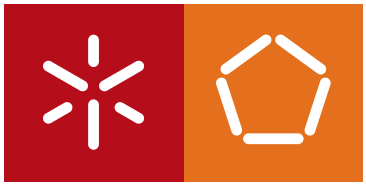
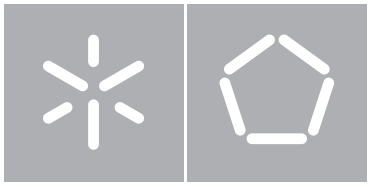


Universidade do Minho
Escola de Engenharia

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

Abstract

Contents

1. Introduction **1**
- 1.1. Context . . . 1
- 1.2. LIP Research Group . . . 2
- 1.3. Motivation, Goals & Scientific Contribution . . . 2
  - 1.3.1. The Top Quark system and Higgs boson decay . . . 3
  - 1.3.2. Goals . . . 5
  - 1.3.3. Scientific Contribution . . . 5
- 1.4. Dissertation Structure . . . 6

2. Technological Background **7**
- 2.1. Hardware . . . 7
  - 2.1.1. Homogeneous systems . . . 7
  - 2.1.2. Heterogeneous systems . . . 9
- 2.2. Software . . . 13
  - 2.2.1. pThreads . . . 14
  - 2.2.2. OpenMP, TBB and Cilk . . . 14
  - 2.2.3. Message Passing Interface . . . 15
  - 2.2.4. CUDA . . . 15
  - 2.2.5. Parallelization frameworks for heterogeneous systems . . . 16
  - 2.2.6. Profiling and debugging . . . 17

3. `ttH_dilep` Application **18**
- 3.1. Application Flow . . . 18
- 3.2. Critical region computational characterization & optimization . . . 21
  - 3.2.1. Computational characterization . . . 21
  - 3.2.2. Initial optimizations . . . 24

4. Parallelization Approaches **26**
- 4.1. Shared Memory Parallelization . . . 27
  - 4.1.1. Implementation . . . 28
  - 4.1.2. Performance Analysis . . . 30
- 4.2. GPU Parallelization . . . 34
- 4.3. MIC Parallelization . . . 37
- 4.4. Scheduler Parallelization . . . 37

Appendix A. Test Environment **38**

Appendix B. Theoretical Performance Models **40**
- Appendix B.1. Amdahl's Law . . . 40
- Appendix B.2. Roofline Model . . . 40

Appendix C. Test Methodology **42**

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.

List of Figures

1.1. Schematic representation of the $t\bar{t}$ system. . . . 4
1.2. Schematic representation of the $t\bar{t}$ system with the Higgs Boson decay. . . . 5

2.1. Schematic representation of a homogeneous system. . . . 7
2.2. Schematic representation of a modern multicore CPU chip. . . . 9
2.3. Schematic representation of a heterogeneous system. . . . 10
2.4. Schematic representation of the NVidia Fermi architecture. . . . 11
2.5. Schematic representation of the Intel MIC architecture. . . . 12
2.6. Schematic representation of CUDA thread hierarchy. . . . 16

3.1. Schematic representation for the `ttH_dilep` application flow. . . . 19
3.2. Callgraph for the `ttH_dilep` application on the compute-711 node . . . 20
3.3. Callgraph for the `ttH_dilep` application on the compute-711 node for 256 variations per combination. . . . 20
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. . . . 21
3.5. Arithmetic intensity for various domains of computing problems. . . . 22
3.6. Instruction mix for the `ttDilepKinFit` with no and 512 variations, left and right images respectively. . . . 22
3.7. Miss rate on L1, L2 and L3 cache of `ttDilepKinFit` for various number of variations. . 23
3.8. Roofline of the compute-601 system with the computational intensity of `ttDilepKinFit` for 1 and 512 variations. . . . 23
3.9. Speedup of the `ttH_dilep` application with the `TRandom` optimization. . . . 24

4.1. Schematic representation of the event-level parallelization model. . . . 26
4.2. Schematic representation of the `ttDilepKinFit` sequential (left) and parallel (right) workflows. . . . 27
4.3. Schematic representation of the parallel tasks accessing the shared data structure (left) and the new parallel reduction (right). . . . 28
4.4. Schematic representation of the event-level parallelization model. . . . 30
4.5. Theoretical speedup (Amdahl's Law) for various number of cores. . . . 31
4.6. Speedup for the parallel non-pointer version of `ttH_dilep` application with static (left) and dynamic (right) scheduling. . . . 31
4.7. Speedup for the parallel pointer version of `ttH_dilep` application with static (left) and dynamic (right) scheduling. . . . 32
4.8. Theoretical speedup (Amdahl's Law) for various number of cores. . . . 32
4.9. Theoretical speedup (Amdahl's Law) for various number of cores. . . . 33
4.10. Theoretical speedup (Amdahl's Law) for various number of cores. . . . 34
4.11. Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-401 node. . . . 34
4.12. Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-511 node. . . . 35
4.13. Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-601. . . . 35
4.14. Execution times of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations. . . . 35
4.15. Schematic representation of the `ttDilepKinFit` workflow. . . . 36

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 member 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 and 113 different nationalities.

CERN research focus on the basic constituents of matter, which started by studying the atomic nucleus but quickly moved into high energy physiscs (HEP), focusing on the interaction between particles. The instrumentation used in the nuclear research, physics-wise, 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, and collide them in the detectors (this collision is called an event). The detectors record various characteristics, such as energy and momentum, of particles resultant from complex decay processes of the original particles. These experiments are performed to test and validate specific HEP theories by comparing the results of the collision to the expected theoretical model.

It started with a small low energy particle accelerator, the Proton Synchrotron [2] inaugurated in 1959, but the facilities were iteratively being upgraded and expanded. The current facilities are constituted by the older accelerators (some decomissioned while others are still functional) and detectors, as well as the newer Large Hadron Collider (LHC) [3] high energy particle accelerator which is located 100 meter underground and has 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 has their own detector on the LHC and conduct similar or different experiments, but with the use of distinct technologies and research approaches. Currently one of the most popular researches being conducted 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 particle beams that it accelerates).

Fonte upgra

Approximately 600 millions of collisions occur every second in each of the experiment's detectors at the LHC, where the detectors react to the particle interaction and produce electric signals, generating massive amounts of raw data. It's estimated that all the detectors combined produce 25 petabytes of data per year [11, 12]. CERN does not have the financial resources to have the computational power 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 into meaningful information, usable by the remaining tiers. The data is received on a format designed for this reconstruction, with information about detector and software diagnostics. After the reconstruction the data has a different formats, the Event Summary Data (ESD) and Analysis Object Data (AOD), each one with different purposes, containing information of the reconstructed objects and calibration parameters, and 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 24/7. In this tier, the reconstructed data suffers more reprocessing, in order to refine it by filtering only relevante information and reducing the size of the data, now in Derived Physics Data (DPD) format, that is 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 outputs 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 consists of the LHC Optical Private Network.

