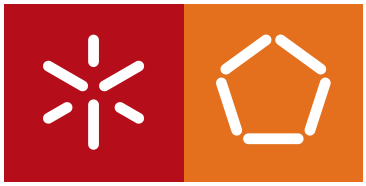
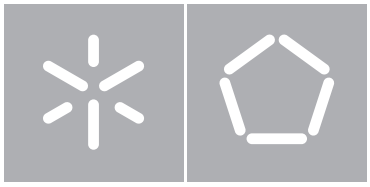


Universidade do Minho
Escola de Engenharia

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

Acknowledgments/Agradecimentos

put your bigass heart out there and don't forget the money you were given

Abstract

Resumo

Contents

Contents

Glossary

Event Head-on collision between two particles at the LHC

Combination A set of two leptons and two jets

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

QPI Quickpath Interconnect, point-to-point interconnection developed by Intel

HT HyperTransport, point-to-point interconnection developed by the HyperTransport Consortium

NUMA Non-Uniform Memory Access, memory design where the access time depends on the location of the memory relative to a processor

ISE Instruction Set Extensions, extensions to the CPU instruction set, usually SIMD

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

Speedup Ratio of the performance increase between two versions of the code. Usually comparing single vs multithreaded applications.

bottleneck

List of Figures

List of Figures

List of Tables

Page

Chapter 1

Introduction

The dissertation is first presented by contextualizing the scientific background of CERN and LIP organizations, as well as their current research projects, which are closely involved in this work. The motivation for the dissertation is presented in section 1.3, with the problem contextualized from a physics perspective in subsection 1.3.1. The Goals, subsection 1.3.2, states the objectives to be achieved by this work, in terms of improving the research and application development quality by implementing a set of solutions for homogeneous and heterogeneous systems, while assessing the efficiency and usability of hardware accelerators in the latter. The scientific contribution of this work is presented in subsection 1.3.3. Subsection 1.4 overviews the structure of this dissertation.

1.1 Context

The European Organization for Nuclear Research [1] (CERN, acronym for *Conseil Européen pour la Recherche Nucléaire*) is a consorcium of 20 european countries, with the purpose of operating the largest particle physics laboratory in the world. Founded in 1954, CERN is located in the border between France and Switzerland, and employs thousands of scientists and engineers representing 608 universities and research groups of 113 different nationalities.

CERN research focus on the basic constituents of matter, which started by studying the atomic nucleus but quickly progressed into high energy physiscs (HEP), namely on the interactions between particles. The instrumentation used in nuclear research is essentially divided into particle accelerators and detectors, alongside with the facilities necessary for delivering the protons to the accelerators. The purpose of the accelerator is to speed up groups of particles close to the speed of light, in opposite directions, resulting in a controlled collision inside the detectors (the collision is called an event). The detectors record various characteristics of the resultant particles, such as energy and momentum, which originate from complex decay processes of the collided protons. The purpose of these experiments is to test and validate specific HEP theories by interpreting the results of the collisions based on the expected theoretical model.

CERN laboratory started with a small low energy particle accelerator, the Proton Synchrotron [2] inaugurated in 1959, but soon its equipment was iteratively upgraded and expanded. The current facilities are constituted by the older accelerators (some already decomissioned) and particle detectors, as well as the newer Large Hadron Collider (LHC) [3] high energy particle accelerator, located 100 meter underground and with a 27 km circuference length. There are currently seven experiments running on the LHC: CMS [4], ATLAS [5], LHCb [6], MoEDAL [7], TOTEM [8], LHC-forward [9] and ALICE [10]. Each of these experiments have their own detector on the LHC and conduct HEP experiments, using of distinct

technologies and research approaches. One of the most popular researches being conducted at CERN is the validation of the Higgs boson theory. During the next year the LHC will be upgraded to increase its luminosity (amount of energy of the accelerated particle beams).

Approximately 600 millions of collisions occur every second at the LHC. Particle detectors react with the particles resultant from the collisions, generating massive amounts of raw data as electric signals. It is estimated that all the detectors combined produce 25 petabytes of data per year [11, 12]. CERN does not have the financial resources to afford the computational power necessary to process all the data, which motivated the creation of the Worldwide LHC Computing Grid [13], a distributed computing infrastructure that uses the resources of scientific community for data processing. The grid is organized in a hierarchy divided in 4 tiers. Each tier is made by one or more computing centers and has a set of specific tasks and services to perform, such as store, filter, refine and analyse all the data gathered at the LHC.

The Tier-0 is the data center located at CERN. It provides 20% of the total grid computing capacity, and its objective is to store and reconstruct the raw data gathered at the detectors in the LHC, converting it into meaningful information, usable by the remaining tiers. The data is received on a format designed for this reconstruction, with information about the event, detector and software diagnostics. The output of the reconstruction has two formats, the Event Summary Data (ESD) and Analysis Object Data (AOD), each with different purposes, containing information of the reconstructed objects and calibration parameters, which can be used for early analysis. This tier distributes the raw data and the reconstructed output by the 11 Tier-1 computational centers, spread among the different countries that are members of CERN.

Tier-1 computational centers are responsible for storing a portion of the raw and reconstructed data and provide support to the grid. In this tier, the reconstructed data suffers more reprocessing, refining and filtering the relevante information and reducing the size of the data, now in Derived Physics Data (DPD) format, then transferred to the Tier-2 computational centers. The size of the data for an event is reduced from 3 MB (raw) to 10 kB (DPD). This tier also stores the output of the simulations performed at Tier-2. The Tier-0 center is connected to the 11 Tier-1 centers by high bandwidth optical fiber links, which form the LHC Optical Private Network.

There are roughly 140 Tier-2 computational centers spread around the world. Their main purpose is to perform Monte-Carlo simulations with the data received from the Tier-1 centers, but also perform a portion of the events reconstructions. The Tier-3 centers range from university clusters to small personnal computers, and they perform most of the events reconstruction and final data analysis. In the CERN terminology, an analysis is the denomination of an application which is designed to process a given amount of data in order to extract physically relevant information about events that may support a specific HEP theory.

1.2 LIP Research Group

The Laboratório de Instrumentação e Física Experimental de Partículas (LIP) [14] is a portuguese scientific and technical association for research on experimental high energy physics and associated instrumentation. LIP has a strong collaboration with CERN as it was the first scientific organization from Portugal that joined, in 1986. It has laboratories in Lisbon, Coimbra and Minho and 170 people employed. LIP researchers have produced several applications for testing various HEP theories of the ATLAS experiment that use Tier-3 computational resources for data analysis. Most of the analysis applications use home-grown skeleton libraries, such as the LipCbrAnalysis and LipMiniAnalysis.

The motivation for this dissertation, presented in section 1.3, results from a close cooperation between the Department of Informatics of the University of Minho and the LIP laboratory in Minho, which began in 2011.

1.3 Motivation, Goals & Scientific Contribution

With an increase of collisions and, consequently, the data being produced by the detectors at the LHC, research groups will need a bigger budget for aquiring and maintaining computational resources to analyze the data and keep up with the deadlines. To add up to the increase in data, research groups working on the same experiment enforce positive competition to be the first to find and publish relevant results. The amount and quality of event processing as a direct impact on the research, meaning that groups with more computational resources ahead of the competition.

Better results are not only obtained by increasing the amount of events analyzed; it is important to take into account the quality of each event analysis. The ATLAS detector has an experimental resolution of 2%, meaning that each measured value for a characteristic of a particle resultant from a collision might not be exact and, therefore, the analysis will have an error associated. It is possible to improve the analysis quality but it will increase its execution time, creating a trade-off between events to analyze and their quality. This issue will be presented in the context of this dissertation with more detail on subsection 1.3.1.

One of the most important analysis being conducted by LIP relates to the Top Quark and the Higgs Boson researches. An application was devised to reconstruct an event obeying the theoretical model of Top Quark decay. It also attempts to reconstruct the Higgs Boson associated with the event. Each event can be reconstructed several times, with some of its parameters varied by a random offset (with a maximum magnitude of 2% of the original value), so that a accurate final reconstruction is obtained by chosing the partial reconstruction that better satisfies the theoretical model. The purpose of this mechanism is to overcome the experimental resolution of the ATLAS detector. The number of reconstructions performed per event directly relates to the application execution time. The theoretical model for this system is presented in subsection 1.3.1 and the analysis application in chapter 3.

While investing in the upgrade of the computational resources of the research group is a valid option to deal with the increase of events to analyze, it is also necessary to take into account if the current resources are being efficiently used by the current applications. Also, hardware is not necessarily getting faster, but wider as the number of cores per chip is increasing rather that its speed (see chapter 2), which can cause big investments to result in small improvements. Current computing clusters are constituted of systems with one or more multicore CPUs (homogeneous systems) and some even utilizing hardware accelerators, very fast and efficient for specific problem domains (heterogeneous systems). It is important to have a knowledge of the newer architectures in order to develop efficient applications that resort to parallelism to efficiently use the system resources. Programming for such architectures (both multicore CPUs and hardware accelerators) requires a set of skills and experience that most physicists (usually self-taught programmers) do not have, developing applications not adequate to take advantage of these architectures.

Increasing the efficiency of an application by resorting to parallelism enables the possibility of performing more reconstructions per event and more events to be processed, while using all the potential of the available computational resources and avoiding needless investments in hardware upgrades.

1.3.1 The Top Quark system and Higgs boson decay

In the LHC, two proton beams are accelerated close to the speed of light in opposite directions, set to collide inside a specific particle detector. From this head-on collision results a chain reaction of decaying particles, and most of the final particles react with the detector for it to record their characteristics. One of the experiments being conducted at the ATLAS detector is related to the discovery of new Top Quark physics. The schematic representation of the Top Quark decay (usually addressed as the $t\bar{t}$ system), resultant from a head-on collision of two protons, is presented in figure 1.1.

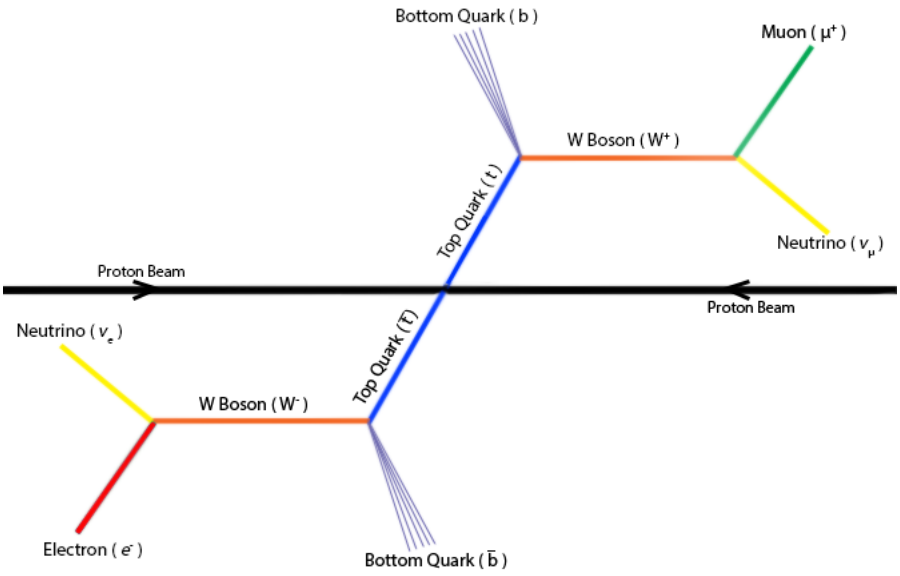


Figure 1.1: Schematic representation of the $t\bar{t}$ system.

The ATLAS detector is able to record the characteristics of Bottom Quarks, which are detected as a jet of particles rather than a single particle, and leptons, the muon (that has a positive charge) and electron (with a negative charge). However, the neutrinos do not react with the detector and, therefore, their characteristics are not recorded. To reconstruct the Top Quarks it is necessary to have the information of all the final particles, so the neutrino characteristics need to be determined. Since the $t\bar{t}$ system obeys a set of properties, and using the information of the quarks and leptons, the neutrinos characteristics can be analitically calculated. The process of reconstructing the neutrinos is referred as kinematical reconstruction. The reconstruction of the whole $t\bar{t}$ system has a degree of certainty associated, which determines its quality. The quality of these reconstructions directly impact the research conducted by LIP.

The amount of Bottom Quark jets and leptons detected may vary between events, due to other reactions occurring at the same time of the Top Quark decay. As represented in figure 1.1, 2 jets and 2 leptons are needed to reconstruct the $t\bar{t}$ system, but the data for an event may have many more of these particles associated. It is necessary to reconstruct the neutrinos, and then the whole system, for every combination of 2 jets and 2 leptons (often referred only as *combination*) available in the input data, and only chose the most accurate reconstruction for a given event.

Another factor affecting the quality of the reconstruction is the experimental resolution of the ATLAS particle detector, which associates an error up to 2% with every measurement made. If the measurements of the jets and leptons are not precise enough the kinematical reconstruction will produce inaccurate neutrinos and affect the overall reconstruction of an event. This might render an event useless, which oth-

erwise would provide relevant physics information. It is possible to overcome this problem by performing the whole $t\bar{t}$ system reconstruction a given amount of times for each combination of 2 Bottom Quark jets with 2 leptons, randomly varying the particle characteristics (momentum, energy and mass of the jets and leptons), with a maximum magnitude of 2% of the original value. The amount of variations performed per combination will directly proportional the final quality of the event reconstruction, as more of the search scope (defined by the experimental resolution error) is covered, compared to performing a single $t\bar{t}$ system reconstruction.

The look for the Higgs Boson is also part of the research being conducted at LIP. Figure 1.2 schematizes the Higgs Boson and Top Quark decay. It is possible to reconstruct the Higgs Boson from the two Bottom Quark jets that it decays to, and it can be performed alongside the $t\bar{t}$ system reconstruction. This adds at least two more jets to the event information, and it is not possible to know before the reconstruction which jets belong to the Higgs decay or the Top Quark decay. Considering this, the Higgs reconstruction must be performed after the $t\bar{t}$ system reconstruction, in such a way that the jets chosen to reconstruct it must not be the ones used in the $t\bar{t}$ system reconstruction. Adding this new jets increases the number of jets/leptons combinations to test in the kinematical reconstruction, and for each $t\bar{t}$ system reconstruction the Higgs must be also reconstructed. Now, the quality of the event reconstruction depends on the quality of both $t\bar{t}$ system and Higgs Boson reconstructions.

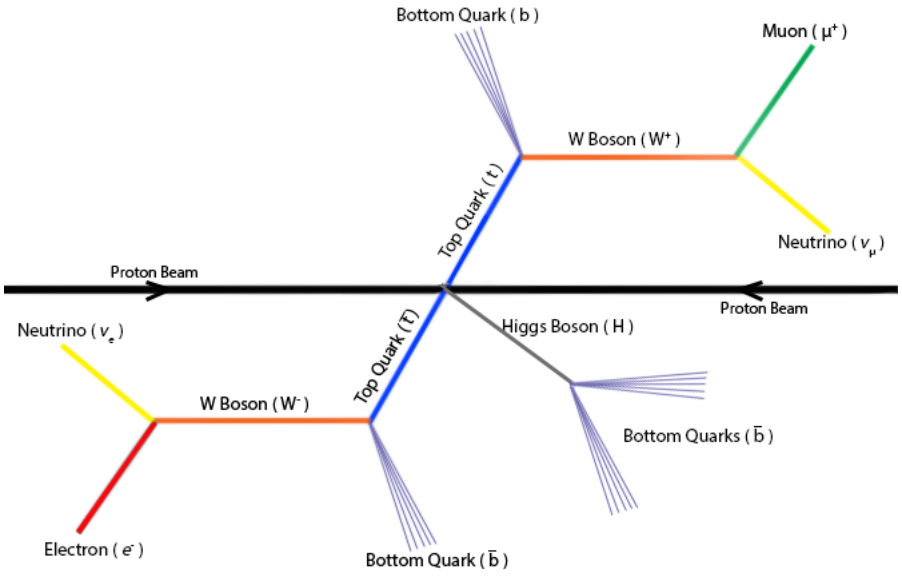


Figure 1.2: Schematic representation of the $t\bar{t}$ system with the Higgs Boson decay.

This specific analysis presented is performed by an application developed by LIP researchers, the `ttH_dilep`. The application receives input data file with a set of event data and attempts to reconstructs both the $t\bar{t}$ system and the Higgs Boson of each event, using the processes described. These files are usually 1 GB long and the LIP processes huge quantities for each deadline using this application for researching on Top Quark and Higgs Boson physics. A in-depth computational analysis of `ttH_dilep` is presented in chapter 3, where its flow is presented, computationally characterized and the critical region affecting the performance is identified.

1.3.2 Goals

It is possible for `ttH_dilep` to perform more variations per event by increasing the performance of the Top Quark and Higgs Boson reconstructions, boosting the quality of the results, within the same, or less, time that the current non-varied event processing occurs. The objective of this dissertation is to take the `ttH_dilep` scientific application made by physicists, which the main concern during its development was the correcteness of the code rather than its performance, and improve its efficiency by (i) identifying the bottlenecks and optimizing the code, (ii) increasing the performance by resorting to parallelism for homogeneous and heterogeneous systems, assessing the performance and usability of hardware accelerators for this type of problem, and (iii) the development of a simple scheduler for managing the workload distribution among various instances of the same sequential or parallel application (i.e., an application which needs to process a large set of separate input files) on homogeneous systems, without changing the application source code.

This work will give a inside perspective of how scientific applications are being developed by self taught programmers with little to no background in computer science, and help define a set guidelines for coding efficient, and possibly parallel, applications. All the changes that will be made to the `ttH_dilep` application, including the introduction of parallelism, will be as modular as possible from the context of this specific application, in such a way that they might be portable to other applications, only requiring. The scheduler will offer parallelization of the data to process at the application level, requiring minor to no changes to the application source. The implementations will be structured in such a way that the parallelization mechanisms and the scheduler can be improved and possibly transformed in a tool used by the researchers at LIP.

1.3.3 Scientific Contribution

This dissertation work aims to improve the quality of a specific research field of LIP, by providing a set of tools and know-how to improve the performance of a set scientific applications and expose the problematic of unefficient usage of computational resources. By improving the applications, and consequently the quality of the research, LIP will gain an advantage over other research groups in the look for new Top Quark and Higgs Boson physics. By experiencing the process of optimizating this kind of scientific applications, it is possible to provide physicists with know-how and tools and mechanisms for optimizing and extract parallelism, increasing the performance of future applications. By developing applications that efficiently use the computational resources it is possible to reduce the investment in new hardware, which otherwise would have small pratical returns.

1.4 Dissertation Structure

This dissertation has 5 chapters and their summary is presented below:

Introduction

The dissertation is first presented by contextualizing the scientific background of CERN and LIP organizations, as well as their current research projects, which are closely involved in this work. The motivation for the dissertation is presented in section 1.3, with the problem contextualized from a physics perspective in subsection 1.3.1. The Goals, subsection 1.3.2, states the objectives to be achieved by this work, in terms of improving the research and application development quality by implementing a set of solutions for homogeneous and heterogeneous systems, while assessing the

efficiency and usability of hardware accelerators in the latter. The scientific contribution of this work is presented in subsection 1.3.3. Subsection 1.4 overviews the structure of this dissertation.

Technological Background

This chapter presents the current technological state of the art in terms hardware and software. Hardware-wise, both homogeneous and heterogeneous system architectures and details are presented in sections 2.1.1 and 2.1.2, respectively. A contextualization of current hardware accelerators is also made in the latter. Software-wise is presented in section 2.2. Various frameworks and libraries are presented for homogeneous systems and accelerators in sections 2.2.1, 2.2.2, 2.2.3 and 2.2.4. Section 2.2.5 presents the available frameworks for parallelization in heterogeneous systems. Finally, current solutions for profiling and debugging parallel applications is presented in section 2.2.6.

`ttH_dilep` Application

The `ttH_dilep` application for event reconstruction is presented in this chapter. Its dependencies are presented. The flow of the application is presented in section 3.1, accompanied by a schematic representation. Its main functions are presented and the schematic flow is compared against a callgraph of the application to help understanding what happens in each of the most important functions. The critical region is identified in section 3.2 and characterized in subsection 3.2.1. Some initial optimizations to the code are presented in subsection 3.2.2.

Parallelization Approaches

For different parallelization alternatives are presented in this chapter. For homogeneous systems, a shared memory parallelization is discussed in section 4.1, where the abstract heuristic used is shown, and the implementation and a performance analysis are presented in subsections 4.1.1 and 4.1.2, respectively. For heterogeneous systems using hardware accelerators, two alternatives are presented: using GPU as an accelerator, in section 4.2, with its implementation and performance discussed and analyzed in subsections 4.2.1 and 4.2.2; using the Intel Xeon Phi as an accelerator in section 4.3 and its implementation discussed in subsection 4.3.1. A software scheduler for managing workload distribution among applications for homogeneous shared memory systems is presented in section 4.4. Its implementation details and performance analysis are shown in subsections 4.4.1 and 4.4.2.

