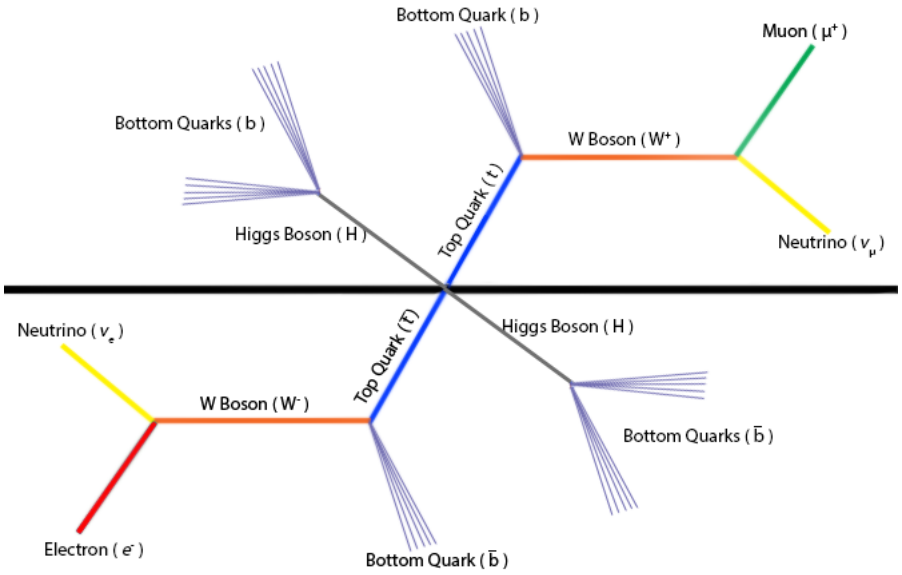


Figure 2.2: Schematic representation of the ttbar system with the Higgs boson decay.

By increasing the kinematical reconstruction performance it is possible to perform 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 the jet combination, variance appliance and Higgs reconstruction as other important tasks to improve, and eventually re-design the workflow of this section of the application. The LIP research group has a big interest on improving the kinematical reconstruction, as well as the overall ttH_dilep analysis, performance, as it would give them an advantage over the other research groups.

3 State of the Art

Most of today's programmers produce code and design applications using sequential programming paradigms. The application behavior is designed and tested only for sequential execution, where the only parallelism is made by the compiler at the instruction level. A few years ago a transition from single core very fast CPUs to slightly slower multicore CPUs started to happen. Unfortunately, these newer CPUs need a different programming paradigm to get the most performance possible when designing an application; however, programmers did not accompany this transition.

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 programming paradigms. 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 accelerating 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 and expect 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, focused on achieving the most performance possible on specific problem domains, opposed to common all-around CPUs. Usually, they are oriented for massive data parallelism processing (SIMD architectures), offloading the CPU from such data intensive operations. Several many-core accelerating devices are available, ranging from the general purpose GPUs, the Intel® Many Integrated Core line, currently known as Intel® Xeon Phi [7], and Digital Signal Processors [8]. A heterogeneous system may have one or more accelerating devices of the same or different types.

Many libraries and frameworks were already developed with these new heterogeneous platforms in mind. They range from frameworks to abstract the inherent complexity of these systems, such as OpenACC [15] or GAMA [15.1], to specialized high performance libraries for some specific scientific domains, such as CuBLAS [25].

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

3.1 Hardware

While having the same (conceptual) purpose, different accelerating devices opt to use different approaches to solve their domain specific problems, leading to small, but important, architectural differences. If these details are not taken into account, it is impossible to make efficient code, underusing the specialized resources of these devices.

The Single Instruction Multiple Data parallelism model is common ground for most accelerating 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. Considering the GPUs as an example, each pixel

that must be rendered is independent from all other pixels, but the same instructions are executed, thus making their processing embarrassingly parallel. For achieving maximum performance, one important characteristic of the code is that it needs to take advantage of the most parallelism possible between the data to be processed. Other device specific properties, with interest for the programmer, will be discussed later.