There are around 140 Tier-2 computational centers 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 related groups terminology, an analysis is a denomination for an application which is designed to process a given amount of data in order to test a specific HEP theory by providing physically relevant information about events that may support the said 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 Portugal has 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 frameworks, such as the LipCbrAnalysis and LipMiniAnalysis.

This dissertation work results from a close cooperation between the Department of Informatics of the University of Minho and the LIP laboratory in Minho.

1.3. Motivation, Goals & Scientific Contribution

With an increase of events and, consequently, the data being produced by the detectors at the LHC, specifically in the ATLAS experiment, the research groups will need a bigger budget for aquiring and maintaining computational resources due to an increase of analysis to perform. To add up to this data

increase, research groups working on the same experiment have a positive rivalry to be the first find and publish relevant results. The finding of these results is directly related to the amount of events processed, meaning that groups with more computational resources are one step ahead.

Better results are not only obtained by increasing the amount of events analyzed; it is important to take into account the quality of each analysis. The ATLAS detector has an experimental resolution of 2%, meaning that each measured value for a characteristic of a resultant particle of a collision might not be real 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 is related to the Top Quark physics and the Higgs Boson. An application was devised that reconstructs an event following the theoretical model of Top Quark decay and then also attempts to reconstruct the associated Higgs Boson. Each event can be reconstructed several times, with some of its parameters slightly varied by a random offset (with a maximum magnitude of 2% of the original value), and by chosing the reconstruction that satisfies the most the theoretical model a better solution is obtained, overcoming the experimental resolution of the ATLAS detector. The more reconstructions per event are performed the longer will take to process an event. 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 analysis applications. Also, hardware is not necessarily getting faster, but wider by increasing the number of cores per chip (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 efficient for specific problem domains (heterogeneous systems). It is important to have a knowledge of these newer architectures in order to develop efficient applications that resort to parallelism in order to better use all the resources available in a system. Programming for such architectures (multicore CPUs and hardware accelerators) requires a set of skills and experience that most physicists (usually self-taught programmers) do not have, causing poorly optimized applications to be developed.

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, from which only some of the final particles react with the detector for recording 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 (the $t\bar{t}$ system), resulting from a head-on collision of two protons, is presented in figure 1.1.

The ATLAS detector is able to record the characteristics of Bottom Quarks, which are detected as a jet 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, necessary for researching their properties, it is necessary to have the information of all the final particles, so the neutrino characteristics must be determined. This

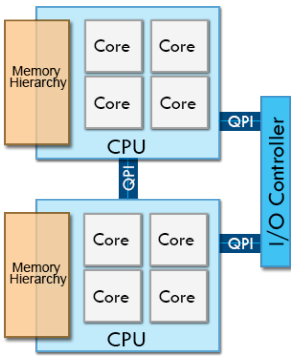


Figure 1.1.: Schematic representation of the $t\bar{t}$ system.

is possible to do as the $t\bar{t}$ system obeys a set of properties, and using the information of the quarks and leptons the neutrinos characteristics are 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 affects the quality of the research being 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, it is needed 2 jets and 2 leptons to reconstruct the $t\bar{t}$ system, but the data for an event may have many of these particles associated. To obtain the best reconstruction for the $t\bar{t}$ system of a given event it is necessary to reconstruct the respective neutrinos and then the whole system for every combination of 2 jets and 2 leptons, and only chose the most accurate reconstruction.

Another factor affecting the quality of the reconstruction is the experimental resolution of the ATLAS particle detector, which associates an error of 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, which might render an event with relevant physics useless. It is possible to overcome this problem by performing the kinematical reconstruction, and then the whole $t\bar{t}$ system reconstruction, a large amount of times for each combination of 2 Bottom Quark jets with 2 leptons, with a random variation to the particle characteristics (momentum, energy and mass) of a maximum magnitude of 2% of the original value. The amount of variations performed per combination will directly impact the final quality of the event reconstruction, as more of the search space (defined by the experimental resolution error) is covered compared to performing a single reconstruction. The more variations are performed the more likely it is to find the best possible reconstruction of the $t\bar{t}$ system.

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.

This specific analysis of events presented is performed by an application developed by LIP researchers, the `ttH_dilep`. The application receives input data file with a set of events and reconstructs the $t\bar{t}$ system and the Higgs Boson for each event using the processes described. These files are usually 1

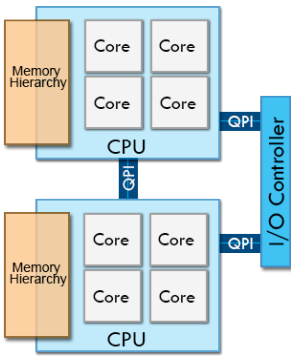


Figure 1.2.: Schematic representation of the $t\bar{t}$ system with the Higgs Boson decay.

GB long and the LIP research requires that hundreds of them are processed by the same application, considering a specific experiment such as the presented in this subsection. A in-depth computational analysis of `ttH_dilep` is presented in chapter 3, where its flow is presented, it is characterized in terms of various metrics (such as computational intensity) and the critical regions are identified.

1.3.2. Goals

By increasing the performance of the Top Quark and Higgs Boson reconstructions it is possible to perform more variations per event, increasing the quality of the results, and increase the throughput of events processed. The objective of this dissertation work is to take a sequential application made by physicists, which the main concern during its development was the correcteness of the code rather than its performance, the `ttH_dilep`, 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 efficiency (performance and usability) of hardware accelerators for this type of problem, and (iii) the development of a simple scheduler for managing the workload 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.

This work will give a inside perspective of how scientific applications are being developed by programmers with little to no background in computer science, and possibly define a set guidelines for coding of efficient applications and the usage of parallelism in such applications. All the changes that will be made to the `ttH_dilep` application, including the introduction of parallelism, will be as independent as possible from the context of this specific problem, in such a way that they might be portable to other applications without requiring major modifications. The work will be structured, implementation wise, so that the parallelization mechanisms and the scheduler are possible to be improved and 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 conducted by LIP, provide a set of tools and know-how to improve the performance of similar scientific applications and expose the problematic of unefficient usage of computational resources. By improving the quality of the research, LIP will gain an advantage over other research groups in the look for new Top Quark physics and in the Higgs Boson discovery. By experiencing the process of optimizating scientific applications of this kind it is possible to provide physicists with some know-how and tools for optimization and parallelization with the goal of increasing the performance in future applications. By developing applications that efficiently

use all the computational resources available it is possible to reduce the investment in new hardware, which otherwise would have small pratical returns.

1.4. Dissertation Structure

This dissertation has X 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..

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 started with 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, specifically with the use of an hierarchy model, and current systems are usually made from multicore CPUs and various types of memory.