Conclusions & Future Work

This chapter concludes the dissertation, presenting an overview of the results obtained by the work developed, on both homogeneous and heterogeneous systems. Guidelines for future work, on improving the test case application and providing parallel solutions abstracted from the programmer for future application development, are presented.

Chapter 2

Technological Background

This chapter presents the current technological state of the art in terms hardware and software. Hardware-wise, both homogeneous and heterogeneous system architectures and details are presented in sections 2.1.1 and 2.1.2, respectively. A contextualization of current hardware accelerators is also made in the latter. Software-wise is presented in section 2.2. Various frameworks and libraries are presented for homogeneous systems and accelerators in sections 2.2.1, 2.2.2, 2.2.3 and 2.2.4. Section 2.2.5 presents the available frameworks for parallelization in heterogeneous systems. Finally, current solutions for profiling and debugging parallel applications is presented in section 2.2.6.

2.1 Hardware

Computer systems originally had a very simple design, where a processing chip (CPU) is connected to a data storage unit (memory). The complexity of the processing chips increased, as well as the memory with the introduction of an hierarchy model. Current computing systems are usually made from multicore CPUs, various types of volatile and non-volatile memory and, in some cases, hardware accelerators.

2.1.1 Homogeneous systems

The most common systems are homogeneous, constituted from one or more CPU chips with their own memory bank (RAM memory) and interconnected by a manufacturer-specific interface. Although these systems use a shared memory model, where all the data is shared among CPUs, when considering a multiple CPU system, each CPU with its own memory bank, the system will have a Non Unified Memory Access (NUMA) pattern. This means that the access time of a CPU to a piece of memory in its memory bank will be faster than accessing memory on the other CPU bank. It is important to have the data on the CPU memory bank that the application will run to avoid the increased costs of NUMA.

Figure 2.1 shcematizes the structure of a homogeneous system, in a shared memory environment with the interconnection between CPUs responsible for the NUMA pattern.

CPU chips

Gordon Moore predicted, in 1965, that for the following ten years the number of transistors on CPU chips would double every 1.5 years [15]. This was later known as the Moore's Law and it is expected to remain valid at least up to 2015. Initially, this allowed the increase in CPU chips clock frequency

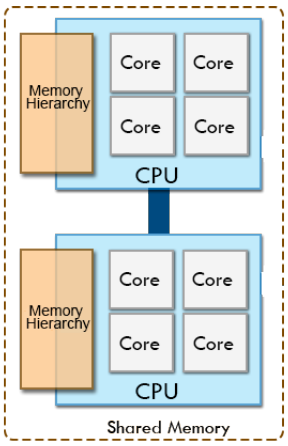


Figure 2.1: Schematic representation of a homogeneous system.

by the same factor as the transistors. Software developers did not expend much effort optimizing their applications and only relied on the hardware improvements to make them faster.

Due to thermal dissipation issues, the clock frequencies of CPU chips started to stall in 2005. Manufacturers shifted from making CPUs faster to increasing their throughput by adding more cores to a single chip, reducing their energy consumption and operating temperature. This marked the beggining of the multicore and parallel computing era, where every new generation of CPUs get wider, while their clock frequencies remain steady.

The CPU chips are designed as general purpose computing devices, based on a simple design consisting of small processing units with a very fast hierarchized memory attached (cache, which purpose is to avoid slow accesses to global memory), and all the necessary data load/store and control units. They are capable of delivering a good performance in a wide range of operations, from executing simple integer arithmetic to complex branching and SIMD (single instruction multiple data, later explained) instructions. A single CPU core implements various mechanisms for improving the performance of applications, at the hardware level, with the most important explained next:

ILP instruction level parallelism (ILP) is the overlapping of instructions, performed at the hardware or software level, which otherwise would run sequentially. At the software level it is denominated as static parallelism, where compilers try to identify which instructions are independent, i.e., the result of one does not affect the outcome of the other, and can be executed at the same time, if the hardware has resources to do so. At the hardware level, ILP can be referred as dynamic parallelism as the hardware dynamically identifies which instructions execution can be overlapped while the application is running. Three mechanisms allow for ILP:

Out of order execution is the execution of instructions in different order as they are organized in the application binary, without violating any data dependencies. This technic exposes ILP, which otherwise would not be possible.

Super Scalarity is a mechanism which allows dispatching a certain amount of instructions to the respective arithmetic units in each clock cycle, increasing the throughput of the CPU. Instructions that are not data dependent can run simultaneously, as long as they use different arithmetic units.

Pipelining is the division of an instruction execution in stages. The stages range from loading the data, instruction execution in, also pipelined, arithmetic units and writing the results back to

memory. For example, this allows for an instruction to be loaded while other is being executed. Moreover, inside an arithmetic unit, multiple instructions can be simultaneously executed, as long as they are in different stages.

Speculative execution is the usage of branch prediction (predict which branch of a conditional jump will be executed, before knowing the condition result), which can use complex algorithms for predicting conditions, and start executing instructions in the chosen branch. If the prediction fails, the results are trashed and the other branch is executed. Current hardware is capable of executing both branches of a conditional jump and accept the one correct once the condition is resolved.

Vector instructions are a special set of intructions based on the SIMD model, where a single instruction is applied to a large set of data simultaneously. CPU instruction sets offer special registers and instructions that allow to take a chunk of data and execute an instruction to modify it in a special arithmetic unit. One of the most common examples is addition of two vectors. The hardware is capable of adding a given number of elements of the vectors simultaneously. This optimization is often done at compile time.

Multithreading is the execution of multiple hardware threads in the same core. This is possible by replicating part of the CPU resources, such as registers, and can lead to a more efficient utilization of the CPU core hardware. If one thread is waiting for data to execute the next instruction, other thread can resume execution while the first is stalled. It also allows a better usage of resources which would otherwise be idle during the execution of a single thread. If multiple threads are working on the same data, multithreading can reduce the synchronization between them and lead to a better cache usage.

A schematic representation of a modern CPU chip on its die is presented in figure 2.2. It is constituted of several, possibly multithreaded, cores, each with their own private level 1 and 2 caches and a level 3 cache shared among all cores. This level 3 cache allows fast comunication and synchronization of data between cores of the same CPU.

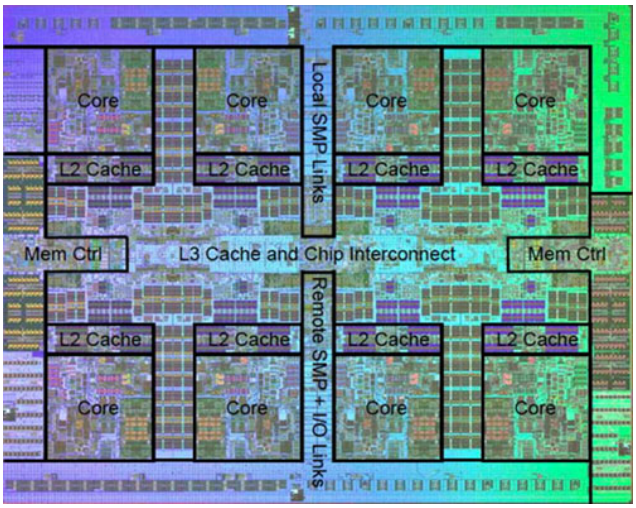


Figure 2.2: Schematic representation on a die of a CPU chip.

2.1.2 Heterogeneous systems

With the emerging use of hardware specifically designed for some computing domains, denominated hardware accelerators, which purpose is to efficiently solve a given problem, a new type of computing platform is becoming increasingly popular. This marked the beggining of heterogeneous systems, where one or more CPU chips, operating in a shared memory environment similar to homogeneous systems, are accompanied by one or more hardware accelerators. The CPUs and accelerators operate in a distributed memory model, meaning that data must be explicitly passed from the CPU to the accelerator memory, and vice-versa.

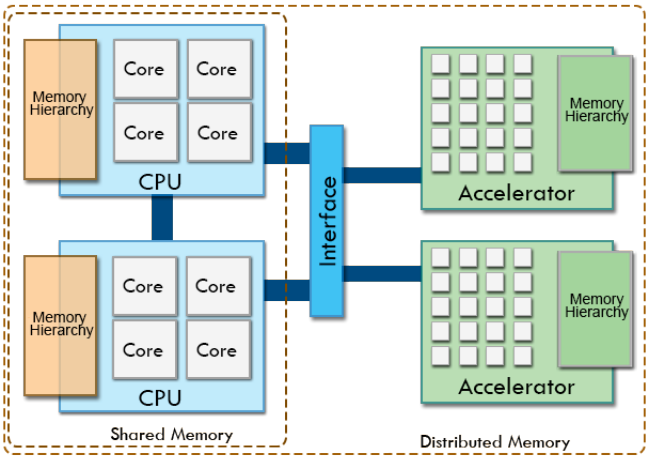


Figure 2.3: Schematic representation of a heterogeneous system.

Figure 2.3 presents a schematic representation of a heterogeneous system. Note that both CPUs must use the same interface to communicate with the hardware accelerators. This interface has a high latency for memory transfers, making it a critical bottleneck for applications that use accelerators.

Hardware accelerators are usually made from a huge amount of small and simple processing units, designed to achieve the most performance possible on specific problem domains, opposed to general purpose CPUs. They are usually oriented for massive data parallelism processing (SIMD architectures), where a single operation is performed on huge quantities of independent data, with the purpose of 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 [16], and Digital Signal Processors (DSP) [17]. An heterogeneous platform may have one or more accelerator devices of the same or different architectures.

As of June 2013, over 50 of the TOP500's list [18] are powered by any kind of hardware accelerator, which indicates an exponential growth in usage compared to previous years. The Intel Xeon Phi is becoming increasingly popular, being the accelerator device of choice in 11 clusters of the TOP500. NVidia GPUs remain as the most used accelerator.

Graphics Processing Unit

The Graphics Processing Units (GPU) was one of the first hardware accelerators on the market. Their purpose is to accelerate image processing, which started of as simple pixel drawing and evolved to support complexity of 3D scene rendering, such as transforms, lighting, rasterization, texturing, depth testing, and display. Due to the industries demand for costumizable shaders, this hardware later allowed some flexibility for the programmers to modify the image synthesing process. This also allowed using this GPUs as a hardware accelerators for other purposes than image processing, such as scientific computing.

The GPU architecture is based on the SIMD model. Its original purpose is to process and synthethise images, which are, computationally, a large set of pixels. The processing of each pixel usually does not depend on the processing of its neighbours, or any other pixel on the image, making the process data indenpent. This allows for processing the pixels simultaneously. The massive data parallelism is one of the most important factors that molded the design of the GPU architecture.

As the GPU manufacturers allowed more flexibility for programming their devices, the High Performance Computing (HPC) community started to use them for solving specific massively data parallel problems. One of the most known examples is matrix arithmetic, such as additions and multiplications. However, GPUs had some important features only oriented for image processing that affected its use in other situations. One example is that initially the GPU only supported float point arithmetic. Due to the increase demand for these devices by the HPC community, manufacturers started to generalize more of the GPUs features and later began producing accelerators specificaly oriented for scientific computing. NVidia is the number one GPU manufacturer for scientific computing GPUs, with a wide range of available hardware known as the Tesla. These devices have more GDDR RAM, processing units and a slightly different structural design suitable for use in cluster computational nodes, with a different size and cooling mechanisms. The chip has suffered some changes too, increasing the cache size and the amount of processing units. The NVidia Tesla C2070 (Fermi architecture [19]) was used during this dissertation work.

The NVidia GPU architecture has two main components: computing units (Streaming Multiprocessors, also known as SM) and the memory module (global external memory, GDDR5 RAM, and a 2-level in-chip cache and shared memory block). Each SM contains a set of CUDA cores, NVidia designation for their basic processing units that perform both integer and float point arithmetic (additions, multiplications and divisions). These SMs also have some specialized processing units for square root, sins and cosines computation, as well as a warp scheduler (warps are later explained) that matches the CUDA threads to CUDA cores, load and store units, register files and the L1 cache/shared memory. The L2 cache is shared among all the SMs in a GPU chip.

A warp is a set of CUDA threads (it has a size of 32 CUDA threads in the Fermi architecture), scheduled by the SM scheduler to run on its SM at any given time. A warp can only be constituted by CUDA threads from the same block[1].

Accesses to the GPU global memory have a high latency associated that can cause the CUDA threads to be stalled waiting for data. The strategy behind the GPU architectures is to provide the device with a huge amount of threads, allowing the schedulers to keep a scoreboard of which warps are ready to execute and which are waiting for data to load. This grants the scheduler with enough threads to always have a warp ready for execution, preventing the starvation of the SMs.

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/s in each direction of the channel). Memory transfers

[1]The GPU thread hierarchy and organization is presented in section 2.2.4.

between the CPU and GPU must be minimal as it greatly restricts the performance.

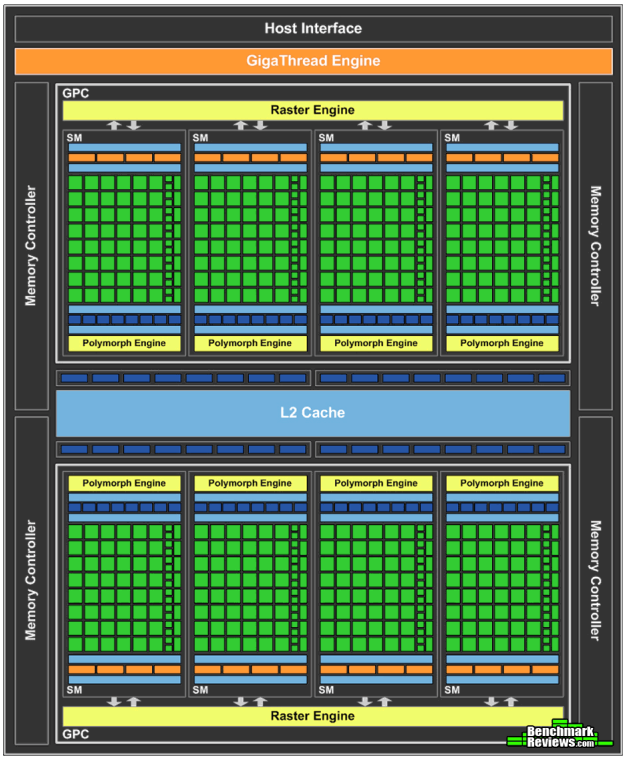


Figure 2.4: Schematic representation of the NVidia Fermi architecture.

In the Tesla C2070, with its architecture schematized in figure 2.4, each one of th 14 SM in the chip have 32 CUDA, making a total of 448 CUDA cores. There are 4 Special Functional Units (SFU) in each SM to process special operations such as square roots and trignometric arithmetic.

These devices have a slightly different memory hierarchy than CPUs, but still with the faster and smaller memory closer to the processing units (CUDA cores). Each CUDA thread can use up to 63 registers, but decreasing with the amount of threads used, 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 high latency global memoy).

Within each 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 all SMs, opposed to the L1 cache. 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 the control of coalesced memory accesses. As the load units fetch memory in blocks of 128 bits, it is possible to reduce the amount of loads by synchronizing and grouping threads that need to load data which is in contiguous positions. This grouping is made by the memory controller, but only if the threads need the contiguous data simultaneously.

Intel Many Core Architecture

The Intel Many Integrated Core (MIC) architecture, with the current production device known as Intel Xeon Phi, has a different conceptual design than the NVidia GPUs. A chip can have up to 61 multithreaded cores, with 4 threads per core. Rather than extract performance by resorting to massive parallelism of simple tasks, the design favors vectorization as each core has 32 512 bit wide vector registers [16].

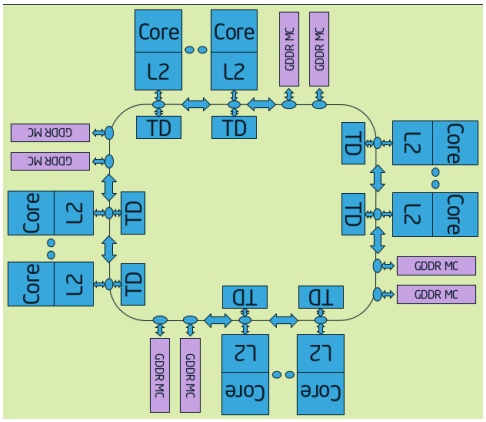


Figure 2.5: Schematic representation of the Intel MIC architecture.

The vector registers are capable of holding 16 single precision float point values. Each core has L1 cache with a size of 64 KB for data and 64 KB for instructions, and 512 KB L2 cache. There is no shared cache between the cores inside the chip. The device is produced with 6 or 8 GB GDDR5 RAM, with a maximum bandwidth of 320 GB/s. Its design is more oriented for memory bound algorithms, as opposed to GPUs (Fermi only has a bandwidth of 192.4 GB/s). Intel claims that it will later launch a device tuned for compute bound problems.

Unlike conventional CPUs, the MIC cores do not share any cache, therefore cache consistency and coherence is not assured by the hardware. It works as distributed memory system, but consistency can be assured by software, with a high latency. The cores are connected in a bidireccional ring network, as represented in figure 2.5. The MIC uses the same instruction set as conventional x86 CPUs. Intel claims that this allows to easily port current applications and libraries to run on this device.

The MIC architecture has some simplifications compared to the CPU architecture, so that it is possible to fit so many cores inside a single chip. MIC does not have out of order execution, which greatly compromises the use of ILP. Also, the clock frequency is only of 1 GHz, less than half of the modern CPUs.

The Xeon Phi has two operating modes:

Native: the device acts as system itself, with one core reserved for the operative system. The application and all libraries must be compiled specifically to run on the device, as well as copied, along with the necessary input data, to the device memory prior to execution. No further interaction with the CPU is required.

Offload the device acts an accelerator, accessory to the CPU. Only part of the application is set to run on the Xeon Phi, and data must be explicitly passed between CPU and device each time code will execute in it. All library functions called inside the device must be explicitly compiled and it is not possible to have an entire library compiled simultaneously for the Xeon Phi and CPU.

Other hardware accelerators

More hardware accelerators are coming to the market due to the increasingly popularity of GPUs and Intel MIC among the HPC community. Texas Instruments developed their new line of Digital Signal Processors, best suited for general purpose computing while very power efficient. Their capable of delivering 500 GFlop/s (giga float pointing operations per secong), consuming only 50 Watts [17].

ARM processors are now leading the mobile industry and, alongside the new NVidia Tegra processors [20] that are steadly increasing their market share, are likely to be adopted by the HPC community[2] due to their low power consumption while delivering high performance [21]. The shift from 32-bit to 64-bit mobile processors is happening due to the increase in complexity of mobile systems and applications.

2.2 Software

Most programmers are only used to code and design sequential applications, showing a lack of know-how to develop algorithms for parallel environments. This issue is even greater when considering heterogeneous systems, where programming paradigms shift when considering different hardware accelerators. The mainstream industry is still adopting the use of multicore architectures with the purpose of increasing their processing power, causing a lack in the academic formation of programmers in terms of optimization and parallel programming. Self taught programmers have an increased obstacle due to the lack of theoretical basis when using these new parallel programming paradigms.

Programming for multicore environments requires some knowledge of the underlying architectural concepts. Shared memory, cache coherence and consistency and data races are architecture-specific aspects that the programmer does not face in sequential execution environments. However, these concepts are fundamental not only to ensure efficient use of the computational resources, but also the correctness of the application.

Heterogeneous systems 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 usually cannot be easily ported to these devices expecting high performance. To optimize the code for these devices it is necessary a deep understanding of the architectural principles behind their design.

The most important aspect for ensuring the correcteness of an application is to control data races, i.e., concurrent accesses of different threads to shared data. As an example, when modifying shared data, the programmer must ensure that different threads are not simultaneously changing the same piece of memory by serializing the operations. Further control is required if the order of the operations is important. If one thread wants to change a piece of data while other wants to read it, it is necessary to define which of the threads has the priority, as it can affect the outcome of the rest of both threads operations.

The workload balance between the cores of a single CPU chip is an important aspect to extract performance and get the most efficient usage of the available resources. A bad workload balance may cause some cores of the CPU to be used most of the time while others remain idle, making the application to take more time than necessary to execute. A good load balancing strategy ensures that all the cores are used as most as possible. Considering a multi-CPU system, it is important to manage the data in such a way that it is available in the memory bank of the CPU that will need it to avoid the NUMA

[2] e.g. the ARM based Montblanc project will replace the MareNostrum in the Barcelona Supercomputing Center (BSC)

latency. The same concepts apply when balancing the load between CPU and hardware accelerators, with the increased complexity of the high latency data transfers of the PCI-Express interface.