These 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 raises the issue of managing the amount of work to be processed on the CPU and on the accelerating device, in such a way that neither of them becomes stalled waiting for the other to complete, and thus not wasting computational resources. This issue is commonly known as load balancing. However, it can also depend 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 their execution time is not predictable. Load balancing is one of the most important issues to deal when tuning a parallel algorithm for hybrid systems.

3.1.1 Graphics Processing Unit

There are several accelerating devices currently arriving, or already, on the market. The first, and most common, are General Purpose Graphics Processing Unit (GPGPU). Recently, GPGPU makers allowed drivers to execute code that is not related to 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.

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

More recently, NVIDIA® [10] launched a line of GPUs designed for scientific computation rather than image processing. 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). In this dissertation two different NVIDIA® GPUs will be used, the NVIDIA® Tesla C2070 (Fermi architecture [11]) and the NVIDIA® Tesla **Nome kepler** (Kepler architecture [12]).

NVIDIA® GPUs architecture has two main components: Streaming Multiprocessors (SM) and GDDR5 ram. 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 later be explained), which match CUDA™ threads to CUDA™ cores, load and store units, register files and a 2 level cache.

NVIDIA® considers that a parallel task is represented by a set of CUDA™ threads, which will execute the same instructions (however, conditional jumps are a special case that will be explained next) but on different data. A simple way to visualize this concept is by considering the problem of multiplying a scalar with a matrix as an example. 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.

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 represent all the calculations of 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 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.

When programming for these devices, conditional jumps must be avoided at all costs. 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 same warp, the two conditional branches will be executed sequentially, doubling the warp execution time.

Since the GPU is connected by PCI-Express interface, the bandwidth 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 next.

NVIDIA® Fermi Architecture

The relevant architectural details of this architecture, specifically for the Tesla C2070, are explained in this section.

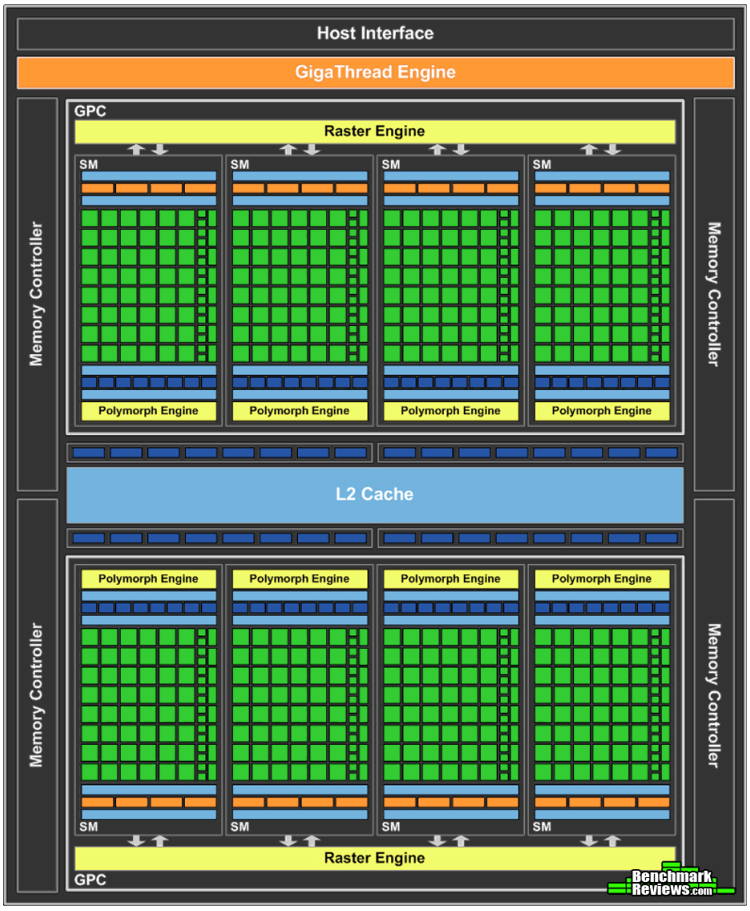


Figure 3.1: Schematic representation of the NVIDIA® Fermi architecure.