2.1.1. Homogeneous systems

The most common are homogeneous systems, constituted from one or more CPU chips with their own memory bank (RAM memory) and interconnected by a specific interface, which is manufacturer-specific. Although the system uses a shared memory model, where all the data is always available for each CPU, in the case of a multiple CPU system, since the memory is distributed in one bank per CPU 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.

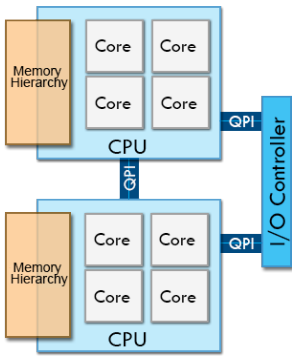


Figure 2.1.: Schematic representation of a homogeneous system.

Figure 2.1 shcematizes the structure of an homogeneous system, in a shared memory environment with an 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. This enabled the increase in CPU chips clock frequency 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 reduce the 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, explained below) instructions. A single CPU core implements various mechanisms for improving the performance of applications, at the instruction 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. The three mechanisms presented next allow for ILP to be used.

- ***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. This stages range from loading the data, instruction execution in, also pipelined, arithmetic units and writing the results back to memory. This allow, as an example, 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 based on previous conditional jumps, and start executing instructions in the predicted 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 usage. 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 done at compile time.

Multithreading is the execution of multiple 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 core hardware. If one of the threads is waiting for data to execute the next instruction, other thread can resume execution while the first is stalled. It also can allow 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 is presented in figure 2.2. It is constituted of several, possibly multithreaded, cores, each with its own 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.

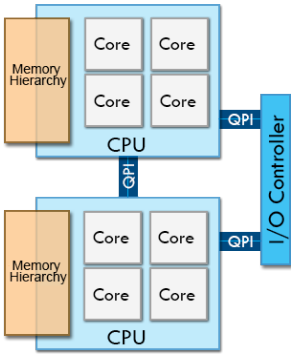


Figure 2.2.: Schematic representation of a modern multicore CPU chip.

2.1.2. Heterogeneous systems

With the emerging use of hardware designed for specific computing domains, hardware accelerators, which purpose is to efficiently solve a small range of problems, as opposed to general purpose CPU chips. This marked the begining of heterogeneous systems, where one or more CPU chips, operating in a shared memory environment as in 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 and vice-versa.

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 spot in applications performance.

Hardware accelerators are usually made from small 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, offloading the CPU from such 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

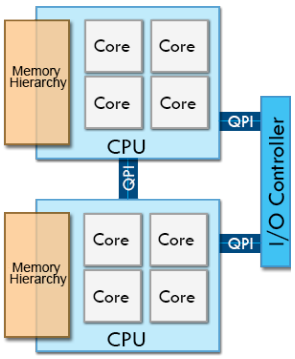


Figure 2.3.: Schematic representation of a heterogeneous system.

(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 when compared to previous years. The Intel Xeon Phi is becoming increasingly popular, being the accelerator device of choice in 11 clusters of the TOP500. The most used accelerator are NVidia GPUs.

Graphics Processing Unit

One of the first accelerators to arrive on the market is the General Purpose Graphics Processing Unit (GPGPU). Their purpose is to accelerate image processing, which started of as simple pixel drawing and evolved to complex capabilities of 3D scene rendering, such as transforms, lighting, rasterization, texturing, depth testing, and display. They later allowed for some flexibility due to the industries demand for costumizable shaders, which also enable the possibility of using this hardware as a hardware accelerator for other purposes than image processing.

The GPU architecture is based on the SIMD model. Its original purpose is to process, or synthethise, an image, which is a large set of pixels. The processing of each pixel does not usually depend on the processing of its neighbours, or any other pixel on the image, making it data indenpent and allowing the processing of every pixel to be performed simultaneously. This massive parallelism is one of the most important factors that affected 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 massive parallel problems, such as some matrix arithmetic, such as additions and multiplications. However, GPUs had some important features that were only oriented for image processing and affected its use in other situations. One example is that it 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, with a wide range of available hardware. This category of devices, known as the Tesla, have more GDDR RAM, processing units and a slight different structural design suitable for use in cluster computational nodes (in terms of size and cooling). 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 hierarchy (global external memory, GDDR5 RAM, and an in-chip

2-level cache and shared memory block). Each SM contains a set of CUDA cores, NVidia designation for their 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 explained next) to match 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 a given time. A warp can only be constituted by CUDA threads from the same block.

GPU global memory accesses have a high latency associated, which can cause the CUDA threads to be stalled waiting for data. The strategy behind the GPU architectures is to provide the device with a high number of threads, allowing the schedulers to keep a scoreboard of which warps are ready to execute and which are waiting for data to load. With a high number of threads, the scheduler 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 between the CPU and GPU must be minimal as it greatly restricts the performance.

The relevant architectural details of this architecture, specifically for the Tesla C2070, are explained in this section. The Fermi architecture is schematized in figure 2.4.

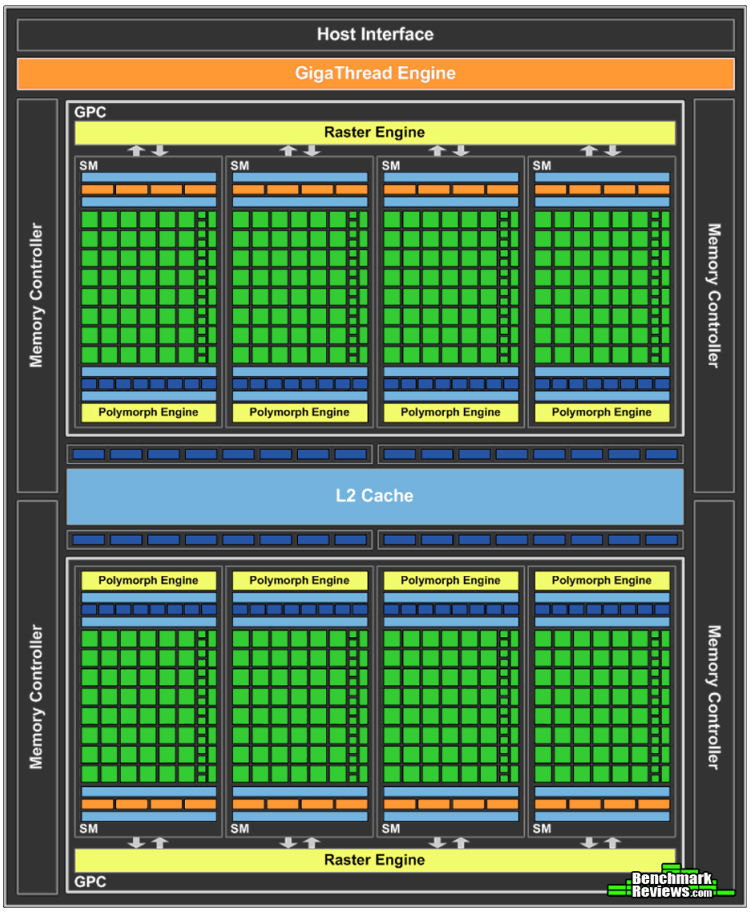


Figure 2.4.: Schematic representation of the NVidia Fermi architecture.