Some computer science groups developed libraries that attempt to abstract the programmer from specific architectural and implementation details of these systems, providing an easy API as similar as possible to current sequential programming paradigms. Some frameworks that attempt to abstract the inherent complexity of heterogeneous systems are already in the final stages of development. The most used are presented through the next subsections.

2.2.1 pThreads

Threads are the elemental unit that can be scheduled by the operating system. POSIX Threads (pThreads) are the standard implementation for UNIX based operating systems with POSIX conformity, such as most Linux distributions and Mac OS. The pThreads API provides the user with primitive for thread management and synchronization. Since this API forces the user to deal with several implementation details, such as data races and deadlocks, the industry demanded the development of high level libraries, which are usually based on pThreads.

2.2.2 OpenMP, TBB and Cilk

OpenMP [22], Intel Threading Building Blocks (TBB) [23] and Cilk [24] are the response for the industry demands for a higher abstraction level APIs.

The OpenMP 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. It is based in a work sharing strategy, where a master thread spawns a set of slave threads and compute a task in a shared data structure.

Intel TBB employs a work stealing heuristic, where if the task queue is empty a thread attempts to steal a task from other busy threads. It provides a scalable parallel programming task based library for C++, independent from architectural details, only requiring a C++ compiler. It automatically manages the load balancing and some cache optimizations, while offering parallel contructors and sychronization primitives for the programmer.

Cilk is a runtime system for multithreading programming in C++. It maintains a stack with the remaining work, employing a work stealing heuristic very similar to Intel TBB.

2.2.3 Message Passing Interface

The Message Passing Interface (MPI) [25], designed by a consorcium of both academic and industry researchers, has the objective of providing a simple API for parallel programming in distributed memory environments. It relies on point-to-point and group messaging communication, and is available in Fortran and C. The data must be explicitly split and passed among the processes by the programmer. It is often used in conjunction with a shared memory parallel programming API, such as OpenMP, for work sharing between computing nodes, with the latter ensuring the parallelization inside each node.

2.2.4 CUDA

The Compute Unified Device Architecture (CUDA) is a computing model for hardware accelerators launched in 2007 by NVidia. It aims to provide a framework for programming devices similar architecture

to the NVidia GPUs. It has a specific instruction set architecture (ISA) and allows programmers to use GPUs for other purposes than image rendering.

NVidia considers that a parallel task is constituted by a set of CUDA threads, which execute the same instructions (conditional jumps are a special case that will be later explained) 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 to consider 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 thread is the most basic data independent parallel element, which can run simultaneously with other CUDA threads but itself cannot be parallelized, and is organized in a hierarchy presented in figure 2.6. 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 matrix example, each CUDA thread corresponds to an element of the matrix, computing its value, and is organized in a block of many CUDA threads that can represent all the computations for a single line of the matrix. The grid holds all the blocks responsible for computing all the new values of the matrix. Note that both the blocks and the grid have a limited size.

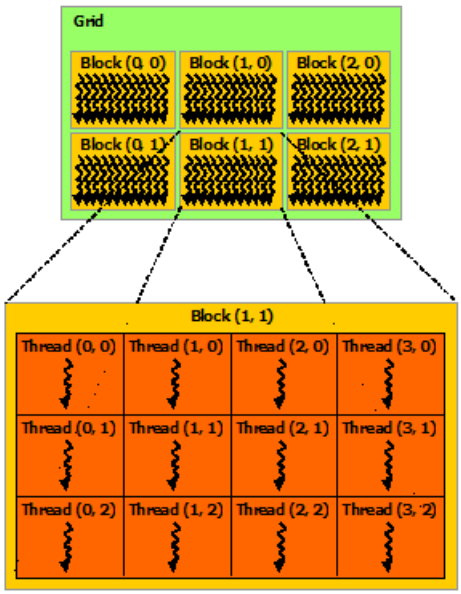


Figure 2.6: Schematic representation of CUDA thread hierarchy.

When programming these devices, conditional jumps must be avoided if they cause different CUDA threads within the same warp execute different branches. 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 divergent branches will be executed sequentially, doubling the warp execution time.

2.2.5 Parallelization frameworks for heterogeneous systems

OpenACC

OpenACC [26] 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 [27]. Also, they are working alongside with the members of OpenMP to create a new specification supporting accelerator devices in future OpenMP releases [28].

GAMA

The GAMA framework [29] 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 all the architecture details 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.

2.2.6 Profiling and debugging

VTune

Intel VTune profiler [30] is a proprietary tool for performance analysis of applications. It provides an easy to use tool which analyzes the applications, identifying its bottlenecks, without any change to the source code. VTune also provides visualization functionalities making profiling of parallel applications a simple task for developers with small experience.

Performance API

The Performance API (PAPI) [31] specifies an API for hardware performance counters in most modern processors. It allows programmers to measure the performance counters for specific regions of an application, evaluating metrics such as cache misses, operational intensity or even power consumption. This analysis helps classifying the algorithms and identify possible bottlenecks at a very low abstraction level.

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 [32], 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 [33]. 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.

Chapter 3

`ttH_dilep` Application

The `ttH_dilep` application for event reconstruction is presented in this chapter. Its dependencies are presented. The flow of the application is presented in section 3.1, accompanied by a schematic representation. Its main functions are presented and the schematic flow is compared against a callgraph of the application to help understanding what happens in each of the most important functions. The critical region is identified in section 3.2 and characterized in subsection 3.2.1. Some initial optimizations to the code are presented in subsection 3.2.2.

The LIP research group developed the `ttH_dilep` application to reconstruct the Top Quark and Higgs Boson resultant from a collision, and fits in the Tier-3 grid computational resources. Its name derived from the problem it was design to solve: the tt is relative to the reconstruction of the $t\bar{t}$ system; the H is derived from the Higgs boson reconstruction; the *dilep* is the name of the function responsable for the kinematical reconstruction, as it needs two leptons (di-lep) as input.

The application has two main dependencies in external libraries. The first, and most important, is on ROOT [34], a object oriented framework, developed at CERN, which provides a set of funcionalities oriented for handling, analyzing and displaying results relative to the large amounts of data collected at the LHC. It provides an API for reading and storing data in the standard formats accepted by all the tier centers, classes for representing physic elements, mathematical routines, pseudo-random number generators, histograming, curve fitting minimization and data visualization methods. It was originally designed and developed by physicists, self taught programmers. This results in a framework with room for improvement, such as code restructuration in some modules related to the data analysis that would increase its performance. Other possible optimizations are possible by replacing some mathematical functionalities by using faster libraries, such as BLAS [35] or MKL [36]. The Parallel ROOT Facility (PROOF) [37] is a ROOT extension for data distribution on distributed memory systems.

The second dependency is on the LipMiniAnalysis library. It is a modified version of LipCbrAnalysis, a library developed LIP for in-house use, which provides a skeleton (not an API) for creating an analysis application. It offers a code structure common for most applications developed by LIP, where the programmer codes specific parts of it depending on the analysis. The current implementation of this library is not suitable for parallelization in shared or distributed memory, as later explained in chapter 4.

During the following sections the flow of the application and an early profiling, identifying and characterizing the bottlenecks, will be presented.

3.1 Application Flow

This section describes the workflow of the ttH_dilep analysis. The application flow is schematized in figure 3.1. It has two main steps repeated for every event in the input data file:

Load Event Data: information relative to the event, the Bottom Quark jets and leptons characteristics, as well as other control data, are loaded to a global state. Most of this state belongs to the LipMiniAnalysis and it is overwritten every time a new event is loaded. The function responsible for loading and processing all events is named Loop.

Process Event: most of the event processing is performed in the DoCuts function. In this function applies a set of filters to an event (referred as cuts), where it can be rejected if it does not match the criteria of any of the filters. These cuts test many characteristics of the events data, such as the number of isolated leptons with opposite signs, the number of jets and the value of the particles masses. Only the events that reach cut number 20 are fit for the $t\bar{t}$ system and Higgs boson reconstructions, which are performed in this filter by a complex function named ttDilepKinFit. From a computational point of view, all the cuts are simple with the exception of cut 20.

ttDilepKinFit: this is the function responsible for the event reconstruction. It has an outer loop that iterates through all the possible combinations of 2 Bottom Quark jets and 2 leptons. The combination to process is determined and an inner loop iterates through the number of variations per combination, defined at compile time. These are the variations of the particle characteristics responsible for improving the reconstruction quality. The next step is to apply the variation to the particles characteristics and then attempt to reconstruct $t\bar{t}$ system (kinematical reconstruction). If a reconstruction is possible, the Higgs Boson is also reconstructed and the combined probability of the reconstructions is computed. If not, that variation is discarded, as it is not possible to reconstruct the Higgs Boson without information relative to the $t\bar{t}$ system. The probability depends on the accuraccy of the kinematical and Higgs Boson reconstructions. Most of the data manipulated by this cut is stored in the global state of LipMiniAnalysis.

This schematic representation of the application flow was designed based on the source code and the callgraphs analysis, obtained by using the callgrind tool from Valgrind [38]. Besides giving an inside of the application structure, this tool provides simple profiling information, measuring how much percentage of time is spent in each function, which is very useful for a rough assement of the possible bottlenecks. The callgraph for the application is presented in figure 3.2 and, since the objective is to run as many variations per combination within a reasonable time frame, as explained in section 1.3, a callgraph for 256 variations per combination is presented in figure 3.3. Note that only the relevant functions are included in the callgraphs below, as the originals are much larger.

When not performing variations, a third of the execution time is spent in writing outputs to files, using the ROOT library (note that all classes with a capital T as prefix belong to ROOT) and the rest is spent processing the events. The cut number 20, ttDilepKinFit, occupies most of the event processing time, where 6.2% of the time is on the kinematical reconstruction (dilep). It is clear that this cut is the portion of the application uses most of the event processing time. This becomes even more evident when considering 256 variations per event, where ttDilepKinFit uses 99.6% of the application execution time. For this number of variations, it is possible to see that the pseudo-random number generator

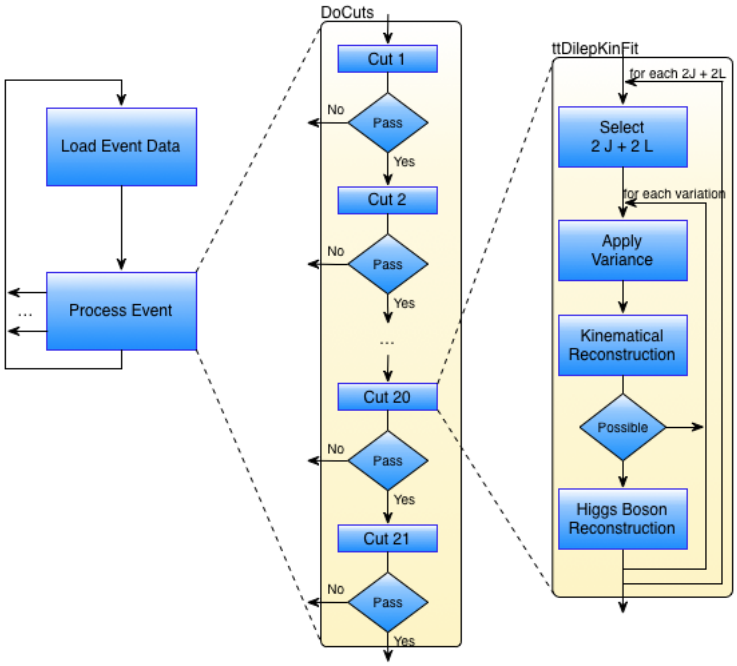


Figure 3.1: Schematic representation for the `ttH_dilep` application flow.

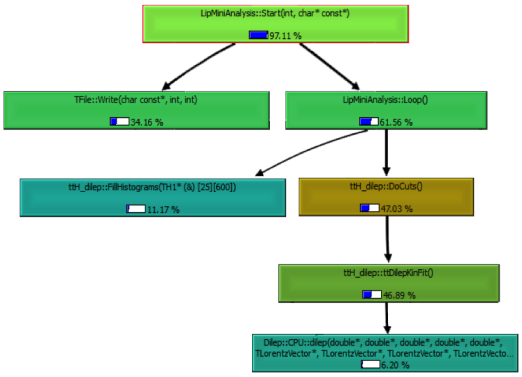


Figure 3.2: Callgraph for the `ttH_dilep` application on the compute-711 node[1].

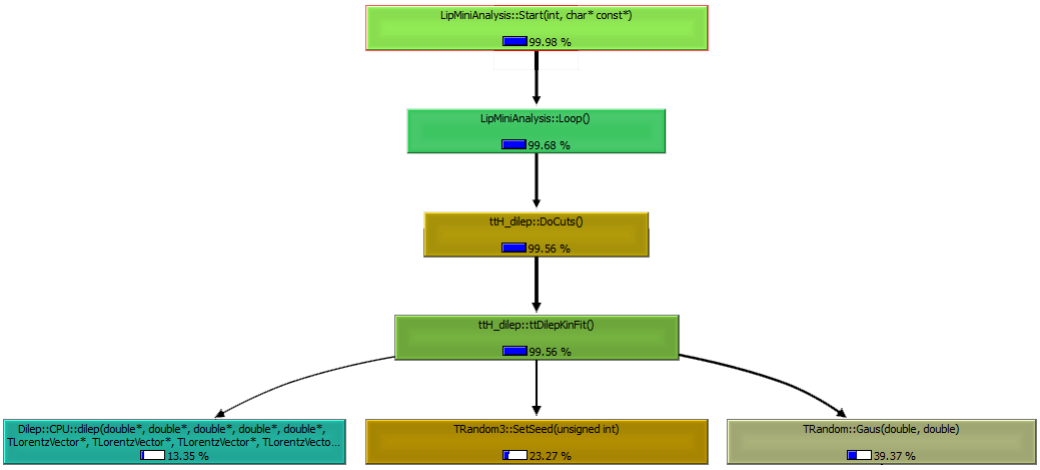


Figure 3.3: Callgraph for the `ttH_dilep` application on the compute-711 node for 256 variations per combination.

(TRandom and TRandom3 classes) is taking a substantial part of the cut execution. Table 3.1 presents the percentage of the application execution time spent on ttDilepKinFit, for various variations per combination. In section 3.2 a computational analysis of the critical region is presented, as well as some early optimizations to the application.

of variations/combination	1	2	4	8	16	32	64	128	256	512
% of time	46.9	62	76.9	87	92.9	96.3	98	98.9	99.6	99.7

Table 3.1: Percentage of the total execution time spent on the ttDilepKinFit function for various numbers of variations per combination.

3.2 Critical region computational characterization & optimization

This section focus on the computational characterization of the ttDilepKinFit function. It will be analyzed in terms of instruction mix, arithmetic and computational intensity and miss rate, with the purpose of understanding how this region of the code behaves, for various variations per combination, and classify it as a memory or compute bound algorithm. The test system used in this section is the compute-711[2]. Some initial optimizations, as well as other changes, made to the original application will be addressed in section 3.2.2.

3.2.1 Computational characterization

The ttDilepKinFit, often referred as KinFit, is the most time consuming task in ttH_dilep application. Figure 3.4 presents the evolution of the absolute and relative execution time of the KinFit function and the I/O of the application, which was also identified by callgraph 3.2 as a time consuming task for a low number of variations, and the rest of the computations.

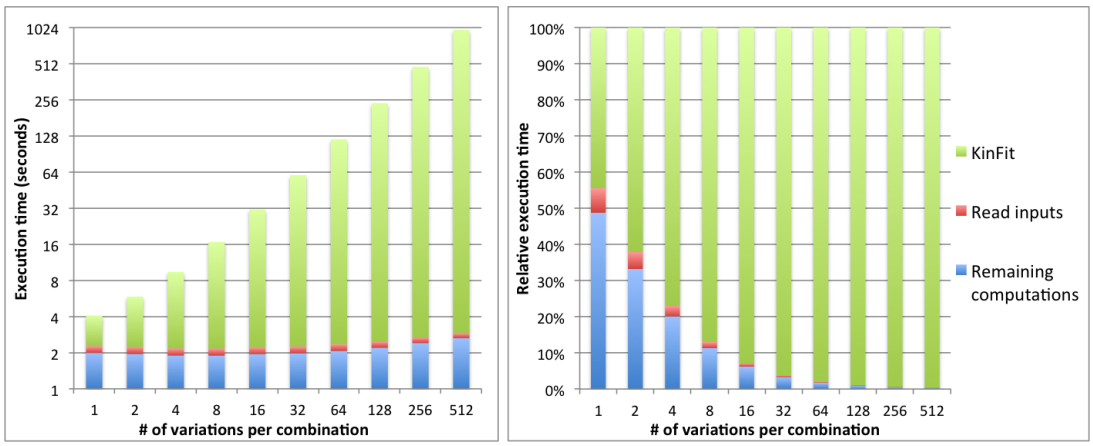


Figure 3.4: Absolute (left) and relative (right) execution times for the ttH_dilep application considering the ttDilepKinFit (KinFit) function, I/O and the rest of the computations.

While the I/O and the event processing, excluding the KinFit, execution times remain constant. There is only a slight increase for 256 and 512 variations as it causes more events to be reconstructed, which otherwise would not, and pass to the final cut 21 (see figure 3.1). As expected, the execution time of KinFit increases linearly with the number of variations per combination. This indicates that the

[2]See appendix for characterization of all the systems used.

arithmetic intensity of KinFit has the complexity of *O(N)*, where *N* is the number of variations. Consider a given input data file. With no variations, it is possible to consider that the time to process the events remains constant as many times the application is exectued, while using the said input data file. This is the initial problem size. The increase in variations, causing the number of reconstructions grows linearly, is responsible for the increase in the problem size. So, for *N* variations it is expected that KinFit execution time will increase by the same factor. This analysis is supported by the experimental data in figure 3.4, left graph.

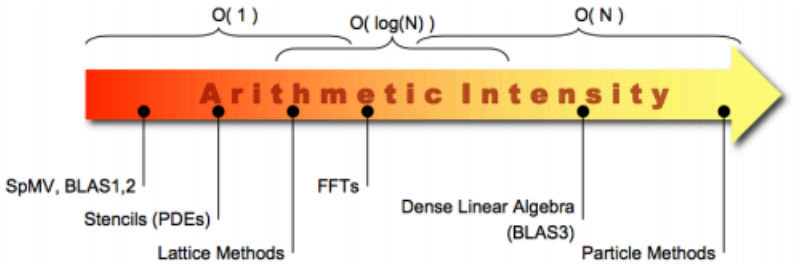


Figure 3.5: Arithmetic intensity for various domains of computing problems.

Figure 3.5 presents the complexity for various computational purposes. Problems with *O(1)* complexity are not likely to get any benefit from parallelization, opposed to problems with *O(N)* complexity, which are more easy to be efficiently parallelized.

For the rest of the computational characterization the `ttDilepKinFit` function was analyzed resorting hardware performance counters, using the Performance API [31]. The application was tested on the compute-601 system for various numbers of variations per combination. The instruction mix is presented in figure 3.6. It is evident, by analyzing the charts, that the performance of this function cannot be evaluated using the common FLOPS metric, as float point operations only account for 14% and 12% of the total instructions, for no and 512 variations respectively. `ttDilepKinFit` is very diverse in terms of instructions, with loads, branches and stores being the most used. The increase in the number of variations does not have a significant impact on the type of instructions issued.

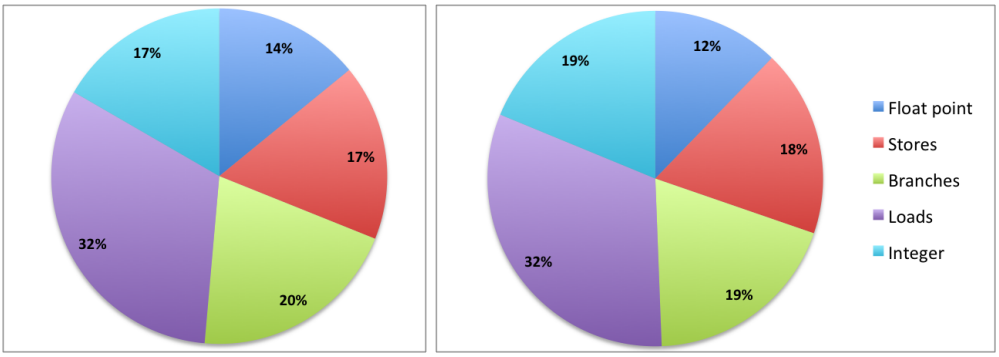


Figure 3.6: Instruction mix for the `ttDilepKinFit` with no and 512 variations, left and right images respectively.

The miss rate, for the 3 cache levels, is presented in figure 3.7. The miss rate on L1 cache remains constant at 70%, despite the change in variations. Even though this value is high, it is justified by the huge amount of different computations necessary to reconstruct an event, where data is only reused between subsequent instructionsher (considering the scope of the small L1 cache size), rather than periodically reused. The L2 cache miss rate is fairly low and decreases with the number of variations. As it has a larger size than L1 cache, the data is reused more often without causing misses to the L3 cache. Due

to the reuse of data in the L2 cache, and considering the larger size of the L3 cache, the miss rate on the last cache level increases with the number of variations. This is not a consequence of a poor cache management, but results from the decrease of the L2 cache miss rate, where most L2 cache misses are caused by the need of new data on the RAM memory. The use of more cores, in a shared memory environment, may result in a decrease in the miss rate, specially on L2 cache, as each core will process a smaller set of the data.