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 is 4 Special Functional Units (SFU) to process square roots, sins and cosines.

Memory wise, these devices have a slightly different memory hierarchy than the CPUs, but still with the faster and smaller memory is 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 the cache).

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 (only shared between threads of the same block) or vice versa. 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 the 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 the threads that need to load data, preferably if it is continuous on the address space (such as 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 the fewer memory accesses possible [11].

Finally, on the Fermi architecture it is only possible to run one kernel (piece of CUDA™ code designed to be ran by each CUDA™ thread) 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, and it is now known as SMX. This allows having up to 2880 CUDA™ cores in only one chip.

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.

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 [12].

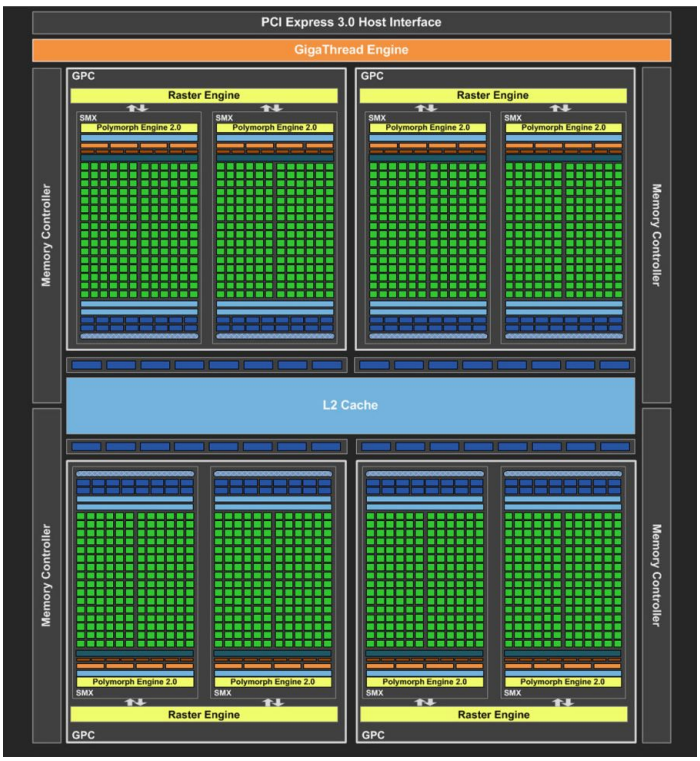


Figure 3.2: Schematic representation of the NVIDIA® Kepler architecture.

3.1.2 Intel® Many Integrated Core

The Intel® Many Integrated Core (MIC), currently known as Intel® Xeon Phi, architecture from Intel, Knights Corner, has a different conceptual design than the Nvidia GPUs. A chip can have up to 61 multithread cores, with 4 threads each, and focus more on vector instructions [13].

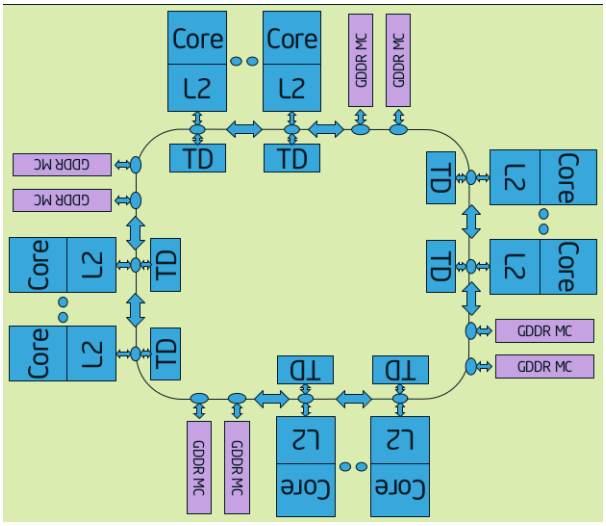


Figure 3.3: Schematic representation of the Intel® MIC architecture.