In the Tesla C2070, each SM has 32 CUDA cores and 14 SM per chip, making a total of 448 CUDA cores. In each SM there are 4 Special Functional Units (SFU) to process special operations such as square

roots and trignometric arithmetic.

These devices have a slightly different memory hierarchy than the CPUs, but still with the faster and smaller memory closer to the processing units (CUDA cores). Each CUDA thread can have up to 63 registers, but it decreases with the use of more threads, 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 use of coalesced memory accesses. Since the load units get 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 .

Intel Many Core Architecture

The Intel Many Integrated Core (MIC architecture), currently known as Intel Xeon Phi, has a different conceptual design than the Nvidia GPUs. A chip can have up to 61 cores, multithreaded 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].

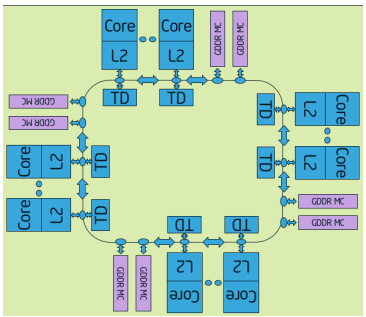


Figure 2.5.: Schematic representation of the Intel MIC architecture.

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 to memory bound algorithms, as opposed to GPUs (Fermi only has a bandwidth of 192.4 GB/s). Later Intel claims that it will launch a version more oriented 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 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, in such a way 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 , where 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, prior to execution. No further interaction with the CPU is required.

Offload , where 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) and consume only 50 Watts [17].

ARM processors are now leading the mobile industry and, alongside the new NVidia Tegra processors [20] which are steadly increasing their market share, are likely to be adopted by the HPC community[1] due to the 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 produce 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 the processing power, causing a lack in the academic formation of programmers in terms of optimization and parallel programming, as it is not a fundamental skill. Self taught programmers have an increased obstacle due to the lack of theoretical basis when trying these new parallel programming paradigms.

Programming for multicore environments require 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.

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

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 cannot be easily ported to these devices expecting high performance. To optimize the code for these devices it is necessary to deeply understand 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, if the purpose is to change the data, the programmer must ensure that different threads are not simultaneously changing the same piece of data by serializing the operations. If the order of the operations is important, further control is required. 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 the second thread operations.

The balancing of the workload between the cores of a single CPU chip, and even between CPU and hardware accelerators, is an important aspect to extract performance and get the most usage possible from 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, causing 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. The same concepts apply to load balancing between CPU and hardware accelerators, with the increased complexity of transferring data between them with a high latency cost.

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 close 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 next.

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 mostly 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, with 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 explained next) but on different data. This set of instructions is considered a CUDA kernel, in which the programmer defines the behavior of the CUDA threads. A simple way to visualize this concept is 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 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 example, each CUDA thread corresponds to an element of the matrix, computing its value, and is organized in a block of many CUDA threads, which 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 block and the grid have a limited size.

When programming these devices, conditional jumps must be avoided if 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.

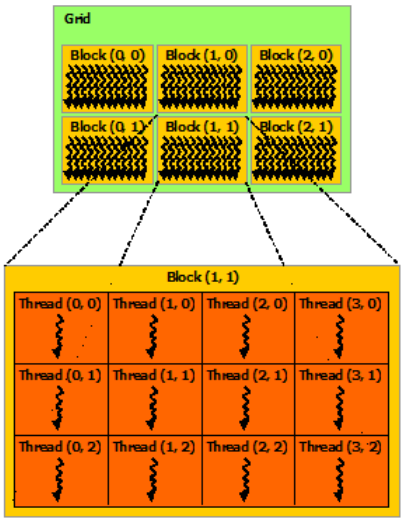


Figure 2.6.: Schematic representation of CUDA thread hierarchy.

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 the architecture from the programmer. The programmer to still needs to have some knowledge of each different architecture, and it is necessary to instruct the framework about how tasks should be divided, in order to fit the requirements of the different devices.

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

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

uses a relaxed memory consistency model, where the programmer can use a synchronization primitve to enforce memory consistency.

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.

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 solve the problem presented in section 1.3, which runs in the Tier-3 CERN 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 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 for 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 tiers centers, classes for representing physic elements, mathematical routines, pseudorandom number generators, histograming, curve fitting minimization and data visualization methods. It is originally designed and developed mostly by physicists self taught on programming. This results in a framework that has room for improvement, such as code restructuration in some modules related to the data analysis. Other possible optimizations could pass through replacing some mathematical functionalities by dependencies on faster libraries, such as BLAS [35] or MKL [36]. ROOT has an extension for data distribution on distributed memory systems, the Parallel ROOT Facility (PROOF) [37].

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 provides a set of functions that are common ground for most applications developed by LIP. This library is currently not suited for parallelization in shared or distributed memory, as later explained in chapter 4.

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

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 elements, which are repeated for every event in the input data file:

Load Event Data: information relative to the event, all 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 an event is named `Loop`.

Process Event: most of the event processing is performed in the `DoCuts` function. In this function an event is tested in a set of filters (referred as cuts), with the possibility of being rejected in any of

them and other event is loaded. This cuts test many characteristics of the events, 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 computed in this filter by the complex `ttDilepKinFit`. From a computational point of view, all the other cuts are simple.

`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 there is an inner loop that iterates through the number of variations per combination defined at compile time. The next step is to apply the variation to the particles characteristics and then the it attempts to reconstruct $t\bar{t}$ system (kinematical reconstruction). If a reconstruction is possible, then the Higgs Boson is reconstructed and the probability of the reconstruction computed. If not, the reconstruction of that variation is discarded, as it is needed to know which jets were used in this reconstruction to avoid using them in the Higgs Boson reconstruction. 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.