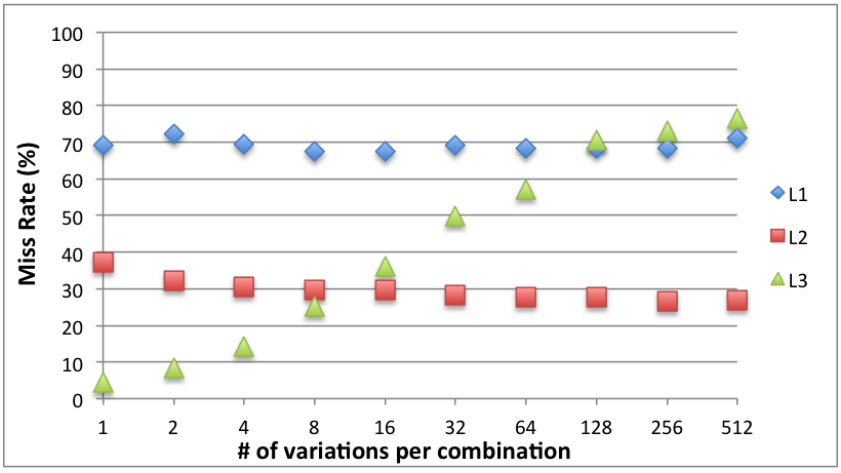


Figure 3.7: Miss rate on L1, L2 and L3 cache of `ttDilepKinFit` for various number of variations.

Operational intensity [39] is a performance metric used to characterize the bottlenecks of a given algorithm, based on the assumption that accesses to the system RAM memory are the main performance limitation on current systems. It allows classifying an algorithm as memory or compute bound, depending on which factor limits its performance. It requires a Roofline model[3] for the system in which the operational intensity is calculated. The operational intensity is defined by the amount of float point operations performed per each byte of data fetch from the system RAM memory. However, this is not a reliable metric for this specific problem as only 17% of the instructions of `ttDilepKinFit` are float point arithmetic (refer to chart 3.6). Instead, the computational intensity is considered a more reliable metric as it contemplates all kinds of instructions, with the exception of load and store instructions, which are not relevant for the algorithm structure.

Figure 3.8 presents the Roofline model for the compute-601 system with the computational intensity for `ttDilepKinFit` with no and 512 variations. For both number of variations it is obvious that this is a compute bound problem, only limited by the peak computational performance of the CPU. This indicates that the optimizations must be focused on increasing the computational throughput rather than on RAM memory access patterns. Note that cache optimizations are related to the computational throughput.

3.2.2 Initial optimizations

One limiting factor for the `ttDilepKinFit` function is the pseudo-random number generation (PRNG) performed by the `TRandom` ROOT class, specifically when performing a high number of variations, as seen in callgraph 3.3. When applying a variation to a combination in the `ttDilepKinFit` function, 12 PRNG values are needed at most. After generating the uniformly distributed numbers, a transformation is applied on them so that they obey a Gaussian distribution. The number is generated and transformed by the `Gauss` function, which occupies 39.4% of the total execution time. In the input

[3]Roofline model for the test system is presented and explained in appendix Appendix .2.

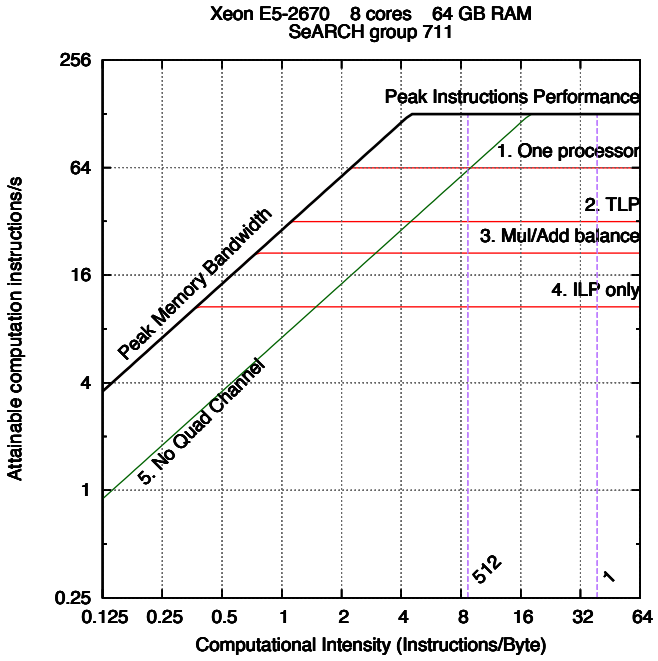


Figure 3.8: Roofline of the compute-601 system with the computational intensity of `ttDilepKinFit` for 1 and 512 variations.

data file used[4] there are 1867 events that reach the cut 20. This means that 244056 combinations are varied, with a total of 2928672 values generated for just a single variation per combination. This is also likely to happen with other input data files as they hold a similar number of events. In the `ttDilepKinFit` surce code, the `TRandom` generator is being reset with a new seed for every combination, causing 23.4% of execution time to be spent on the `SetSeed` function.

The pseudo-random number generator used by the `TRandom3` class is the Mersenne Twister [40], currently one of the most used generators for applications highly dependable on pseudo-random numbers. This algorithm produces 32-bit uniformly distributed pseudo-random numbers with a period of 2^{19937}. It has a relatively heavy state which is an integrant part on the algorithm flow. The generator is thread safe as long as different states are being used by different threads. The state can be shared among the threads but, modifying it must be serialized, and the number generated by one thread will affect the number generated by others. A proper implementation of this generator for a parallel algorithm must use one state per thread.

Since the period of the Mersenne Twister is 2^{19937} (approximately $4.3 * 10^{6001}$) and the maximum amount of numbers generated in the test file used, for 512 variations per combination, is $1.5 * 10^9$, it is not necessary to reset the seed of the `TRandom3` generator for every combination, which can significantly reduce the `ttDilepKinFit` execution time. Figure 3.9 illustrates the speedup obtained through this optimization. For a larger number of variations (from 16 to 512) the application is 1.7 to almost 2 times faster, without affecting the quality of the results.

NVidia offers, in the cuRand library [41], a parallel implementation of the Mersenne Twister for GPUs [42], which is important for porting the critical region to run on heterogenous systems with this hardware accelerator. It uses a precomputed set of 200 parameters, which can also be generated by the user, for the configuration of the generator's state, but offering a smaller period of 2^{11213}. The pseudo-random number generation is thread safe, allowing up to 256 threads sharing the same state per block. Two different

[4] See Appendix for a complete characterization of the test methodology.

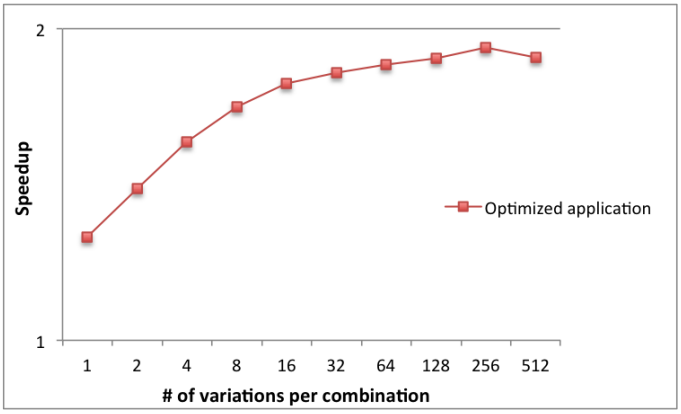


Figure 3.9: Speedup of the `ttH_dilep` application with the `TRandom` optimization.

blocks can safely operate concurrently on the same SM.

`TRandom` uses the Acceptance-Complement Ratio algorithm [43] for transforming the pseudo-random numbers from an uniform to a gaussian transformation. It is allegely 66% faster than the Box-Muller transformation [44] and similar in performance to the Ziggurat method [45]. The cuRand library only offers the Box-Muller transformation with a basic pseudo-random number generator so, to accuratly replicate the results, it is needed to replicate the `TRandom` gaussian method on GPU using the cuRand implementation of Mersenne Twister.

Other changes were made to the application, structuring the computation of the variations and kinematical reconstruction in modules, in such a way that when developing different parallel versions of `ttDilepKinFit` it is easy to switch between them, and also increasing code readbility. The definition of the number of variations to perform is set by environment variables, at runtime, rather than at compile time as it is on the original application, easing the user interaction.

Chapter 4

Parallelization Approaches

For different parallelization alternatives are presented in this chapter. For homogeneous systems, a shared memory parallelization is discussed in section 4.1, where the abstract heuristic used is shown, and the implementation and a performance analysis are presented in subsections 4.1.1 and 4.1.2, respectively. For heterogeneous systems using hardware accelerators, two alternatives are presented: using GPU as an accelerator, in section 4.2, with its implementation and performance discussed and analyzed in subsections 4.2.1 and 4.2.2; using the Intel Xeon Phi as an accelerator in section 4.3 and its implementation discussed in subsection 4.3.1. A software scheduler for managing workload distribution among applications for homogeneous shared memory systems is presented in section 4.4. Its implementation details and performance analysis are shown in subsections 4.4.1 and 4.4.2.

As presented in chapter 3, the critical region of the `ttH_dilep` application is the `ttDilepKinFit` function, which execution time increases linearly with the number of variations per combination. With the initial optimizations already applied to the original application, presented in section 3.2.2, the next step is to improve the performance by exploiting parallelism in the critical region. This chapter presents 4 parallelization approaches, 2 for homogeneous systems and 2 for heterogeneous systems, one using GPU and other using Intel Xeon Phi as hardware accelerators.

Although the critical region is only the `ttDilepKinFit` function, the best parallelization strategy is to simultaneously perform the reconstruction of different events, as it always exploits the same degree of parallelism independent from the amount of variations performed. The event processing is data independent and no synchronizations are required, reducing the parallelization overhead. Figure 4.1 presents a schematic representation of this parallelization approach. However, as explained in section 3.1, all the event data is stored globally on the LipMiniAnalysis library, and also part on the application. Every time an event is loaded the global data is overwritten, which, in a shared memory environment, causes the intermidiate results of the event processing of one thread to be overwritten by the processing of other thread.

Each input file has around 1 GByte size, which allows to store all events on RAM memory. The use of a proper data structure for storing the event information would benefit the application modularity and enable the possibility of this parallelization approach. In the current implementation, each event data is loaded from the input data file before processing it. With a global data structure, the application would benefit from the performance of sequential read of the hard drives, compensating the overhead of the data structure creation. The complexity of the changes necessary to both the application source code and LipMiniAnalysis library render this approach unreliable for the timeframe of this thesis dissertation.

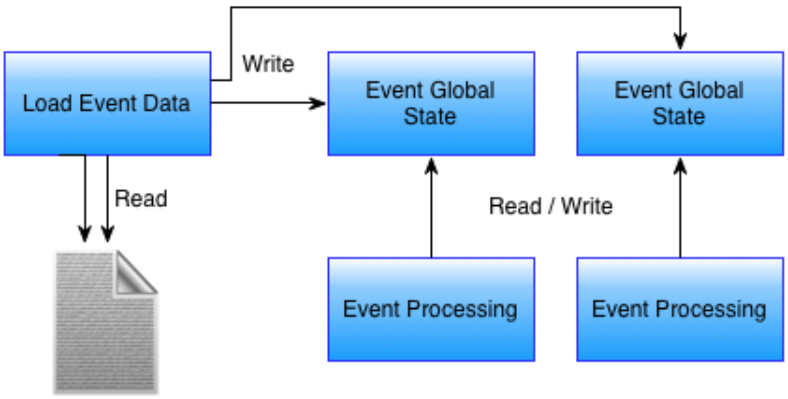


Figure 4.1: Schematic representation of the event-level parallelization model.

The `ttDilepKinFit` handles mostly private data to the function. The global data modified by this function can be controlled without any modifications to the LipMiniAnalysis library. Looking at the information presented in section 3.2.1, specifically in figure 3.4, most performance gains are expected to occur for 16 to 512 variations per combination as for that problem size the `ttDilepKinFit` occupies most of the application execution time.

The `ttDilepKinFit` execution is irregular, meaning that, depending on the event to be reconstructed, its execution time may vary. This is caused by the number of Bottom Quark jets and leptons combinations, which differs between events. Also, the execution flow is dependent on the kinematical reconstruction. If the $t\bar{t}$ system has a possible reconstruction, `ttDilepKinFit` attempts to reconstruct the Higgs Boson. If not, the Higgs Boson is not reconstructed reducing the function execution time. This translates in an irregular workload likely to affect the performance if the load balancing strategy is not suited for the problem.

The kinematical reconstruction is performed on the `dilep` function. The $t\bar{t}$ system obeys a set of properties expected from the Top Quark theoretical model. To reconstruct both Top Quarks it is required to know the characteristics of all resultant particles from their decay. However, since the neutrinos do not react with the detector, and their characteristics are not recorded and it is necessary to infer them using various properties of the system, such as momentum and energy conservation. Once the neutrinos characteristics are determined, the Top Quarks are possible to reconstruct. `dilep` analitically solves a system of 6 equations to infer the neutrinos characteristics and then reconstruct the Top Quarks. The function is dependent on only one class from ROOT, `TLorentzVector`, and only handles data private to the function.

4.1 Shared Memory Parallelization

The left scheme of figure 4.2 illustrates the current workflow of the `ttDilepKinFit`. The best approach is to parallelize the loop that iterates through all the jets/leptons combinations. However, the computation of the combinations cannot be performed in parallel as for chosing one combination it is necessary to know all the combinations chosen so far. Also, the combinations information is stored globally to the application so a data structure must be created to allow reconstructions of different combinations to be performed simultaneously.

For the shared memory parallelizaiton, the number of parallel tasks will be equal to the number of total reconstructions to process, which is the number of combinations times the number of variations per

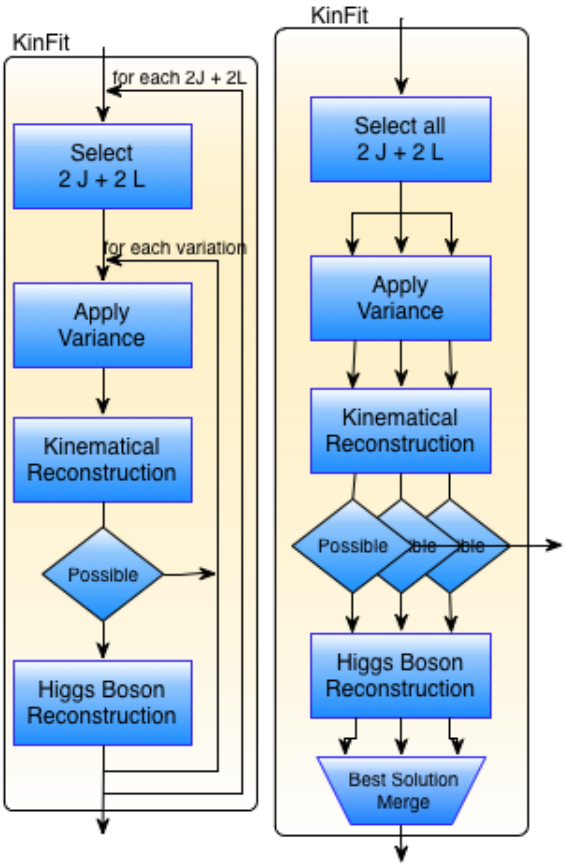


Figure 4.2: Schematic representation of the `ttDilepKinFit` sequential (left) and parallel (right) workflows.

combination. This small granularity of the tasks allows for the scheduler to better distrubute the work among the threads. Each task has a combination assigned and it applies a variation and reconstructs it. The result of the reconstruction is stored in a memory state private to the task so that, after all variations of all combinations are computed, a reduction is performed to find the best solution among all tasks. The reduction can also be parallelized, reducing its complexity from $O(N)$ to $O(log_2(N))$, where N is the number of elements to reduce.

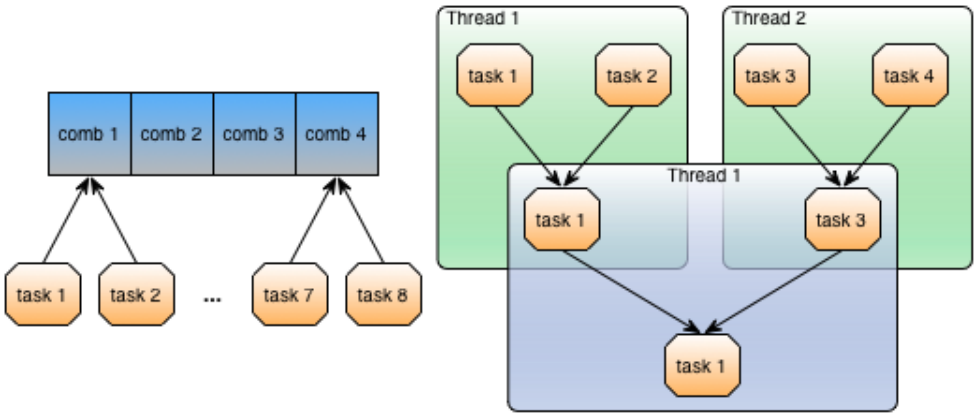


Figure 4.3: Schematic representation of the parallel tasks accessing the shared data structure (left) and the new parallel reduction (right).

Figure 4.3 presents the parallelization strategy using concurrent tasks sharing the combinations data structure and the strategy for the parallel reduction. This modifications to the workflow are schematized

in the right scheme of figure 4.2. As stated before, chosing the combinations and building the data structure cannot be performed in parallel. This reduces the size of the parallel region and adds an extra overhead to each `ttDilepKinFit` call. This overhead is also increased by the best solution merge, which does not occur in the sequential application.

4.1.1 Implementation

The implementation process was iterative, where every major change is tested in terms of performance and correctness of the application output before proceeding to the next change. Since the application code is very complex has a heavy global state, small modifications, specially when introducing parallelism, can completly alter the application behavior and affect the results. The OpenMP library was used to implement the parallelization as it is the most adopted among scientists and easier to include in C/C++-like applications such as `ttH_dilep`. It offers many of the functionalities required by this parallelization strategy, such as parallel reductions, various workload schedulers and explicit thread management and synchronization primitives. Also, OpenMP allows for the number of threads to be defined by environment variables, without any changes to the code, which eases the user interaction without changing its input parameters.

Each step of the `ttDilepKinFit` workflow (see figure 4.2, left image) was individually parallelized to control the impact of the concurrent tasks on the overall behavior. Then, all the parallel regions were concatenated, resulting in the workflow presented in figure 4.2, right image. At this point the main concern was the correcteness of the application, rather than its performance.

The first task performed by `ttDilepKinFit` is selecting all possible combination. This sequential task can be performed simultaneously to building the data structure holding its information. The total amount of combinations, which depends on the amount of jets and leptons n, pairing two jets with two leptons, being $k = 4$, in the same combination regardless of their order, i.e. $(j_1, j_2) = (j_2, j_1)$, is, according to the formula for mathematical combinations, presented in equation 4.1. The average number of combinations for the input data file is 131.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \text{ with k} = 4 \text{ then } \binom{n}{4} = \frac{n!}{8(n-4)!} \tag{4.1}$$

All the information of the Bottom Quark jets and leptons (ROOT `TLorentzVector` class instances), as well as other control and auxiliary information is stored in a class built for this purpose. The function responsible for the variation of the parameters was implemented as a method of this class, increasing the modularity of the code. Each task creates a local copy of the combination to apply the variation and reconstruct. This keeps the integrity of the original combination parameters, allowing it to be varied any number of times. Otherwise, applying a variation to an already varied combination would result in inaccurate physics results.

The size of one data structure element is 2 kB, making an average data structure size of 262 kB per event. Even though the data structure size is not big, the overhead of its construction might prove to be a factor limiting the performance. Note that the number of variations to perform does not affect the size of the data structure.

One of the major problems of this parallelization is to control the accesses to the global state on the LipMiniAnalysis library. In `ttDilepKinFit`, 34 global variables of mostly ROOT classes instances and vectors, are read and writen inside the parallel region. The access to this variables can be serialized

but that would alter the behavior of concurrent tasks reconstructing different variations of the same combination. Further analysis of the source code reveals that even though these variables are global they are only modified by the `ttDilepKinFit` function and their purpose is only to store intermidiate results. The most efficient solution is to create a local copy of the global state in each thread (note that a thread contains one or more parallel tasks), removing any data dependencies and avoiding serial accesses that can cause contention on the shared resources and degrade the performance.