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, but Intel will also launch a different version of the chip tuned for compute bound problems.

Unlike the 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 MIC uses the same instruction set as common Intel® CPUs (x86). 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 Software

Application development for homogeneous systems with multicore CPUs is not as recent as one may think. 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.

However, developing applications for heterogeneous systems, with both CPU and accelerating devices, poses a series of new challenges due to its different programming paradigm. Even though, there are some frameworks that attempt to abstract the inherent complexity of these systems.

Frameworks that attempt to ease the programmer's job, while providing scalable and flexible solutions, which will be used during the dissertation, will be presented through the next subsections.

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 is OpenMP [14]. 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 fine-tuning of the code for the most experienced programmers, 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 [15] is a framework for heterogeneous systems with accelerating 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 [16]. Also, they are working alongside with the members of OpenMP to create a new specification supporting accelerating devices in future OpenMP releases [17].

3.2.3 GAMA

GAMA stuff.

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 [26], 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 [18]. 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 CERN Tier-0 computing clusters and from there is distributed to the 10 Tier-1 data centers, spread by different countries, which are used for central processing and reconstruction of data events and simulation of Monte Carlo events. 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 [19].

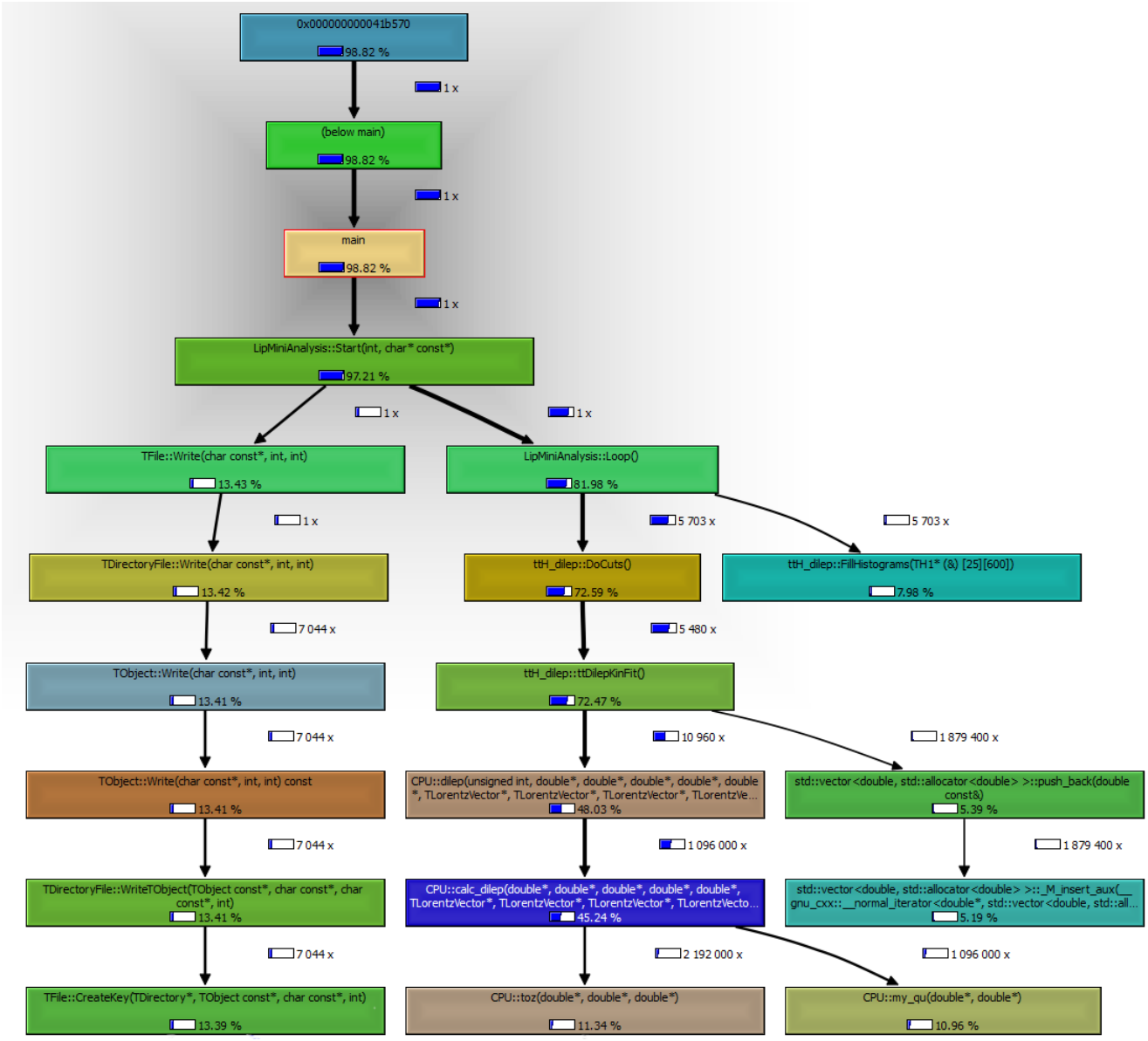


Figure 4.1: Callgraph generated using the Valgrind tool [27] for the ttH_dilep analysis with 100 dilep executions per event.