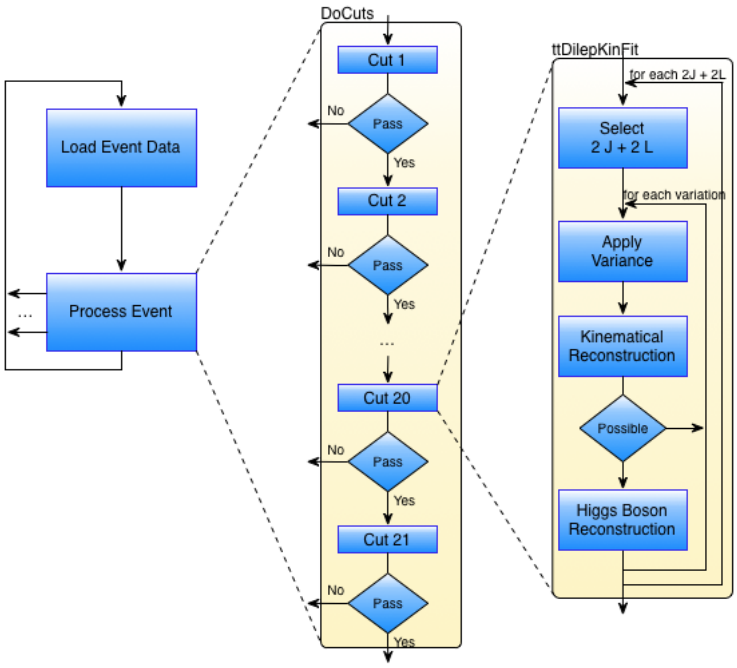


Figure 3.1.: Schematic representation for the `ttH_dilep` application flow.

This schematic representation of the application flow was designed based on the source code analysis and the callgraphs obtained by using the callgrind tool from Valgrind [**Valgrind**]. 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 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 performing only one variation per combination 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 only 6.2% of the time is on the kinematical reconstruction (`dilep`). It is clear that the most complex cut is the portion of the application that must be optimized as it uses most of the execution time. This becomes even more evident when considering 256 variations

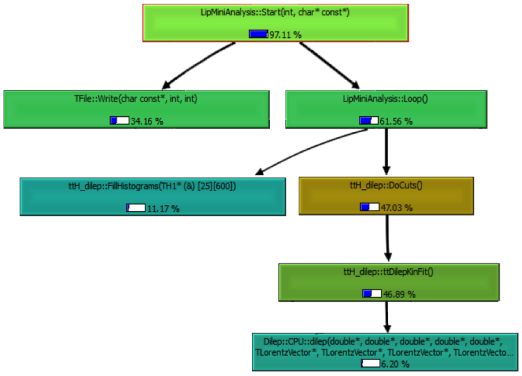


Figure 3.2.: Callgraph for the `ttH_dilep` application on the compute-711 node[1].

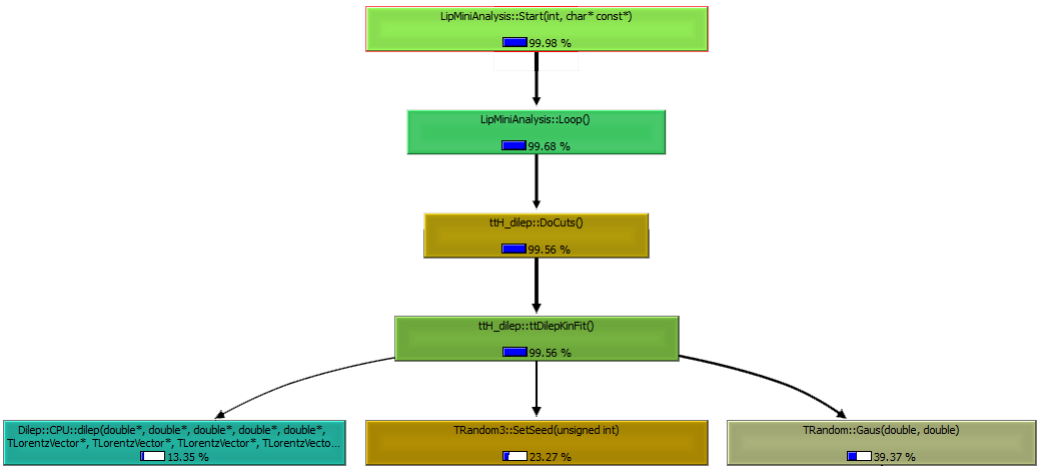


Figure 3.3.: Callgraph for the `ttH_dilep` application on the compute-711 node for 256 variations per combination.

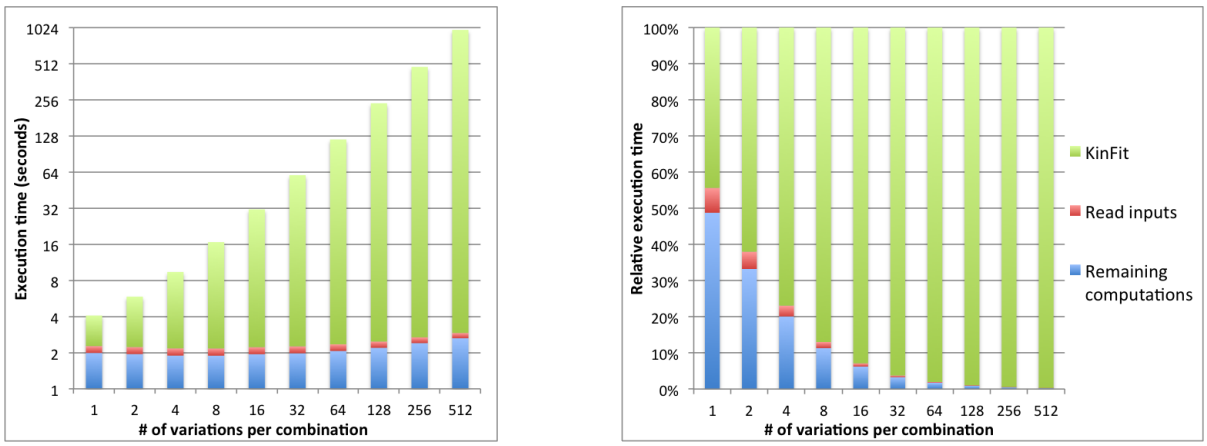


Figure 3.4.: Absolute (left) and relative (right) execution times for the `ttH_dilep` application considering the `ttDilepKinFit` (KinFit) function, I/O and the rest of the computations.

per event, where `ttDilepKinFit` uses 99.6% of the application execution time. Now, it is possible to see that the pseudo-random number generator (`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 will 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, the I/O of the application, which was also identified by the callgraph 3.2 as a time consuming task for a low number of variations, and the rest of the computations.

While the I/O and all event processing, with the exception of KinFit, execution times remain constant, with only a slight increase for 256 and 512 variations as it causes more events to be reconstructed, which otherwise would not be possible, and they need to be processed at the final cut 21 (see figure 3.1). As expected, the execution time of KinFit increases linearly with the number of variations to perform per

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

combination. This indicates that the arithmetic intensity of KinFit has the complexity of *O(N)*, where *N* is the number of variations to perform per combination. 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 same input data file. This is the initial problem size. The only way of increasing the problem size, while maintaining the same input data file, is to perform more variations and the number of reconstructions grow linearly with the with this increase. 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.