After reconstructing all variations of all combinations for an event only the best reconstruction is necessary. Each solution quality is measured by a scalar value, computed comparing the reconstructions to the theoretical model, and the higher the value the better the reconstruction. The best solution is a set of 16 `TLorentzVectorWFlags` class instances (from LipMiniAnalysis), an extension to the ROOT `TLorentzVector` class, and a scalar with the solution quality. A class was created holding all this information and implementing all comparator operations, which increases the implementation overhead but reduces the complexity of the reduction process. Since OpenMP only supports parallel reduction for scalar values, a custom parallel reduction was implemented. The reduction is not performed among all tasks but rather among the threads used; during the reconstructions, each thread automatically holds the information of only the best solution, which is then used in the reduction, diminishing the amount of elements to compare. After the reduction, the best solution information is copied to the global state.

The algorithm used for the reduction is simple. The threads are grouped two by two in each level of the reduction tree and their solutions are compared. The thread with the lower id keeps the best solution of its pair. Threads which do not have a pair check a queue of unpaired threads and pick one. If the queue is empty, they are put in there. An example reduction is presented in figure 4.4.

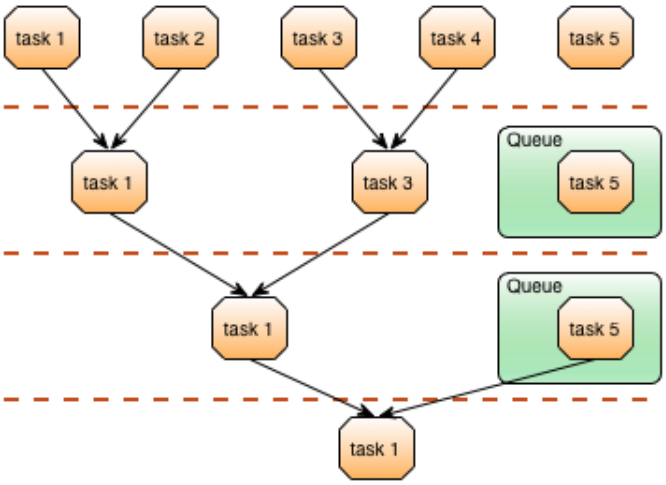


Figure 4.4: Schematic representation of a possible best solution reduction in `ttDilepKinFit`.

As mentioned in section 3.2.2, the pseudo-random number generator used for applying the variations is the `TRandom3` ROOT class. It uses the Mersenne Twister algorithm which heavily depends on a huge global state. Concurrent threads cannot use the same global state because one thread random number generation would influence the other thread generation, requiring serialization of the process. However, it is perfectly possible that different threads use different global states, eliminating the resource contention and correlations between random numbers. In this implementation, each thread has its own instance of the `Trandom3` class, ensuring that there are no global states shared and allowing for thread safe pseudo-random number generation.

OpenMP offers a set of different schedulers, from which the most relevant are the static and dynamic

schedulers. The static scheduler defines the number of parallel tasks that each thread will process prior to the parallel region execution. This scheduler has a small processing footprint and is very efficient for balancing regular workloads. For irregular workloads, such as `ttDilepKinFit`, the static scheduler porduces a poor workload balance, which may affect the performance. The dynamic scheduler requires more computational power but performs a better job at balancing irregular workloads. The scheduler monitors the execution load of each thread and maps the tasks to the threads at runtime. Parallel implementations performance using both schedulers are analyzed and compared in subsection 4.1.2.

Intel VTune tool was used to analyze and identify the bottlenecks of the parallel implementation. With the help of this profiler, the constructor of the data structure of the combinations was identified as the main performance bottleneck. When analyzing the data structure source code, its evident that some of its parameters are only read and not modified during each event processing. Instead of copying these parameters for each element of the data structure, it is possible to use a pointer to their memory position in the global state. The access to this data can be parallel as it is read-only. However, it may decrease the performance when using various CPUs, as threads in one CPU may have to access data in other CPU, causing NUMA accesses. The performance of this implementation, addressed as pointer version, is compared against the previous implementation, addressed as non-pointer version, in subsection 4.1.2.

4.1.2 Performance Analysis

The performance analysis of the various shared memory implementations will be presented in this section. Many metrics for evaluating performance will be used, such as speedup and throughput, always compared to the original application, and the results will be analyzed and discussed. The compute-711 system was used to conduct the tests in this section.

Figures 4.6 and 4.7 present the speedups for various number of threads, using one and two CPUs, with and without Multithreading[1]. The maximum theoretical speedup is also presented in figure 4.5, for comparing the efficiency of the parallelization with the already optimized original sequential application. It is expected that the overhead of creating the data structure and merging the results may slightly affect the performance, specially for a small number of variations.

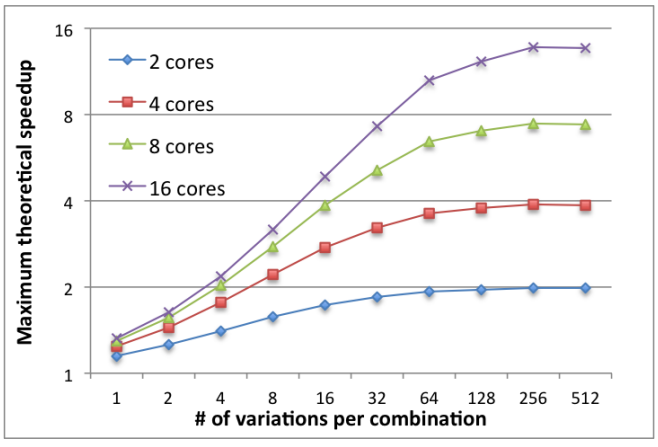


Figure 4.5: Theoretical speedup (Amdahl's Law) for various number of cores.

The maximum theoretical speedup calculation, presented in figure 4.5, is based on the Amdahl's Law [46] which states the maximum attainable speedup depends on the number of processors and the

[1]Refer to appendices and for test system and methodology characterization.

percentage of the execution time spent on the region to parallelize. The efficiency of the parallelization of an application can be measured by the difference of the speedup curve to the respective Amdahl's speedup curve. The Amdahl's Law was calculated for the original application with the initial optimizations to assess the efficiency of only the parallelization implementation.

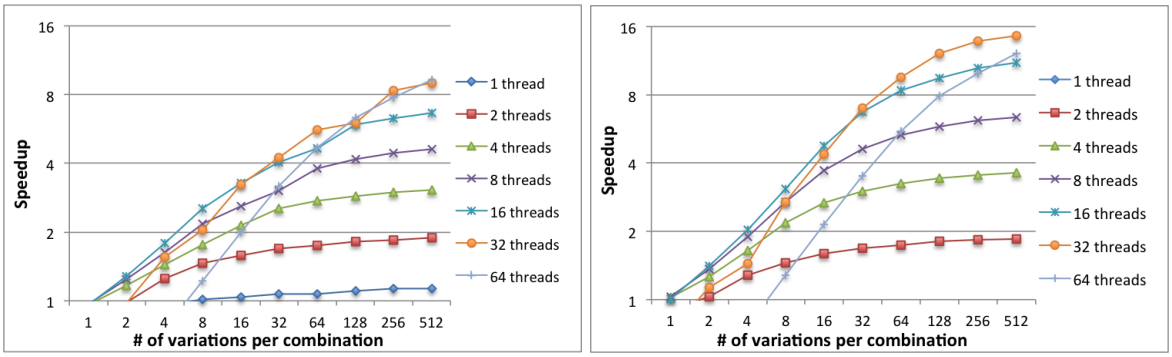


Figure 4.6: Speedup for the parallel non-pointer version of `ttH_dilep` application with static (left) and dynamic (right) scheduling.

Even with the performance increase provided by the optimizations in section 3.2.2, for a low number of variations per combination the overhead of the thread management, data structure creation and best solution merge is too high and causes the application to be slower than the original. With the increase of threads, the thread management overhead increases and the performance is even lower. Overall, the best results are obtained using the dynamic scheduler. The dynamic scheduler overhead is evident when looking for the results using 1 thread, where the overhead difference is only due to the schedulers: for 8 and more variations the static scheduler implementation offers speedups, while the dynamic scheduler implementation performance is always worse than the original application. Using all available cores (16 threads) the speedup is 6.7 and 9.2, for the static and dynamic scheduler versions respectively, for the best case of 512 variations. The use of hardware multithread (32 threads) benefits the performance in both cases, but the speedup is bigger for the dynamic scheduler implementation. It allows hiding the high latency of RAM memory acesses by scheduling threads which are ready to execute while others are idle waiting for the data. Multithreading managed by software (64 threads) offers speedups better than using 16 threads, but only high number of variations, such as 256 and 512. The best efficiency (speedups closest to the theoretical maximum) is obtained for 2, 4 and 32 threads of the dynamic scheduler implementation.

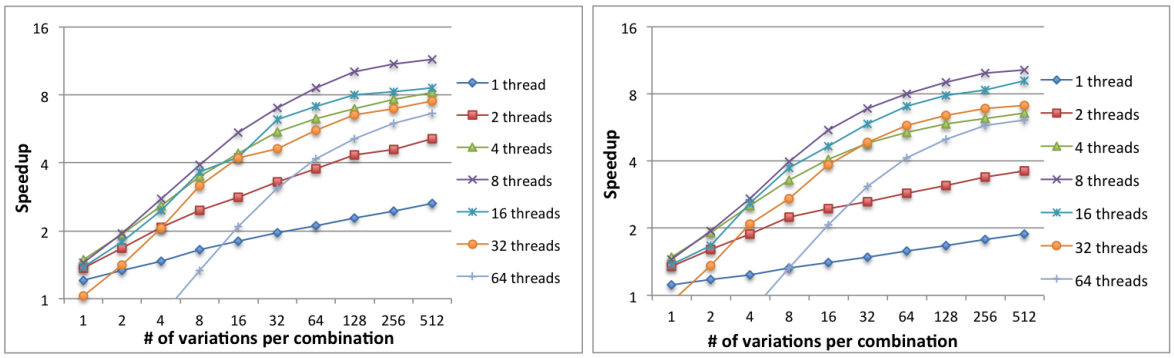


Figure 4.7: Speedup for the parallel pointer version of `ttH_dilep` application with static (left) and dynamic (right) scheduling.

Similar to the non-pointer version, the dynamic scheduling provides the best performance for high

number of threads in the pointer version. However, for 2, 4 and 8 threads the low overhead of the static scheduler gives this implementation an advantage against the dynamic scheduler version. When both CPUs are used, the performance of the static scheduler version degrades relatively to the dynamic scheduler. The difference between the overheads of the two schedulers is evident when comparing both implementations using only one thread. The most important results refer to the efficiency of the static scheduler implementation for 2, 4 and 8 threads, as they present superscalar speedup[2], mostly due to the pseudo-random number generation optimizations allied to the low overhead of constructing the data structure, opposed to the higher overhead of the non-pointer version. It was not possible to test the speedup of only one CPU using multithreading as the current implementation of OpenMP does not allow to manually match the threads to the cores, and multithreading is only enabled when all available physical cores are occupied.

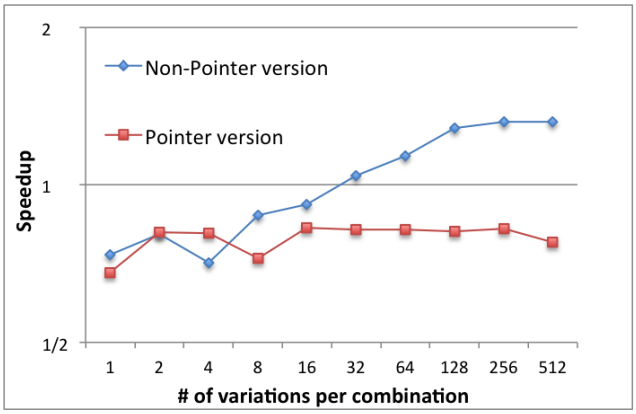


Figure 4.8: Speedup provided by using hardware multithreading for the non-pointer version.

The speedup provided by the use of hardware multithreading, presented inf figure 4.8, which helps to improve the CPU resource usage by scheduling multiple threads per core, is only noticeable for 32 or more variations in the non-pointer implementation. It provides a speedup up to 1.3 over using only 1 thread per core, by diminishing the impact of high latency RAM memory accesses. In the pointer implementation, hardware multithreading degrades performance due to the usage of pointers to shared memory, resulting in the increase of the latency of memory accesses proportional to the number of threads usedwhen using multiple CPUs.

Considering both dynamic scheduler non-pointer and static scheduler pointer implementations, which are the best performing, the non-pointer version offers the best speedup for 32 threads. However, the best efficiency is obtained by the pointer version for 2, 4 and 8 threads, due to the reduced overhead of constructing the data structure. It provides a speedup of 11.5 for 8 threads and 512 variations, only matched by the non-pointer implementation using 16, 32 and 64 threads. For higher number of threads the performance does not increase, which is an expected behavior since threads in different CPUs share a pointer to the same data. The non-unified memory accesses to shared memory degrade the performance as the data cannot be properly stored in the cache preventing a set of memory management optimizations. Chosing the best implementation depends on the system, since the non-pointer version is best for multi-CPU systems while the pointer version is best for single CPU systems. VTune did not identify more bottlenecks on the application code, not considering functions on the LipMiniAnalyisis library.

One possible metric to measure the increase of processing throughput in the `ttDilepKinFit` is the increase of kinematical reconstructions performed relatively to the sequential application. Only the

[2]Superscalar speedup occurs when the speedup is higher than the number of CPU cores used.

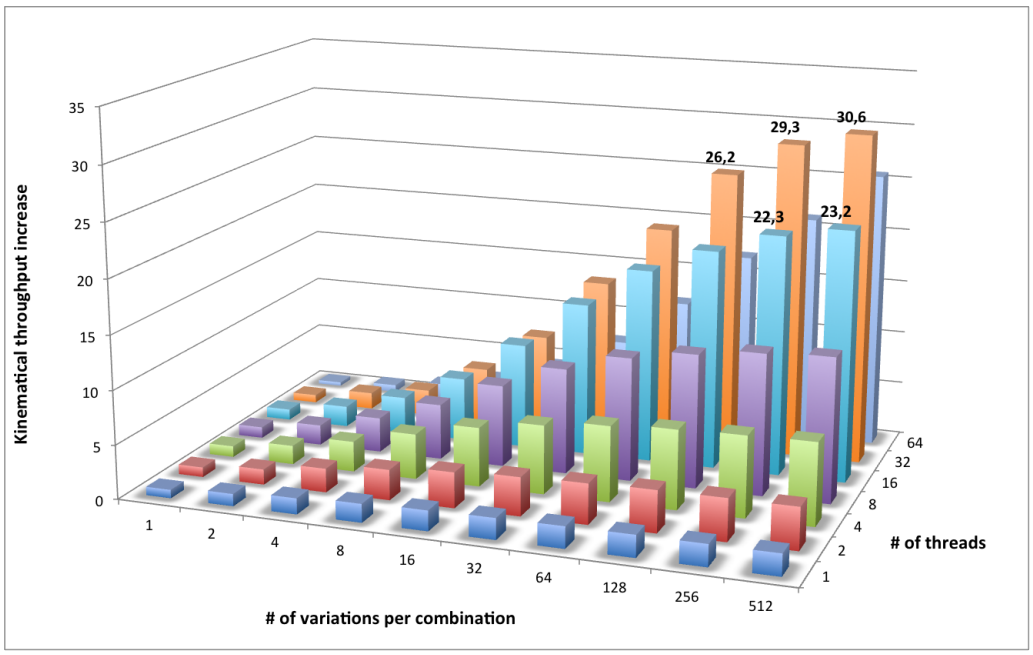


Figure 4.9: Kinematical reconstruction throughput for various number of variations and threads.

kinematical reconstruction is suitable for this purpose as it is always executed, opposed to the Higgs Boson reconstruction which is only performed if the $t\bar{t}$ system is reconstructed. The speedup of the kinematical reconstruction relatively to the original application is presented in figure 4.9. The non-pointer implementation performs up to 30 times more kinematical reconstructions than the original application in the same time, for 512 variations and 32 threads. This value does not translate directly into overall application speedup at this only considers a regular task inside the bigger and irregular `ttDilepKinFit`.

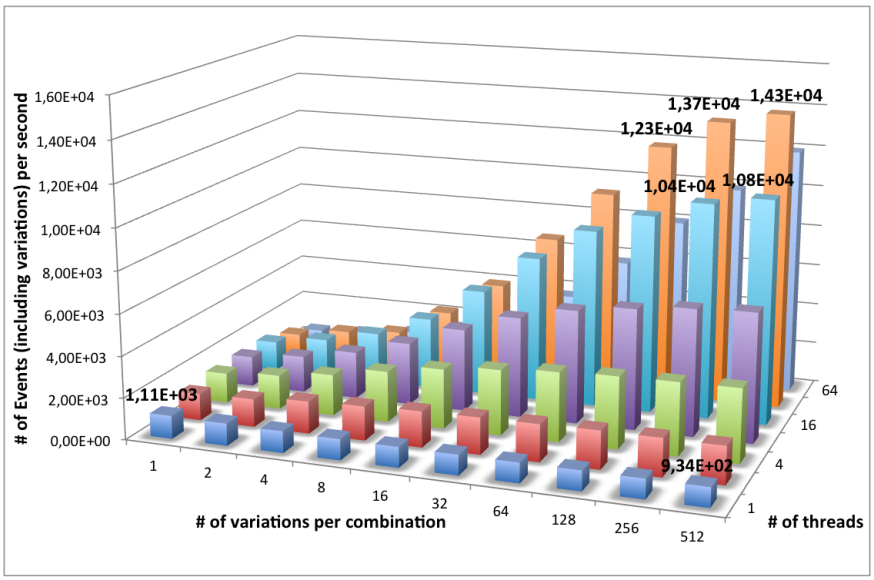


Figure 4.10: Event processing throughput for various number of variations and threads of the non-pointer version.

One of the concerns for research groups is the amount of events that their applications are able to process. Figures 4.10 and 4.11 present the event throughput for the non$_p$*ointerandpointerversions*, *respectively.Theev*
pointerversion, *whichis*15*timeshigherthantheoriginalforthesamenumberofvariations.However*, *thepointerver*

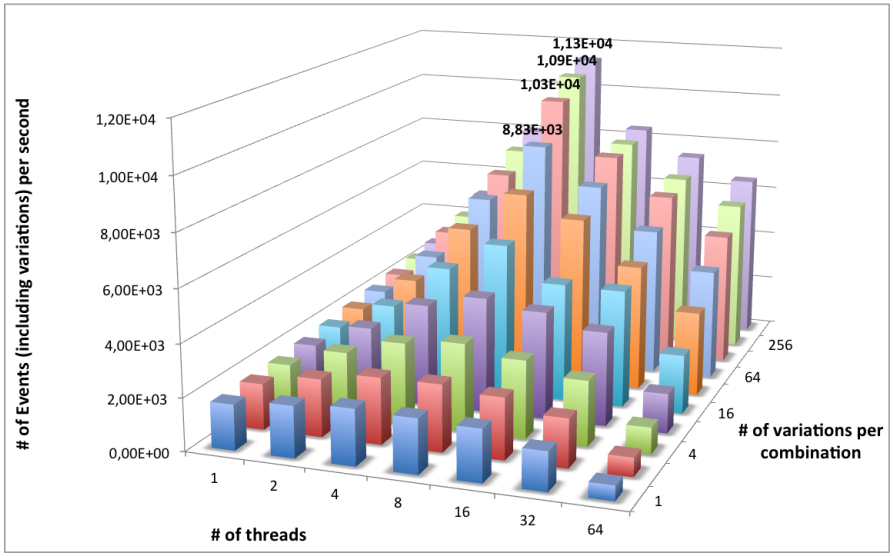


Figure 4.11: Event processing throughput for various number of variations and threads of the pointer version.

Performance analysis on various computational systems

Scientific computer clusters are not always made of high end computing nodes, such as the compute-711 test system used in this chapter. A simpler performance analysis on 3 dual-socket test systems common among research groups is presented in this subsection. Only the speedup and execution time will be compared, as an in-depth analysis was already made in section 4.1.2.

Only the two best implementations, non-pointer with static scheduling and pointer with dynamic scheduling, are tested for three different number of threads: one per core using one CPU; one per core using both CPUs; one per hardware thread if hardware multithreading is supported. The test systems used are the compute-401, compute-511 and compute-601 nodes of the SeARCH cluster[3].