It is in this Tier-3 that the ttH_dilep analysis application fits. It was developed by the LIP researchers to solve the problem explained in the section 2. The application has two main dependencies: the ROOT framework [20] 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 construction 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 DPD data format that arrives at the Tier-3.

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 after within the ttDilepKinFit method, analytically determines the neutrinos characteristics for each jets and leptons combination, reconstructing the ttbar system. It can produce two to four possible result particles. These results are used in the remaining of the ttDilepKinFit to determine 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 calculated probability of the Higgs reconstruction. As seen from the callgraph, most of the application execution time is spent in the Loop method, so there is the place where most efforts of optimization must be focused. The rest are auxiliary and I/O functions.

5 Conclusions and Future Work

The workflow of the ttDilepKinFit method, explained in the previous section, needs to be changed so that it can be 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 calculated) 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 amount of 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 combination times the number of variations per combination. Finally, iterate through all of the 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 presented alternative workflows.

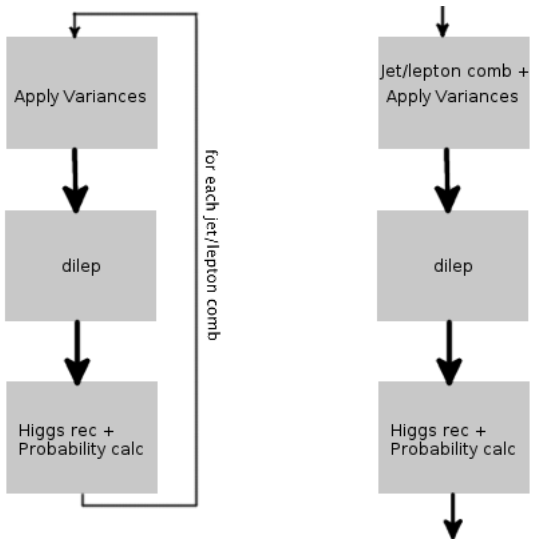


Figure 5.1: Current workflow (left) vs alternative workflow (right) of the ttDilepKinFit method.

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 dependent on each other.

To ease 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. Figure Z1 illustrates the current workflow of ttDilepKinFit and the alterations that will be made.

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

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.