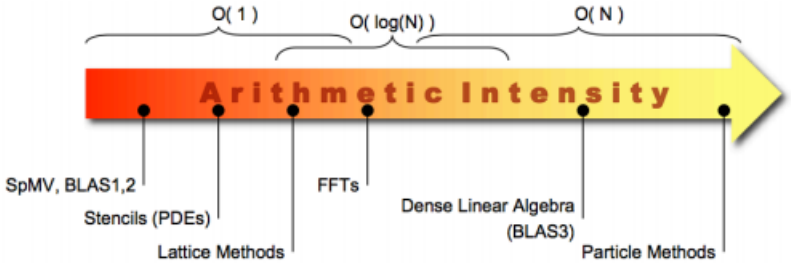


Figure 3.5.: Arithmetic intensity for various domains of computing problems.

Figure 3.5 presents the complexity for various computational purposes. The problems with *O(1)* complexity are not likely to get any benefit from parallelization, opposed to problems with *O(N)* complexity that is more easy to extract performance from parallelization.

The rest of the computational characterization resorted to the analysis of hardware performance counters, using the Performance API [31], and was performed for the application with no and 256 variations, on the compute-601 system. The instruction mix is presented in figure 3.6. By the analysis of the charts is evident that this functions performance cannot be evaluated using the 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 number of variations does not have a significant impact on the type of instructions issued.

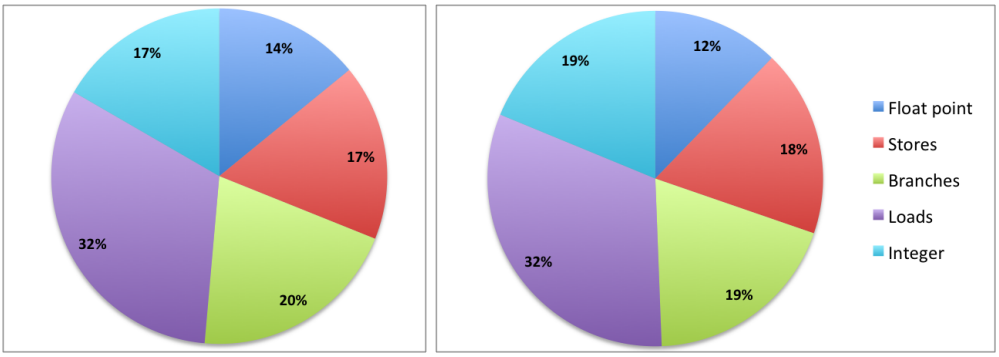


Figure 3.6.: Instruction mix for the `ttDilepKinFit` with no and 512 variations, left and right images respectively.

The miss rate, for the 3 cache levels, is presented in figure 3.7. The miss rate on L1 cache remains constant in 70% when changing the number of variations. This value is high but it is justified by the huge amount of different computations necessary to reconstruct an event, where data is only reused between one instruction to the other (considering a small scope due to the small cache size), rather than reused by a large set of instructions. 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 a miss to the L3 cache. Due to the reuse of data in the L2 cache, and considering the large size of the L3 cache, the miss rate 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 fetch

instructions of new data not yet on cache. 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.

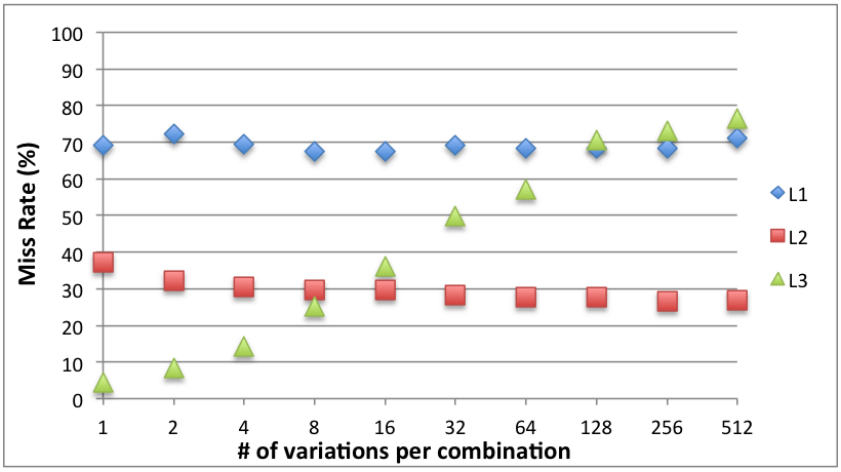


Figure 3.7.: Miss rate on L1, L2 and L3 cache of `ttDilepKinFit` for various number of variations.

Operational intensity [38] is a performance metric to characterize the bottlenecks of a given algorithm, based on the assumption that accesses to the system RAM memory are the main performance limitation, also classifying it as memory or compute bound. It requires a Roofline model[3] for the system on which the operational intensity is calculated. The operational intensity is the amount of float point operations performed per each byte of data that is 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` is float point arithmetic (refer to chart 3.6). Instead, the computational intensity is considered a more reliable metric as it considers all kinds of instructions, with the exception of loads and stores which are not relevant for the algorithm structure.

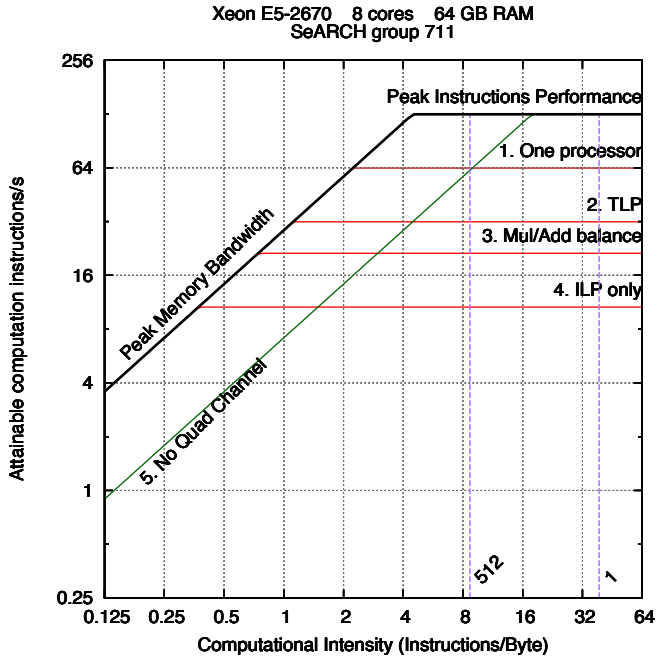


Figure 3.8.: Roofline of the compute-601 system with the computational intensity of `ttDilepKinFit` for 1 and 512 variations.

[3]Roofline models for the test systems are presented and explained in appendix Appendix A.2.

In figure 3.8 is presented the Roofline model for the compute-601 system with the computational intensity for `ttDilepKinFit` with no and 512 variations. For both number of variations 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 memory accesses. Note that this not include optimizations for accessing the CPU cache.