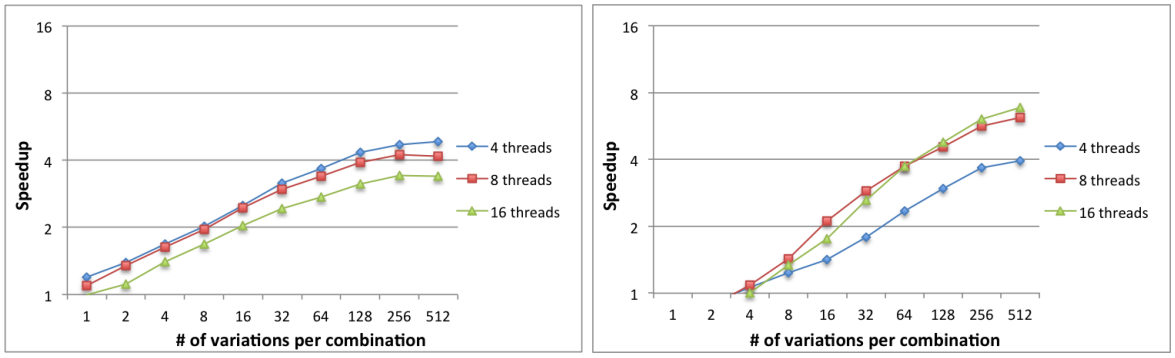


Figure 4.12: Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-401 node.

Much like the compute-711, the best performance is obtained on the non-pointer version for 16 threads (using all multithreaded cores). However, the multithreading gains are not as significative as with the the compute-711 system. The best efficiency is obtained using 4 threads, the entire cores of one CPU, for the pointer version, attaining a speedup of 4.8 for 512 threads.

The speedups for the compute-511, in figure 4.13, are very similar in the two versions, with a slight advantage for the pointer version. However, for 24 threads it is far from the peak performance as the

[3]See appendix for the characterization of all test systems.

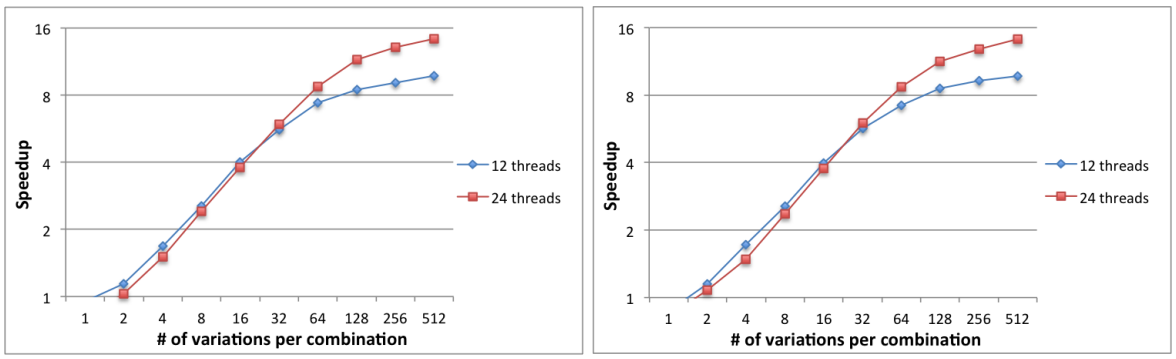


Figure 4.13: Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-511 node.

system as a total of 24 physical cores. This is due to the poor memory management characteristic of AMD CPUs, which is specially evident in a NUMA accesses. Even though the speedup is quite good for 12 threads, the execution time is the worse of all the systems, as presented in figure 4.15.

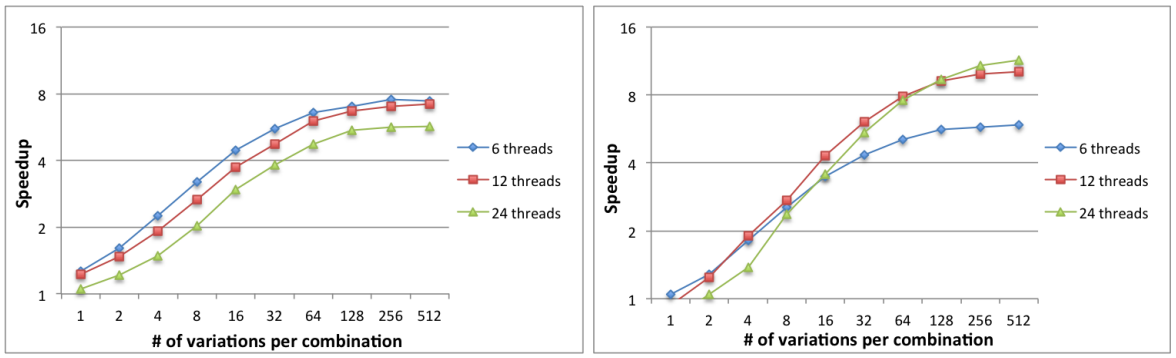


Figure 4.14: Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-601.

The compute-601 system behavior is also similar to the compute-711. The pointer version offers the best, and superscalar, speedups when using only one CPU with 6 threads, but for a higher number of threads the speedups are always worse. However, the best performance is achieved by the non-pointer version for 24 threads, but only for 256 and 512 variations. For any other amount of variations the best is to use 12 threads.

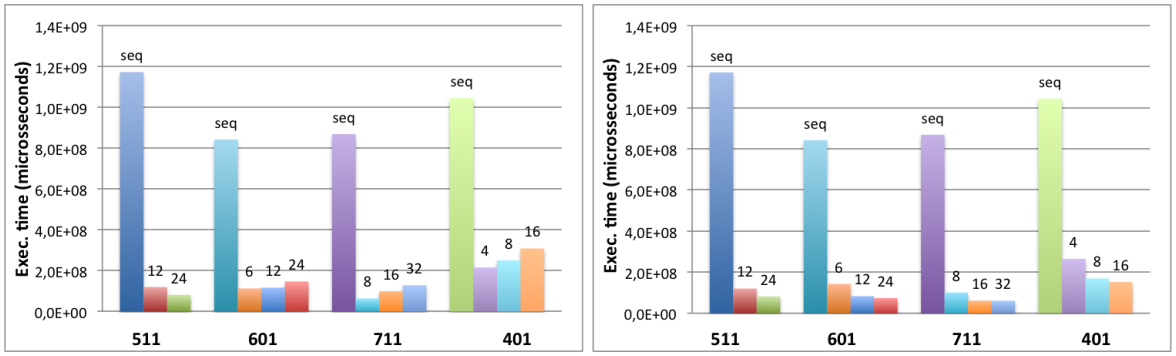


Figure 4.15: Execution times of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations for 512 variations.

Figure 4.15 presents the execution time for all test systems. For each system, the bars represent, from left to right, the original application, parallel with number of threads equal to the number of cores in one CPU, number of threads equal to the total number of physical cores and number of threads equal to the total number of cores with multithreading, if available. For the pointer version, the best system is the compute-711 for 8 threads. The compute-511 system is the second best for 24 threads. However, it presents the worst efficiency as it has more physical cores than any other system. The best overall results occur when using the non-pointer version with all available hardware threads for the compute-711. Note that the compute-711 is the most recent high end system of all tested. The slower system is the compute-401, which is also the oldest and with fewer cores and smaller memory bandwidth. All the systems present the same behavior: for the pointer version, the execution time increases when using more than one CPU, due to the hazardous non unified memory accesses; for the non-pointer version, the execution time diminishes with the increase of threads, with hardware multithreading slightly increasing the performance due to better resource usage.

4.2 GPU Parallelization

Programming for GPUs still presents a series of important limitations imposed by the drivers due to the hardware characteristics. It uses C language for programming and it is not possible to use C++ classes. Also, libraries must be programmed specifically for the hardware accelerator, which prevents the use of any functions of LipMiniAnalysis and ROOT frameworks. This restricts the region of `ttDilepKinFit` suitable to parallelize on GPU due to many dependencies on ROOT functions and classes.

The variation computation and kinematical reconstruction uses the `TLorentzVector` class from ROOT for holding the Bottom Quarks and leptons information. They have the particularity of always being executed, unlike the Higgs Boson reconstruction. However, it is possible to eliminate these dependencies, but causing an additional overhead. The other parts of `ttDilepKinFit` are heavely dependent on ROOT and are not possible to port to GPU. Considering these factors, a parallel workflow was devised and is presented in figure 4.16. The implementation details are presented in subsection 4.2.1.

Similar to the shared memory parallelization, selecting all the combinations of Bottom Quark jets and leptons is a serial task. Then, a process of data marshalling is applied to transform all the information on the `TLorentzVector` classes into standard arrays. These arrays must be copied to the GPU memory so that the variation can be applied. The Mersenne Twister implementation available in the NVidia cuRand library is used as the pseudo-random number generator (already discussed in section 3.2.2). Then, the kinematical reconstruction is performed and the results are transferred back to the CPU memory. In the CPU, the data unmarshalling, i.e., transforming the results from the arrays into the ROOT classes to be used by the rest of the application, is performed and the Higgs Boson reconstruction is parallelized on the CPU.

There are several factors which will restrict the performance. The data marshalling and unmarshalling will add a sginificant overhead to `ttDilepKinFit`, but it cannot be avoided. Even though the amount of data to transfer is small, with a similar size to the data structure used in the shared memory implementation, it occurs for every event and the kernel execution can only begin after all the data is copied. Assynchronous memory transfers can be used but they will not benefit the performance as it forces blocks of threads to run serialized to others, limited by the combinations that arrive to the GPU memory. It would increase the performance if all the threads would work on the data as it arrives, but that is not the case with this algorithm. The kernel itself is very complex with the variations and kinematical reconstruc-

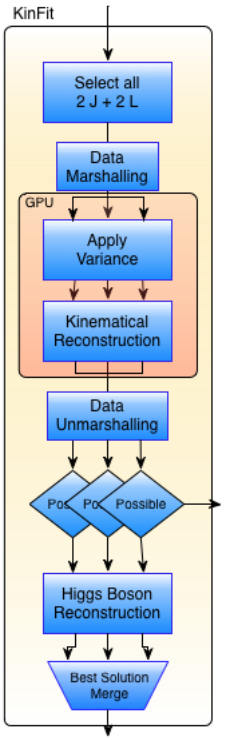


Figure 4.16: Schematic representation of the `ttDilepKinFit` workflow on GPU.

tion. The computations are not homogeneous, with many operations performed in different data, which will result in a very high resource usage (namely registers and L1 cache). Register spilling might limit the performance as high register usage is an inherent characteristic to the kernel and cannot be optimized. Finally, the data structure used in the shared memory implementation is also present, as it is used to store all the combinations, so that they can be marshalled and transferred to the GPU memory, and is later used in the parallel Higgs Boson reconstruction. The best solution merge is also a required step.

During the GPU execution, the CPU is idle waiting for the results, and vice-versa. This inefficient usage of the computational resources is caused by the inexistence of a data structure holding all the event information. Figure 4.17 presents the current and optimal execution flows between CPU and GPU. The optimal flow could be implemented if the event data was loaded to a data structure, rather than overwritting all information on memory. This would require a restructure of the application and LipMiniAnalysis, which is not possible in the timeframe of this dissertation work, but could render the usage of hardware accelerators a valid option for event processing.

4.2.1 Implementation

The data marshalling is performed on the data structure built when selecting the combinations, using the same process as with the shared memory parallelization implementation. On the variation computation and kinematical reconstruction, the most important data is the energy, mass and momentums of the Bottom Quark jets and leptons on the combination. The energy and momentums are double precision float point values of the `TLorentzVector` class, while the mass is computed based on those values. The mass function was copied from the ROOT source code to be implemented on GPU. The rest of the information is copied to various arrays, one per each particle. Other data, such as 4 `TLorentzVectorWFlags` class instances per event, from the LipMiniAnalysis library, and other control information (scalars) are also copied to specific arrays.

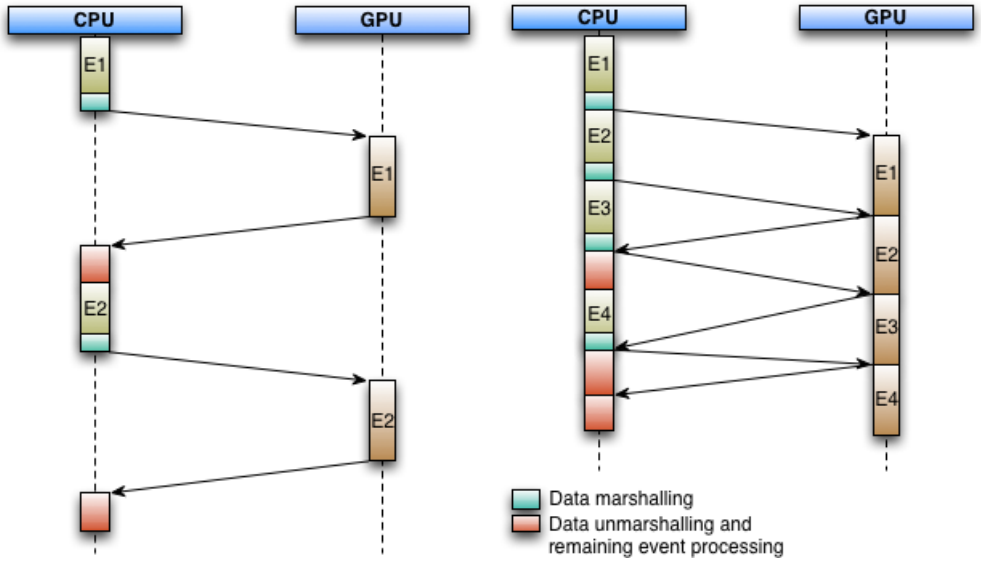


Figure 4.17: Execution flows for the implemented workflow (left) and optimum workflow (right) of the `ttH_dilep` application.

The variation computation uses the Mersenne Twister algorithm available in the NVidia cuRand library [41], which produces uniformly distributed pseudo-random numbers. The transformation algorithm used in the ROOT `TRandom3` class, to map the values from a uniform to gaussian distribution, was ported from the ROOT source code to the GPU. The objective is to keep the changes in the `ttDilepKinFit` algorithms to a minimum. One precomputed state was used per block of threads for the Mersenne Twister algorithm, which allows up to 256 threads to simultaneously generate pseudo-random numbers. The required numbers are generated at the beginning of the variation function, so that all the calls to the generator occur simultaneously, increasing the parallelism which benefits this kind of hardware accelerators.

Both the variation computation and kinematical reconstruction source code was modified to work on the input arrays, rather than resorting to ROOT classes. The results of the kinematical reconstruction are also stored in an array. However, since the kinematical reconstruction can return none, two or four instances of the `TLorentzVector` class, depending on the solution found, it is necessary an additional array to store the output size of each thread. Without this information, solutions can be assigned to the wrong combinations during the data unmarshalling. The varied parameters are also copied to the CPU memory and the data structure is updated.

The parallelization of the Higgs Boson reconstruction was implemented using OpenMP, with the dynamic scheduler as the workload is irregular since some variations of the combinations might not be reconstructable. The best solution merge is similar to the one presented in section 4.1.1.

The number of threads per block is limited to 256 due to the pseudo-random number generator restraints. The best block size for a kernel can be calculated using the NVidia CUDA Occupancy Calculator [47], and depends on the number of registers that the kernel uses, the GPU architecture and cache size. The result of the calculation is presented in figure 4.18, which indicates that the best configuration uses 256 CUDA threads per block. The number of CUDA threads is equal to the number of variations times combinations, which varies between events, and, if possible, the threads may be organized in blocks of 256.

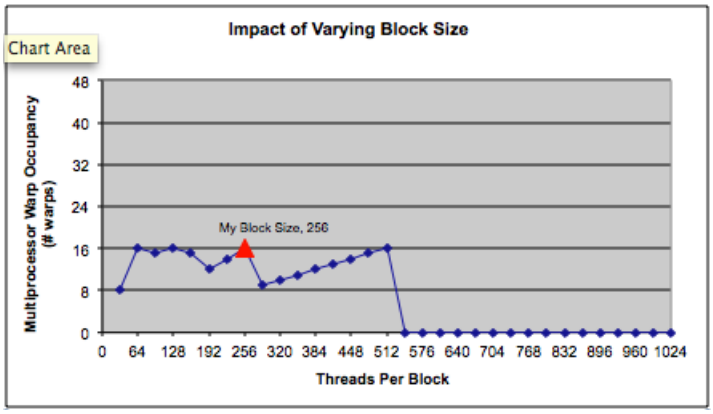


Figure 4.18: Optimal number of threads for the GPU kernel, according to the NVidia CUDA Occupancy Calculator.

4.2.2 Performance Analysis

The GPU implementation was tested in the compute-511 system with a NVidia Tesla C2050[4]. The focus is on evaluating the limiting factors as it was not expected great performance due to the details already referred in 4.2.

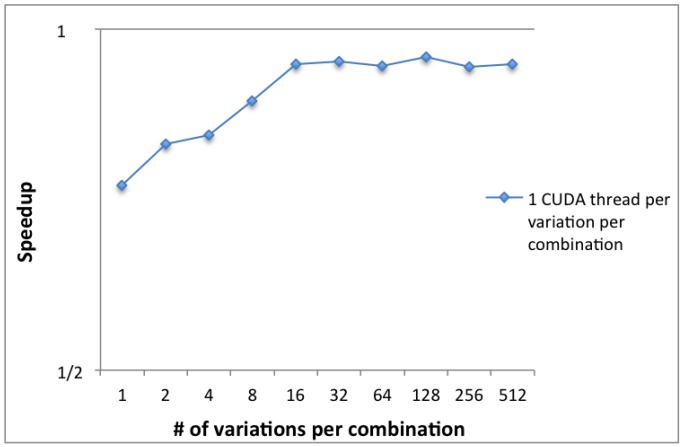


Figure 4.19: Speedup of the `ttH_dilep` application for the GPU parallel implementation.

Figure 4.19 illustrates the speedup for this implementation. For all different variations tested, the implementation is always slower than the original application. All the data (un)marshalling, transfer and unefficient usage of resources greatly restricts the potential performance gains.

Figure 4.20 shows the relative execution time of the application that is spent on the GPU and on the CPU. Since the CPU and GPU cannot work simultaneously, the graph also shows the percentage of time that each of the resources is idle. On average, the GPU is idle 70% of the time, wasting important computational resources. Before optimizing memory transfers or the kernel itself, the low GPU usage is the main bottleck in which most efforts must be focused.

Even though the speedups do not motivate any efforts on optimizing this implementation, the low GPU usage is due to the global event data and a restructuration of the application, and LipMiniAnalysis library, would greatly improve the implementation. Introducing a data structure holding all the event information would allow the workflow presented in the right image of figure 4.17 to be implemented and

[4]See appendix for more hardware details.

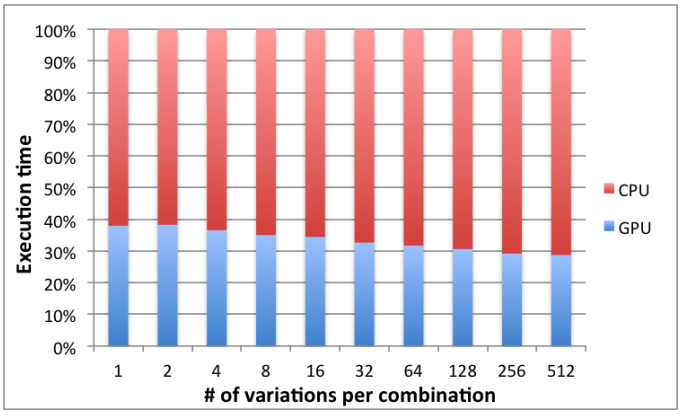


Figure 4.20: Relative execution time of `ttH_dilep` application on CPU and GPU.

explore all the potential performance increase offered by the GPU.

4.3 MIC Parallelization

The Intel Xeon Phi has two operating modes. The native mode is supposed to run all the application on the device, requiring all code to be compiled with the Intel compiler and specific flags. In this mode, the device reserves one core to run the operating system, and it is required to copy all libraries, binaries and input files to the Xeon Phi memory prior to execution. In the offload mode the CPU operates the Xeon Phi as an accelerator, similar to GPUs. The objective is to do preliminary tests comparing both operating modes.

Since the native Xeon Phi execution model is similar to a multicore CPU, the workflow of the shared memory implementation is suitable for the problem. In theory, no changes to the shared memory implementation would be necessary to run on the device. Figure 4.21 illustrates the workflows for the native and offload device operative modes.

For the offload mode, the implementation will suffer from the same problems as the GPU implementation. For each event, it is required to copy the data structure holding all the combination information. Even though it is possible to use C++ classes on the accelerator, it is not possible to have a library simultaneously compiled for the CPU and Xeon Phi without modfying the source code. However, that is not possible to do in the ROOT and LipMiniAnalysis libraries within the timeframe of the dissertation work. This restricts the region of the application to parallelize on the Xeon Phi to only the variations computation and kinematical reconstruction. Also, by not using the ROOT classes, the same data (un)marshalling processes used in the GPU implementation are required. The Higgs Boson reconstruction is performed in parallel on CPU.

The resource usage efficiency will be similar to the GPU implementation, where the CPU is idle when the Xeon Phi is computing, and vice-versa. The purpose is to evaluate the usability of the accelerator, as Intel claims that porting (unefficient) code to the device is straight forward.

4.3.1 Implementation

The implementation for the Xeon Phi native operating mode requires the compilation of the ROOT library, since both LipMiniAnalysis and the application depends on it. ROOT offers specific compilation for Intel Xeon Phi devices, developed by CERN OpenLab. However, it does not work on the compute-

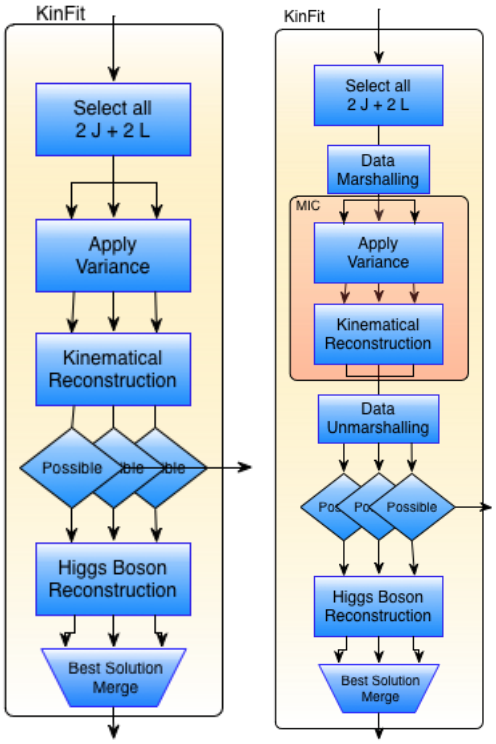


Figure 4.21: Workflows for the native (left) and offload implementation (right) on the Intel Xeon Phi.

711 test system. Much of the development time for this implementation was spent trying to solve the bug on the many ROOT makefiles. However, since the device is much simpler than the CPU and only runs at 1 GHz, it would not offer a significant performance gain relatively to the CPU shared memory implementation, even with the larger throughput of the device. Therefore, this implementation was left on hold to be continued later on.

The parallel GPU implementation was adapted to run on the Xeon Phi in offload mode, with requiring many modifications modifications. Where each CUDA thread would process one variation per combination, on the Xeon Phi these tasks are grouped to match the device threads. The data (un)marshalling is similar to the GPU implementation and the data transfers are implemented using compiler pragma directives. These directives offer a simple way to implement the data transfers but only work with standard C structures, such as arrays.

Even though the Xeon Phi was already tested with static benchmarks and relatively simple algorithms, the drivers offered huge limitations for the implementation of such complex algorithms. During the preliminary implementation, the drivers returned many different errors when offloading the functions that did not even got a description. Intel support advised to look into headers of the respective sections of the drivers to help identify the errors source. The implementation was stalled by a timeout error from the driver that was left unsolved, even though the device was properly installed and was fully working according to the Intel control tools.

Both native and offload implementations were left on hold since it was for the LIP research group best interest to focus on the shared memory and application scheduler implementations.

4.4 Scheduler Parallelization

The idea for an application scheduler derived from the high performance and efficiency delivered by the shared memory pointer implementation running on a single CPU. Huge amounts of eventy data files, which usually have 1 GB size, need to be processed everytime a new batch arrives from the Tier-2 computational centers. This scheduler is an application that manages the execution of those files among a given amount of the same, or different, applications. As an example, instead of using one application to execute with 16 threads on the compute-711 system (consider the shared memory implementation on section 4.1), two applications with 8 threads of the pointer version could run simultaneously, providing better resource usage and process the same amount of data in less time.

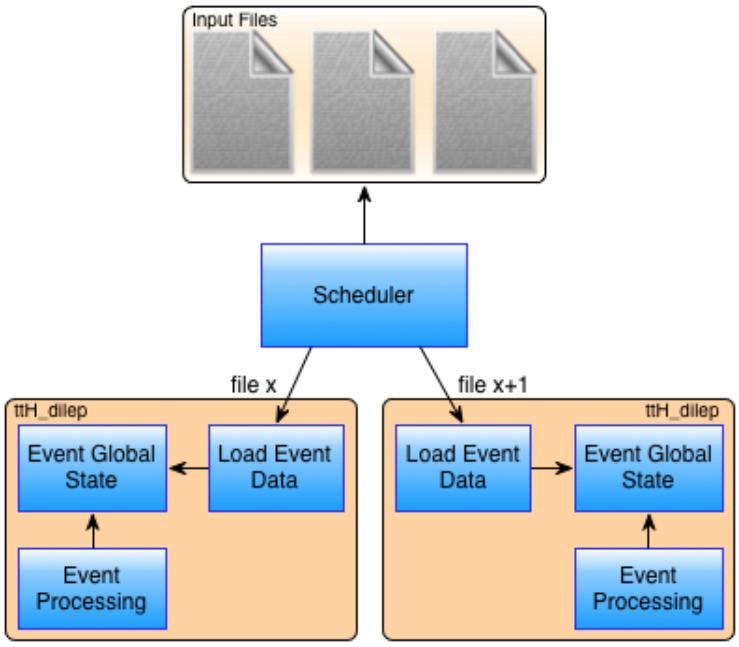


Figure 4.22: Schematic representation of the application scheduler.

Figure 4.22 presents the schematic representation for the proposed scheduler workflow designed for a shared memory environment. Although it will be tested with the parallel shared memory implementation (pointer version, due to its efficiency) of the `ttH_dilep` application, it is designed to virtually work with any kind of application and input data files. The scheduler is tuned and configured by the following set of parameters:

Application: the name of the binary file of the application, or applications, to execute.

Input parameters: the input parameters of the applications to execute, without the input data file specified. Multiple parameters are required if different applications are used.

Input data files: range of the input data files to process. They are usually numbered (for example sample_001 to sample_500). Multiple files can be specified if different applications are used.

Number of applications: the maximum number of applications running simultaneously on the system.

Threads: the number of threads to run per application. A standard way to set the threads must be implemented by the application. If it uses OpenMP, it is possible to do so by setting environment variables.

The amount of applications to run simultaneously, and the number of threads per each application, will directly affect the performance of the scheduler. If not properly configured, it can lead to resource starvation, in the case that the total number of threads/applications is not enough to use all available resources, or overwhelm the system resources, where a high number of threads is used. At the moment, the best configuration can only be obtained by benchmarking the application and different configurations for the scheduler on the specific system to test.

The purpose of the scheduler is to provide a simple way of increasing the usage of the available computational resources to be without requiring the know-how of the underlying concepts of parallel programming. For the scheduler to be adopted by the physicists it needs to be easy to interact with and offer a considerable performance increase. A proper scheduling technic would require the use of micro-benchmarks, with similar characteristics of the applications, to assess the best configuration parameters. It would be necessary to study the different applications characteristics and chose adequate micro-benchmarks for each situation, allowing the scheduler to provide the best configuration for the system. Since this is only one of many proposed parallelization strategies, there was only time to develop a prototype, which needs to be manual tuning and configuration for each system.

Each input data file has a set of different events, meaning that the execution of the same application with different files is irregular. This irregularity benefits the usage of a dynamic load balancing strategy rather than a static, if the overhead of the first is low enough to not restrict the performance. An application execution is never less than a few seconds, providing a coarse task granularity. This means that the overhead induced by using a dynamic load balance strategy is not relevant compared to the time that takes to process a parallel task.

4.4.1 Implementation

Even though the application scheduler was designed considering many of features presented in section 4.4, it was only possible to develop, within the dissertation timeframe, a prototype to serve as a proof of concept. The scheduler is fully operational but with some limitations, later explained, needing more features to be implemented and refined before being ready for physicists to use.

The current prototype reads the configurable parameters from a text file (an example file is presented in 4.23). This file is read at the beginning of the scheduler execution and will affect its behavior and performance. The prototype is only prepared to read and parse the configuration file for running one application with the same set of parameters, but with different input data files.

The file structure is simple and intuitive but it was only designed for applications similar to `ttH_dilep`. The order of the parameters is irrelevant as long as they are properly identified. *bin* and *parameters* refer to the name of the binary file of the application and its parameters, without the flag specifying the file to process. The files are indicated in the *files* field, where they can be enumerated or use the *to* connector that indicates all the files between the files *900* and *950*. The *file_dir* and *file_flag* field indicate the directory in which the files are stored and the flag used by the application to specify the file to process, respectively. *number_of_bins* and *number_of_threads* indicate the amount of applications to run simultaneously and the number of threads per application. In the specific case of this application it is possible to define by an environment variable the number of variations per combination.

A data structure is created holding the information for each application execution (considered as an element of the structure), which size is equal to the number of input data files to process. The string with the application parameters is concatenated with the input data file flag and name and stored in the structure. Also, each element of the structure has the number of threads associated. This allows the

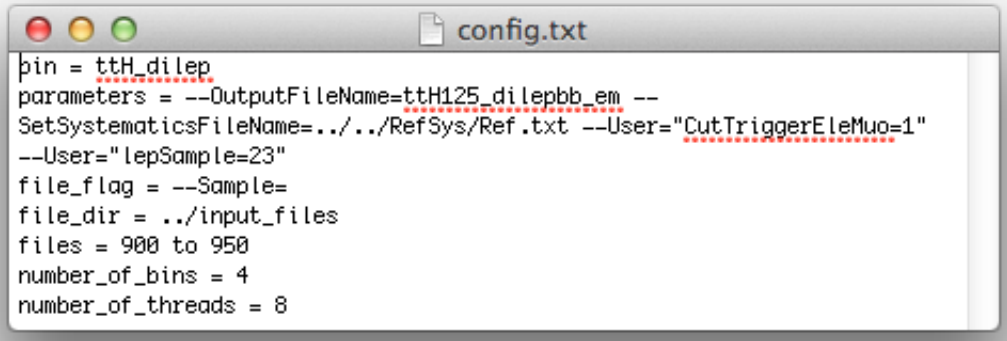


Figure 4.23: Example scheduler configuration file.

scheduler to be abstract enough to work with different applications with irregular parameters and different number of threads. The number of threads of the application is set through environment variables, when it is prepared to be executed, as the parallel `ttH_dilep` application is implemented using OpenMP and it allows set the threads in this way.

OpenMP was used to parallelized the applications execution. The number of threads is equal to the number of applications to run simultaneously, as each thread is responsible for the its application setup and execution. The threads operate in a work stealing pattern, getting hold of an element of the scheduler data structure and processing it, until the data structure is swept. The OpenMP dynamic scheduler is used as it is designed to efficiently handle this kind of irregular load and its overhead is not significant compared to each application execution time.

The implementation of micro-benchmarks to infer the best configuration of the scheduler would depend on the computational characteristics of the applications to manage. The tests performed in subsection 4.4.2 are based on the parallel `ttH_dilep` performance analyzed on section 4.1.2, but this might not valid for other test systems and applications. The benchmarks could be synthetic algorithms designed to evaluate the system performance, which need to be adequate to this type of applications, or the application itself running with one of the input data files, but it would be encessary to guarantee that its execution time is low. The latter alternative would induce an higher overhead but could provide a more accurate evaluation of the test system.

Little changes to the configuration file would be necessary to consider multiple different applications. Its structure is modular, so, to insert a different application, the fields in figure 4.23 would be replicated. This would also require little changes to the current parser on the scheduler and the data structure is already prepared to deal with any kind of application.

4.4.2 Performance Analysis

The performance analysis of the scheduler prototype is presented in this section. All tests are performed on the compute-711 system using the `ttH_dilep` application, with a set of 10 input data files and various numbers of variations per combination. The scheduler runs the most efficient parallel implementation of `ttH_dilep`, the pointer version, avoiding the use of NUMA accesses as each application will have

enough cores in one CPU to process its threads. Different combinations of number of simultaneous applications and threads per application are compared agains the original `ttH_dilep` and non-pointer parallel implementation using the same total number of threads (for example, 4 applications with 4 threads *vs* single application wiht 16 threads), as it is the fastest using both CPUs on the system.

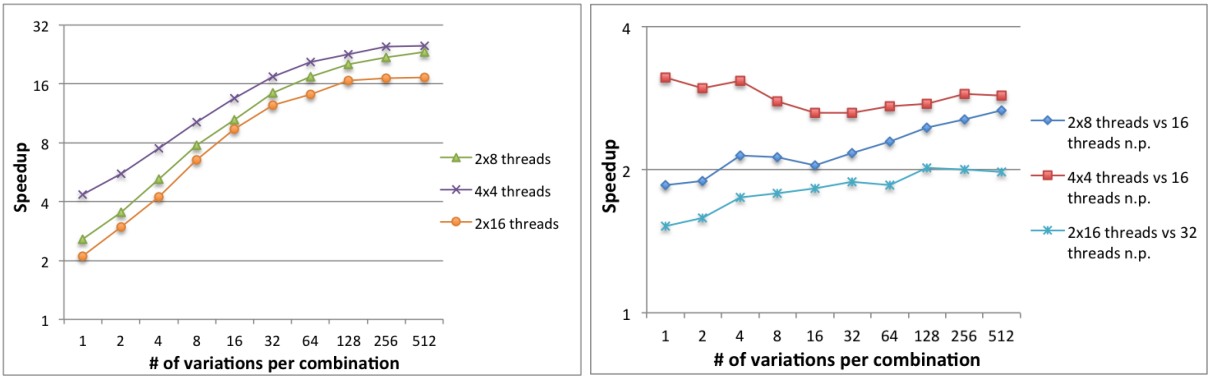


Figure 4.24: Speedup of the scheduler with different setups (simultaneous applications x number of threads per application) versus the original application and the non-pointer (n.p.) parallel implementation.

Figure 4.24 presents the speedup of the scheduler for three setups compared to the original sequential application (left graph). The scheduler performance using 4 applications with 4 threads each (*4x4*) and 2 applications with 8 threads (*2x8*) is very similar, with a slight advantage for the first setup. It achieves a maximum speedup of 25 for 512 variations per combination, while the best performance achieved by the non-pointer parallel implementation, presented in section 4.1.2, has a speedup of 14.6. Note that the speedup is higher than the number of cores because of the PRNG optimizations are not implemented in the original application. The scheduler transposes the efficiency of the parallel pointer implementation on a single CPU to a dual-socket system. The advantage of the *4x4* setup is due to the higher number of applications for the scheduler to manage, allowing for a better load distribution. The worst performance occurs for the *2x16* setup, with all the hardware threads are used, plus 2 other threads are require by the scheduler to manage the applications execution. The use of hardware multithread does not benefit the parallel pointer implementation due to the use of pointers to shared data.

The graph on the right of figure 4.24 presents the speedup of the 3 setups of the scheduler compared to the non-pointer parallel implementation for the same total number of threads. The best performance increase occurs for the *4x4* setup, with a relatively constant speedup of 3 for all different variations. For the *2x8* setup, the speedup increases with the number of variations, with similar behavior for the *2x16* setup. Overall, the scheduler with the parallel pointer implementation offers better performance and, consequently, more efficient resource usage than the best implementation for shared memory.

The highest event throughput, presented in figure 4.25, occurs for the *4x4* setup, as expected from the observed speedups, with 27700 events processed per second for 512 variations. The *2x8* setup has a peak throughput of 25800 events per second, close to the *4x4* setup. The highest throughput for a single parallel implementation was 14300 events per second, presented in figure 4.10. The scheduler provides almost a two-fold increase in event processing, presenting itself as the most viable option for processing in a dual-socket system.

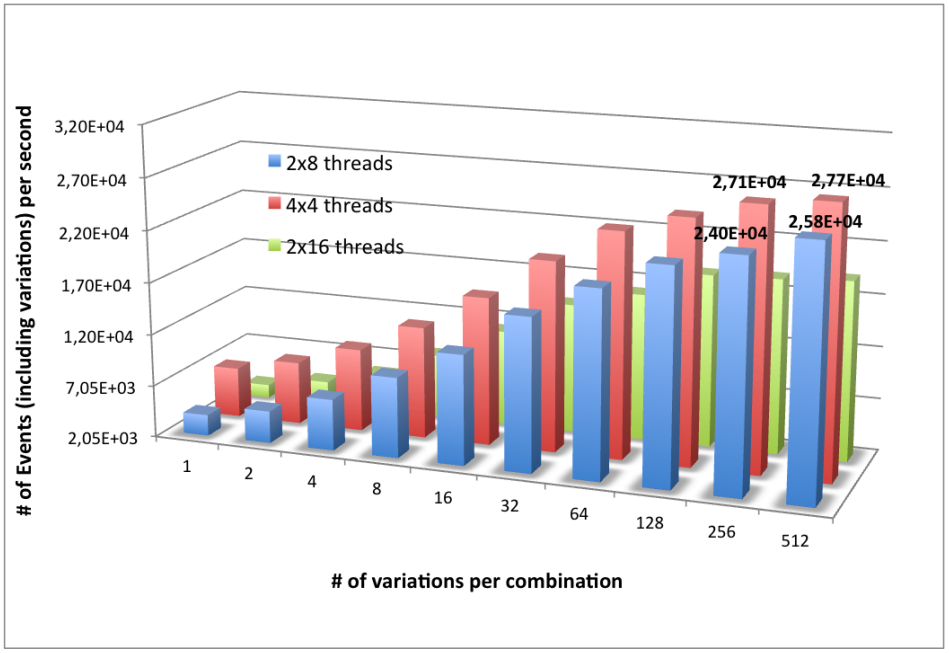


Figure 4.25: Event processing throughput for various number of variations and different scheduler setups.

Chapter 5

Conclusions & Future Work

This chapter concludes the dissertation, presenting an overview of the results obtained by the work developed, on both homogeneous and heterogeneous systems. Guidelines for future work, on improving the test case application and providing parallel solutions abstracted from the programmer for future application development, are presented.

5.1 Conclusions

This dissertation presents 4 different approaches to increase the performance of a scientific application that processes event data gathered at ATLAS, developed by the LIP physics research group, using homogeneous and heterogeneous systems, testing in the latter 2 different hardware accelerators. The objective is to evaluate the usability, performance and efficiency of the different systems and specific accelerators, as well as the quality and structure of the application and libraries.

The scientific application attempts to reconstruct the Top Quarks and Higgs Boson that decay from a head-on collision of two protons, based on the resultant data collected in the ATLAS particle detector at the LHC. The quality of the reconstruction depends on the amount of partial reconstructions performed, with the event parameters slightly varied, with the purpose of overcoming the experimental resolution of the particle detector and find the most accurate final reconstruction. However, increasing the number of partial reconstructions directly affects the application execution time, decreasing the number of events processed and impacting the researched conducted by LIP.

An initial optimization of the pseudo-random number generator, in which the application relies heavely, is proposed. Then, 4 parallel implementation solutions were developed, based on an identification of the critical region restricting the performance. Two implementations target heterogeneous systems with accelerator devices, one using an NVidia GPU and other using the Intel Xeon Phi. Both implementations face many limitations, primarly due to the abscence of a data structure holding all the event information on both the application and LipMiniAnalysis, restricting the performance gains and efficient usage of the GPU. Also, due to this hardware limitations, only part of the application critical region was parallelized on the accelerator, increasing the amount of high latency memory transfers between CPU and GPU, and forcing the rest to be parallelized on CPU.

The parallelization on the Intel Xeon Phi had the purpose of evaluating its performance, but more importantly, its usability, as Intel advertises easy porting of CPU code to the device. The objective was to test the Xeon Phi in both native and offload modes. One of the application dependencies, ROOT library developed by CERN, is supposed to have support for this device but, it did not work and many hours were spent trying to solve the problem. Using the device in offload mode, similar to the GPU, would require

that only part of the critical region is parallelized, as it depends on many functionalities of ROOT that cannot be simultaneously compiled for CPU and Xeon Phi. Its resources would also be inefficiently used due to the lack of a global data structure with all events. Even though the parallelization implementation is simplified by the use of pragma directives available for this device, the driver offered many limitations due to the computational complexity of the portion of the code to parallelize. The Xeon Phi is still new to the market, causing its drivers and libraries to not be completely stable. Many of the errors provided by the drivers were not even documented yet. Due to the dissertation timeframe, both implementations were left on hold, with the only conclusion that the device is not mature enough to provide an easy porting of the code for real applications, opposed to Intel claims.

The remaining 2 parallel implementations were aimed towards homogeneous systems, which constitute all of the LIP computational resources. The first was a shared memory implementation, from which two versions derived. One has a smaller parallelization overhead and better performance on single-socket systems, while the second, with a bigger overhead, but performing better on multi-socket systems. The second implementation is a application scheduler, oriented for multi-socket systems. The idea was based on the efficient use of computational resources of the first version of the shared memory implementation on a single CPU. Multiple instances of this application, processing different input data files, would provide an efficient use of all available computational resources and an even bigger increase in performance, compared to the shared memory implementations. A prototype was developed and tested with promissing results, but its performance is still manually tuned by a set of parameters which may vary from system to system. The scheduler is prepared to work with any kind of scientific application, only requiring that multithreaded applications use a given API to be configured.