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 running a high number of variations, as seen in callgraph 3.3. From an analysis of the source code is seen that for each variation is needed at most 12 PRNG values. These values are generated and then it is applied a transformation to them so that they follow 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 data file used[4] there are 1867 events reach the cut 20. This translates in 244056 combinations reconstructed, with a total of 2928672 PRNG values. This is likely to happen with other input data files as each holds a similar number of events. The `TRandom` generator is being reset with a new seed in every combination, justifying the 23.4% of execution time for the `SetSeed` function.

The pseudo-random number generator used by the `TRandom3` class is the Mersenne Twister [39], currently one of the most used generators for applications highly dependable on 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 in different threads. The state can be shared among the threads but, however, changes to it must be done sequentially, serializing the number generation. In this case, the number generated by one thread will affect the number generated by the remaining, and it is not safe to implement in a parallel application.

Since the period of the Mersenne Twister is 2^{19937} (approximately $4.3 * 10^{6001}$) and the maximum amount of numbers generated, for 512 variations per combination, is $1.5 * 10^9$, it is not necessary to reset the seed of the `TRandom3` generator, which can reduce the `ttDilepKinFit` execution time significantly. 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 credibility of the results.

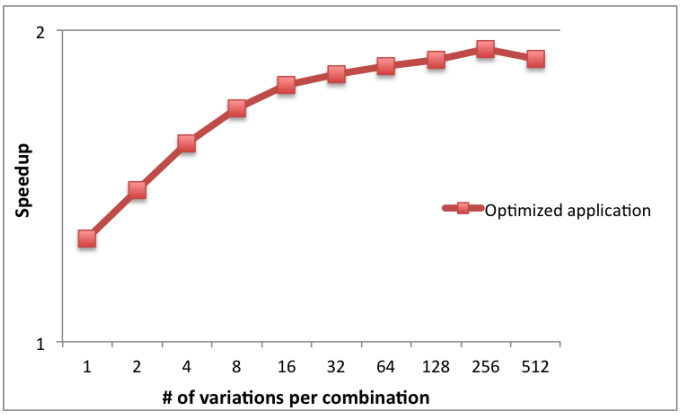


Figure 3.9.: Speedup of the `ttH_dilep` application with the `TRandom` optimization.

NVidia offers a parallel implementation of the Mersenne Twister for GPUs [40] in the cuRand library

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

[41], important when 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, but offering a smaller period of 2^{11213}. The pseudo-random number generation and state update is thread safe, with up to 256 threads sharing the same state for each block. Two different blocks can safely operate concurrently.

`TRandom` uses the Acceptance-Complement Ratio algorithm [42] for transforming the pseudo-random numbers from an uniform to a gaussian transformation. It is allegely 66% faster than the Box-Muller transformation [43] and similar to the Ziggurat method [44]. 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, dividing 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 increase code readbility. Also, the definition of the number of variations to perform is set by environment variables, at runtime, rather than at compile time, easing the user interaction with the application.

4. Parallelization Approaches

As presented in chapter 3, the critical region of the `ttH_dilep` application is the `ttDilepKinFit`, which its 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 resort to parallelization. This chapter presents four parallelization approaches, two on homogeneous systems and two on heterogeneous systems, one using GPU and other using Intel Xeon Phi as hardware accelerators.

Although the critical region is the `ttDilepKinFit` function, the best parallelization strategy is to simultaneously perform the reconstruction of different events. All event processing is data independent so no synchronizations are necessary, 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 to the application and partially in the LipMiniAnalysis library. Every time an event is loaded the global data is overwritten, which, in a parallel shared memory environment, causes the intermidiate results of the event processing of one thread to be overwritten by the processing of other thread.

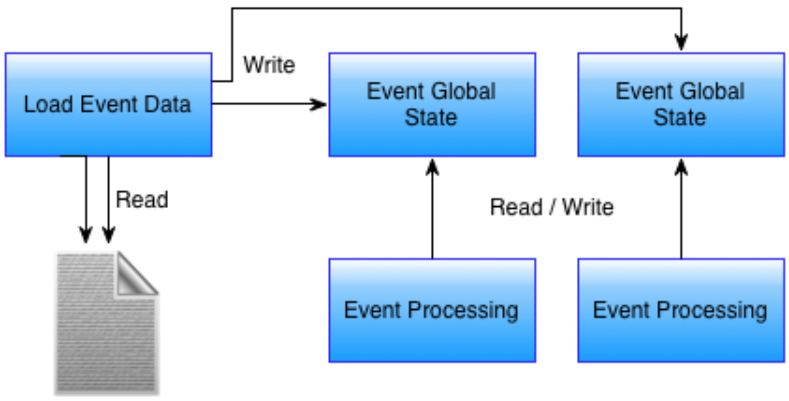


Figure 4.1.: Schematic representation of the event-level parallelization model.

Each input file has around 1 GByte that makes possible to store all events on RAM memory. The use of a proper data structure for storing the event information would benefit the application structure and allow the possibility of this parallelization approach. In the current implementation the event are loaded before the processing. The application would benefit from the sequential read performance of 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.

The `ttDilepKinFit` function mostly handles with data in its scope. The global data modified by this function can be managed without any modifications in the LipMiniAnalysis. Looking at the information presented in section 3.2.1, specifically in figure 3.4, can be expected that most performance gains can occur for 16 to 512 variations per combination as `ttDilepKinFit` occupies most of the application execution time, rather that for a smaller number of variations.

The `ttDilepKinFit` execution is irregular, meaning that dependent on the event to be reconstructed its execution time varies. This is caused by the number of Bottom Quark jets and leptons combinations, which is different for each event. Also, it is dependent on the kinematical reconstruction. If the $t\bar{t}$ system is possible to reconstruct, `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 much more likely to affect the performance if the load balancing strategy is not suited for the problem.

The kinematical reconstruction is performed in the `dilep` function. The $t\bar{t}$ system obeys a set of properties expected from the Top Quark theoretical model. To reconstruct both of the Top Quarks it is needed 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, it is needed to infer them, using various properties, such as momentum and energy conservation of the system. Once the neutrinos characteristics are determined, it is possible to reconstruct the Top Quarks. `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, making it easy to port to GPU. Also, it is the function that only handles data private to the function.

4.1. Shared Memory Parallelization

Figure 4.2 (left) illustrates the 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 every combinations chosen so far. Also, the combinations information is stored globally to the application so a data structure must be created to allow parallel reconstructions of different combinations.