The lack of structure on the code of both the application and LipMiniAnalysis library restricted the performance on heterogeneous systems. The best performance was attained for a shared memory implementation on homogeneous systems and, allied to the use of a prototype scheduler, provided an efficient use of the available resources. The LIP research group has the best interest in these implementations as they increase their productivity and do not require investment in expensive hardware accelerators.

5.2 Future Work

The results obtained in this dissertation, specially for the implementations on heterogeneous systems and the application scheduler, motivates further work on this theme. The application scheduler is on the best interest of the LIP research group and, if its development continues, it may be widely used among the researchers. Guidelines for continuing this dissertation work may include:

- Restructuration of the LipMiniAnalysis library, creating a data structure holding all the event information in the input data file. These files are usually 1 GB size, which is perfectly capable of being stored in RAM memory. It would allow a more efficient implementation of this kind of applications on heterogeneous systems with hardware accelerators.
- Refining the GPU implementation using the new restructured version of LipMiniAnalysis and analyze if the potential performance gains justify investment from the LIP group on this hardware and academic formation on programming for these devices.
- Explore parallelism on heterogeneous systems using parallelization frameworks, such as OpenACC.
- Refining the application scheduler prototype by:

  - Implementing and validating the remaining features for working with any kind of scientific application;
  - Validating its capability of scheduling different applications on the same system;
  - Overall load balancing optimizations by refining current mechanisms, such as implementing specific core binding of the applications;
  - Implementing micro-benchmarks to evaluate the computational system and generate the best setup, considering the computational characteristics of these applications.

- Parallelization of the restructured LipMiniAnalysis. Since the library serves as a skeleton for many of the LIP applications, a parallelization at the event level would allow high performance and efficient applications to be developed by physicists without requiring the programming know-how to extract parallelism on homogeneous platforms.

Bibliography

[1] European Organization for Nuclear Research. *CERN European Organization for Nuclear Research*. 2012. URL: http://public.web.cern.ch/public/ (cit. on p. 1).

[2] European Organization for Nuclear Research. *The Proton Synchrotron*. 2013. URL: http://home.web.cern.ch/about/accelerators/proton-synchrotron (cit. on p. 1).

[3] 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. 1).

[4] European Organization for Nuclear Research. *Compact Muon Solenoid experiment*. 2012. URL: http://cms.web.cern.ch/ (cit. on p. 1).

[5] European Organization for Nuclear Research. *ATLAS experiment*. 2012. URL: http://atlas.ch/ (cit. on p. 1).

[6] European Organization for Nuclear Research. *The Large Hadron Collider beauty experiment*. 2012. URL: http://lhcb-public.web.cern.ch/lhcb-public/ (cit. on p. 1).

[7] European Organization for Nuclear Research. *The Monopole Exotics Detector at the LHC*. 2012. URL: http://moedal.web.cern.ch/ (cit. on p. 1).

[8] European Organization for Nuclear Research. *Total Cross Section, Elastic Scattering and Diffraction Dissociation at the LHC*. 2012. URL: http://totem.web.cern.ch/Totem/ (cit. on p. 1).

[9] European Organization for Nuclear Research. *The Large Hadron Collider forward experiment*. 2012. URL: http://home.web.cern.ch/about/experiments/lhcf (cit. on p. 1).

[10] European Organization for Nuclear Research. *A Large Ion Collider Experiment*. 2012. URL: http://aliceinfo.cern.ch/ (cit. on p. 1).

[11] European Organization for Nuclear Research. *Computing*. 2013. URL: http://home.web.cern.ch/about/computing (cit. on p. 2).

[12] European Organization for Nuclear Research. *Animation shows LHC data processing*. 2013. URL: http://home.web.cern.ch/about/updates/2013/04/animation-shows-lhc-data-processing (cit. on p. 2).

[13] European Organization for Nuclear Research. *The Worldwide LHC Computing Grid*. 2013. URL: http://wlcg.web.cern.ch/ (cit. on p. 2).

[14] 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. 2).

[15] Gordon E. Moore. "Cramming more components onto integrated circuits." In: *Electronics, 38(8)* (1965) (cit. on p. 9).

[16] Intel. *The Intel® Xeon PhiTM Coprocessor 5110P*. Tech. rep. 2012 (cit. on pp. 12, 15).

[17] Texas Instruments. *Digital Signal Processors*. 2012. URL: http://www.ti.com/lsds/ti/dsp/overview.page (cit. on pp. 12, 16).

[18] TOP 500. *June 2013*. 2013. URL: http://www.top500.org/lists/2013/06/ (cit. on p. 12).

[19] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architectur: Fermi*. Tech. rep. 2009 (cit. on p. 13).

[20] NVIDIA Corporation. *Tegra*. 2013. URL: http://www.nvidia.com/object/tegra.html (cit. on p. 16).

[21] Sixto Ortiz Jr. "Chipmakers ARM for Battle in Traditional Computing Market." In: *Computer, 44(4):14-17* (2011) (cit. on p. 16).

[22] OpenACC Corporation. *OpenACC: Directives for Accelerators*. 2013. URL: http://openmp.org/wp/ (cit. on p. 17).

[23] James Reinders. *Intel Threading Building Blocks*. Tech. rep. 2007 (cit. on p. 17).

[24] Massachussets Institute of Technology. *The Cilk Project*. 2013. URL: http://supertech.csail.mit.edu/cilk/ (cit. on p. 17).

[25] Edgar Gabriel et al. "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation". In: (2004), pp. 97–104 (cit. on p. 17).

[26] OpenACC Corporation. *OpenACC*. 2012. URL: http://www.openacc-standard.org/ (cit. on p. 18).

[27] 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. 19).

[28] OpenACC Corporation. *How does the OpenACC API relate to OpenMP API*. 2013. URL: http://www.openacc-standard.org/node/49 (cit. on p. 19).

[29] João Barbosa. *GAMA framework: Hardware Aware Scheduling in Heterogeneous Environments*. Tech. rep. 2012 (cit. on p. 19).

[30] Intel. *Profiling Runtime Generated and Interpreted Code with Intel VTune Amplifier*. Tech. rep. 2013 (cit. on p. 19).

[31] S. Browne et al. "PAPI: A Portable Interface to Hardware Performance Counters". In: *Proceedings of Department of Defense HPCMP Users Group Conference* (1999) (cit. on pp. 19, 25).

[32] Free Software Foundation. *GDB: The GNU Project Debugger*. 2013. URL: http://www.gnu.org/software/gdb/ (cit. on p. 20).

[33] NVIDIA. *CUDA-GDB: The NVIDIA CUDA Debugger User Manual*. Tech. rep. 2008 (cit. on p. 20).

[34] F. Rademakers and P. Canal and B. Bellenot and O. Couet and A. Naumann and G. Ganis and L. Moneta and V. Vasilev and A. Gheata and P. Russo and R. Brun. *ROOT*. 2012. URL: http://root.cern.ch/drupal/ (cit. on pp. 21, 60).

[35] L. S. Blackford et al. "An Updated Set of Basic Linear Algebra Subprograms (BLAS)". In: *ACM Trans. Math. Soft., 28-2* (2002) (cit. on p. 21).

[36] Intel Corporation. *Intel Math Kernel Library*. 2013. URL: http://software.intel.com/en-us/intel-mkl/ (cit. on p. 21).

[37] F. Rademakers and P. Canal and B. Bellenot and O. Couet and A. Naumann and G. Ganis and L. Moneta and V. Vasilev and A. Gheata and P. Russo and R. Brun. *PROOF*. 2012. URL: http://root.cern.ch/drupal/content/proof (cit. on p. 21).

[38] Valgrind Developers. *Valgrind*. 2013. URL: http://valgrind.org/ (cit. on p. 22).

[39] Andrew Waterman SamueL Williams and David Patterson. "Roofline: An insightful Visual Performance model for multicore Architectures". In: *Communications of the ACM, 65-76* (2009) (cit. on pp. 26, 61).

[40] Makoto Matsumoto and Mutsuo Saito. "Variants of Mersenne Twister Suitable for Graphic Processors". In: *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation* (1998) (cit. on p. 27).

[41] NVIDIA. *CUDA CURAND Library*. Tech. rep. 2010 (cit. on pp. 27, 42).

[42] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". In: (2012) (cit. on p. 27).

[43] W. Hoermann and G. Deringer. "The ACR Method for generating normal random variables". In: *OR Spektrum 12, 181-185* (1990) (cit. on p. 28).

[44] G. E. P. Box and Mervin E. Muller. "A Note on the Generation of Random Normal Deviates". In: *Annals of Mathematical Statistics, Volume 29, 610-611* (1958) (cit. on p. 28).

[45] George Marsaglia and Wai Wan Tsang. "The Ziggurat Method for Generating Random Variables". In: *Journal of Statistical Software* (2000) (cit. on p. 28).

[46] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *AFIPS spring joint computer conference, (30) 483-485* (1967) (cit. on pp. 34, 61).

[47] NVIDIA Corporation. *NVIDIA CUDA Occupancy Calculator*. 2013. URL: developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls (cit. on p. 42).

[48] Intel. *Intel Xeon Processor E5-1600/E5-2600/E5-4600 Product Families Datasheet*. Tech. rep. 2012 (cit. on p. 59).

[49] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. University of Virginia. 2013. URL: http://www.cs.virginia.edu/stream/ (cit. on p. 59).

Test Environment

This appendix focuses on characterizing the hardware and software used in all performance measurements of the application for the different implementations developed.

For the shared memory implementation testing was used four dual-socket multicore systems with CPUs of different architectures on the SeARCH cluster, with the purpose of testing a wide range of systems commonly available in small physics research group clusters. The first, named compute-711 node, has two Intel Xeon E5-2670 (Sandy Bridge architecture) [48], using the Quick Path Interconnect (QPI) interface between CPUs, in a Non Unified Memory Access model (NUMA), meaning that the latency of a CPU acessing its own memory bank is lower than accessing the other CPU memory bank. The QPI interface can perform up to 8 GT/s (giga transfers per second) of 2 bytes packets, in each of the two unidirectional links, with a total bandwidth of 32 GB/s. The system features 64 GB of DDR3 RAM with a speed of 1333 MHz, for a maximum bandwidth of 52.7 GB/s. All RAM bandwidths were measured using the STREAM benchmark [49].

The second system, compute-601 node, has the same amount of RAM at the same speed, with a maximum bandwidth of 28.6 GB/s. The two CPUs are Intel Xeon X5650 (Nehalem architecture). The difference of memory bandwidth is due to the different memory controllers, while the one in Nehalem has 3 memory channels the one in Sandy Bridge has 4. The two CPUs are interconnect by a QPI interface, but with a different speed than the Sandy Bridge, performing 6 GT/s in each of the two unidirectional channels, for a total bandwidth of 24 GB/s.

The third is also an Intel system, compute-401 node, with 8 GB RAM and a maximum bandwidth of 18.4 GB/s. The CPUs are the Xeon E5520 (Nehalem architecture). The two CPUs are interconnected by a QPI interface with the same bandwidth of 24 GB/s as the previous system.

The fourth system features two AMDOpteron 6174 CPUs, being the system with the most physical cores. It has 64 GB of DDR3 RAM at 1333 MHz, with a maximum measured bandwidth of 39.8 GB/s. AMDuses HyperTransport (HT) 3.0 technology, a point-to-point interconnection similar to QPI capable of transmitting 4 byte packets through two links, for an aggregate bandwidth of 51.2 GB/s. The characteristics of the CPUs on the three systems are presented in table 1.

The NVidia Tesla C2070 has 14 Streaming Multiprocessors (SM) with 32 CUDA cores each, making a total of 448 CUDA cores working at a frequency of 1.15 GHz. Each SM has 64 KB of L1 cache, with two configurations for individual cache and shared memory for the CUDA threads inside a block (16 KB / 48 KB and vice-versa). The 768 KB of the L2 cache are shared among all SMs. The GPU features 6 GB of GDDR5 memory. NVidia claims a theoretical peak performance of 1.17 TFLOPS (double precision).

The Intel Xeon Phi features 60 multithread cores, with 4 threads per core, working at 1 GHz. It has an aggregated L2 cache size of 30 MB, spread among the cores. The private L1 cache has 64 KB size for data and another 64 KB for instructions. The cores communicate using a bidireccional ring network. The chip has 8 GB GDDR5 memory, capable of a bandwidth of 320 GB/s.

CPU	Intel Xeon E5-2670	Intel Xeon X5650	Intel Xeon E5520	AMDOpteron 6174
Architecture	Sandy Bridge	Nehalem	Nehalem	MagnyCours
Clock Freq.	2.60 GHz	2.66 GHz	2.3 GHz	2.2 GHz
# of Cores	8	6	4	12
# of Threads	16	12	8	12
L1 Cache	32 KB I. + 32 KB D. per Core	32 KB I. + 32 KB D. per Core	32 KB I. + 32 KB D. per Core	64 KB I. + 64 KB D. per Core
L2 Cache	256 KB per Core	256 KB per Core	256 KB per Core	512 KB per Core
L3 Cache	20 MB shared	12 MB shared	8 MB shared	
CPU Interconnection	QPI @4.0 GHz	QPI @3.2 GHz	QPI @3.2 GHz	HT @3.2 GHz
ISE	AVX	SSE 4.2	SSE 4.2	SSE 4a

Table 1: Characterization of the CPUs featured in the three test systems.

The compiler used was the GNU compiler version 4.8, using the -O3 optimizations and the AVX/SSE 4.2/SSE 4a instruction set (depending on the CPU architecture) where the compiler sees fit. The compiler used for the Intel Xeon Phi implementation was the Intel Compiler (ICC) version 13.1. Both compilers feature the OpenMP version 3.2 used in the shared memory implementation. For the GPU implementation was used the CUDA 5 SDK, in conjunction with the GNU compiler version 4.6.3 for the code to run on the CPU (any later versions are not supported by the NVidia NVCC compiler). The ROOT [34] version used was the 5.34/05. Was used the Performance API version 5.0 for measuring the hardware counters of the different CPUs for the characterization of `ttDilepKinFit`.

Theoretical Performance Models

Appendix .1 Amdahl's Law

The speedup that can be achieve by parallelizing an application is not only dependent on the number of parallel tasks but also on the percentage of the code that will run in parallel. This means that it is possible to have an extremely optimized implementation of the parallelization but if only a small part of the code is parallel the speedup will be small.

Amdahl's Law [46] defines the maximum attainable speedup of parallelizing an application, comparing a multithreaded application using N processors with its serial counterpart. The law takes into account the portion of the code, P, that can be paralelized and defines the maximum speedup S that can be obtained.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

Equation 1 defines the maximum attainable speedup resultant from the parallelization of an application according to the Amdahl's Law. The law is used in this work to prove that the small speedups for fewer number of variations per event are close to the theoretical maximum and are limited by the percentage of the code that can be made parallel.

Appendix .2 Roofline Model

The Roofline model [39] was used characterize the system in terms of attainable peak performance. This model uses two metrics for the performance calculation: the peak CPU performance and the memory bandwidth. With the peak values of these two metrics a roofline is drawn, being the theoretical limit for the performance on the system. Then, other ceilings can be added, which further limit the attainable performance. The classic Roofline uses float point computation as the peak CPU performance metric, which is usually advertised as peak performance of the hardware by CPU manufacturers. It may be a good metric for heavy computational algorithms, such as matrix arithmetic, but the type operations on the critical region of `ttH_dilep` (`ttDilepKinFit` function) are much more varied, as shown by the instruction mix presented in section 3.2.1. Instead, the computational intensity was used for measuring the CPU peak performance, as it considers all types of instructions.

The peak computational intensity is calculated with the formula 2. The clock frequency and number of cores are easily obtained by consulting the CPU specifications, while the number of instructions issued per clock cycle is more difficult to obtain. It is based on the super scalarity degree of the processor, i.e., the number of instructions that can be decoded per clock cycle, and then it must be confirmed if

it matches with the number of arithmetic/memory units. The value obtained makes the horizontal roof of the model. The tilted roof of the Roofline model refers to the maximum memory bandwidth of the system and it was determined using the stream benchmark. The values are presented in appendix .

$$C = ClockFreq. * ofCores * ofInstructionsperClock \tag{2}$$

Further ceilings can be added, defining factors, at the hardware level, which affect the performance if not taken into consideration. The most common ceilings are presented in the Roofline model for the test system:

One processor: the usage of one CPU in a dual-socket system halves the maximum performance as only half of the available computational resources are used.

TLP: thread level parallelism (TLP) allows exploiting the usage of all available cores in one CPU. If only one core is used, the maximum performance is limited by a factor equal to the number of cores.

Mul/Add balance: each CPU core has two arithmetic units, one for additions and other for multiplications and divisions. If an algorithm has only additions or multiplications, one of the arithmetic units will be idle, halving the performance. To use both arithmetic units at the same time, the algorithm must have a similar number of additions and multiplications and they must be intercaleted in such a way that the instruction dispatcher is able to keep both units always occupied.

ILP: if the instruction level parallelism is not used the performance will be highly affected. Considering an addition, which takes 2 clock cycles to compute, if only one instruction is issued every 2 cycles the performance is half of the performance if one instruction is issued every cycle. Considering other arithmetic units, such as divisions that use many more clock cycles, the performance could be even more affected.

No Triple Channel: the CPU has 3 memory channels, allowing to receive data from 3 different streams simultaneously. However, this requires the system to have at least 3 RAM modules. If only 1 RAM module is used, the CPU will work in single channel mode, using a third of the maximum memory bandwidth.

Figure 1 illustrates the Roofline model for the compute-601 system (the only with the Performance API required for memasuring the application computational intentsity) used in the `ttDilepKinFit` computational characterization.

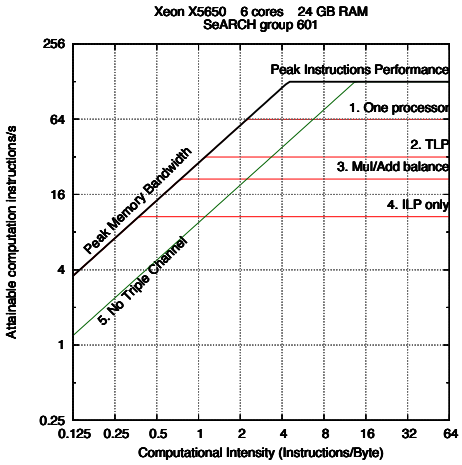


Figure 1: Roofline models for the compute-601 system used in the `ttDilepKinFit` computational characterization.

Test Methodology

The purpose of this appendix is to present and justify the methodology use to conduct the performance and algortihm characterization related tests.

All tests used the same application input data, a file containing 5738 events, from which 1867 reach the `ttDilepKinFit` and the rest are discarded in the previous cuts, of a electron-muon collision. The problem size is considered to be the number of variations to do to each combination of the jets and leptons within an event. The number of variations tested were 2^x, where $x \in \{1, ..., 9\}$.

Various number of threads was used to conduct the tests of the shared memory implementations, depending on the system to run. The test using 1 thread has the purpose of evaluating the overhead of the creation and access to the data structures, as well as other implementation details relative to the parallelization. There is always one test using more threads than available by the hardware and is used to test if the software multithreading (managed by the operating system) has benefits, which can expose problems of memory access, specially on NUMA systems. The number of threads equal to the number of cores in one CPU, making one thread per core, tests the application without the limitations of the NUMA memory accesses and the multithreading. With the number of threads equal to the total number of cores tests the use of both CPUs, but still not using hardware multithreading. The number of threads equal to the available hardware threads tests both CPUs with hardware multithreading active.

In the GPU, the number of threads used was equal to the number of variations times the number of combinations, so that each thread computes a variation of a combination. This provides a high number of threads to hide the latency of memory accesses of the GPU.

It is important to adopt a good heuristic for chosing the best measurement since it is not possible to control the operating system and other background tasks, which can occasionally interfere with the measurements. The mean value is very sensitive to extreme measurements, i.e., the cases when the system has a spike on the workload from other OS tasks and it impacts the measurement, not accurately reflecting the actual performance of the application. The median can be affected by a series of values measured while the system was under some load. Chosing only the best measurement, is not a solid heuristic, since it is more hard to replicate the result.

The heuristic chosen was the *k best* methodology. It accepts the best value within a given range of other k measurements. It is almost as good as the best value heuristic for obtaining the best measurement but offers a solid result capable of being replicated. A 5% interval was used with a k of 4, for a minimum of 16 measurements and a maximum of 32 (in case that there are less than k values within the interval).

The `gettimeofday` function from the C standard libraries was used to measure the execution time of the application and its functions, providing microsecond precision that is enough considering that the original application always takes more than 3 seconds to execute on the test systems used. CUDA Events were used for measuring the portion of the code executed on the GPU, ensuring that the times were properly recorded and by synchronizing the kernel execution.