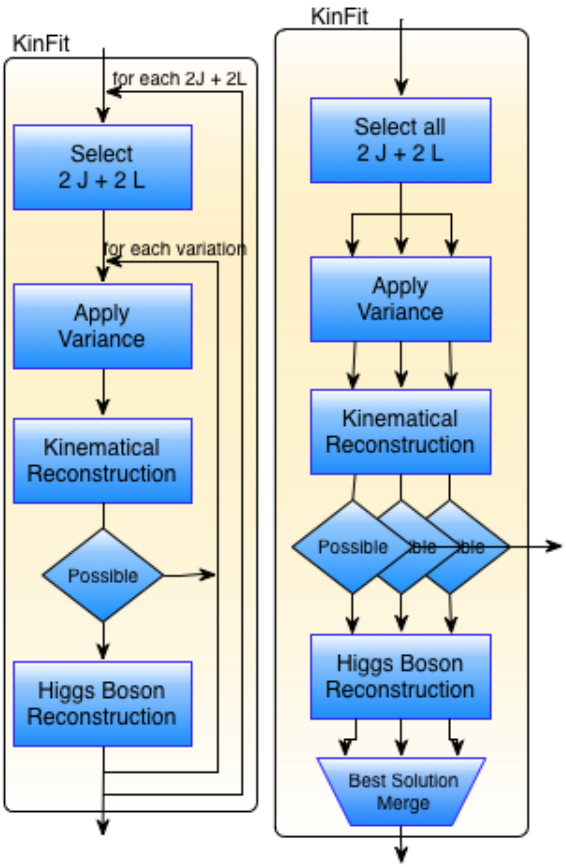


Figure 4.2.: Schematic representation of the `ttDilepKinFit` sequential (left) and parallel (right) workflows.

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 combination. This small granularity allows for the scheduler responsible for the load balance to better distrubute the work among the threads

used. Each task has a combination assigned and it applies a variation and reconstructs it. The result is stored in a state private to the task so that, after all variations of all combinations are computed, only the best solution is considered, performed by a reduction. 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.

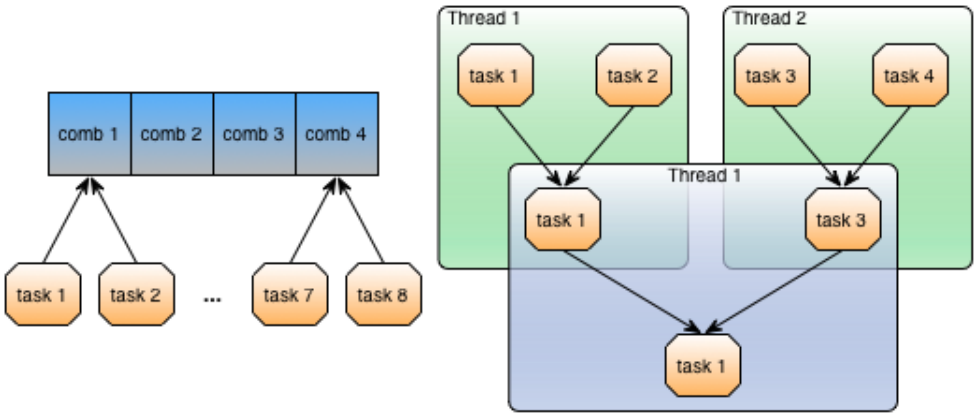


Figure 4.3.: Schematic representation of the parallel tasks accessing the shared data structure (left) and the new parallel reduction (right).

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 figure 4.2 (right). As stated before, chosing the combinations and building the data structure cannot be performed in parallel. This reduces the parallelizable region and adds an extra overhead to each `ttDilepKinFit` call. Also, the best solution merge, which does not happens in the sequential application, increases this overhead. This process is repeated for each event.

4.1.1. Implementation

The implementation was performed iteratively, where every major change impact 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 big global state, small modifications, specially when introducing parallelism, can completly alter the application behavior. OpenMP library was used to implement the parallelization as it is the most used among scientists and easier to include in C-like applications such as `ttH_dilep`. It offers many functionalities required by the 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.

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 application performance was not a concern.

The first task performed by `ttDilepKinFit` is chosing the variations. This can be performed while building the data structure holding its information, and must be a serial process since chosing a combination depends on all previous choices. 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 XX.

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

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, and, on average, each event has 131 combinations, making the average data structure size 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 does not affect the size of the data structure.

One of the major problems of the parallelization is controlling the access to the global state. In `ttDilepKinFit`, 34 global variables of mostly ROOT classes instances and vectors, are read and write 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 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), so that the parallel tasks do not have any data dependencies, avoiding serial accesses to shared resources that can cause contention and degrade the performance.

Only the best solution is needed after reconstructing all variations of all combinations for an event. The best solution, i.e., information resultant from the event reconstruction, can be measured by computing its quality. The result is a scalar value and the higher the value the better the reconstruction. The best solution is a set of 16 LipMiniAnalysis `TLorentzVectorWFlags`, an extension to the ROOT `TLorentzVector` class, and the scalar with the solution quality. To increase the modularity of the code a class was created that holds all this information and implements all comparator operations. This increases the implementation overhead but reduces the complexity of the reduction process. Since OpenMP only supports parallel reduction for scalars, 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 in the parallel reduction. 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 they compare their solutions. The thread with the lower id keeps the best solution of the two. Threads which do not have a pair check a queue of threads with no pairs, and pick a pair. They are put in the queue if it is empty. An example reduction is presented in figure 4.4.

As mentioned in section 3.2.2, the 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 and they would have to be serialized. 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.

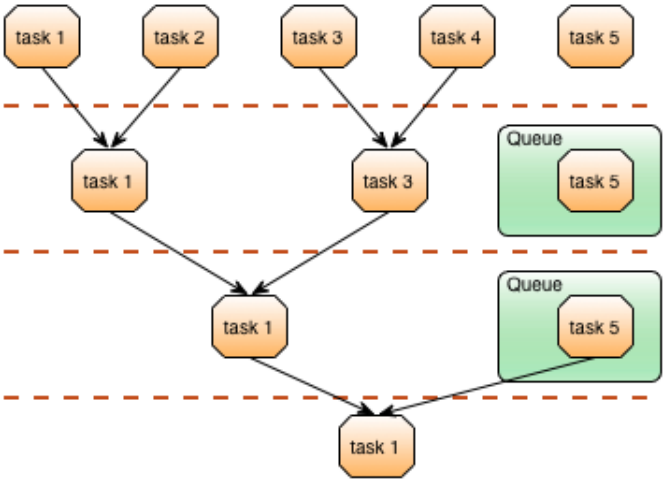


Figure 4.4.: Schematic representation of the event-level parallelization model.

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 its execution. This scheduler requires a small amount of execution time and is very efficient for balancing regular workloads. For irregular workloads, such as `ttDilepKinFit`, the static scheduler porduces a poor workload balance, which can 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 VTune, the constructor of the data structure was identified as the main performance bottleneck. When analyzing the data structure source code, as well as `ttDilepKinFit`, its evident that some of its parameters are only read and not modified during each event processing. Instead of copying these parameters in each data structure element a it is possible to use 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 two CPUs, as threads in one CPU may have to access data in other CPU, instead of holding a copy of the information in cache, which is possible when not using pointers. The performance of this implementation, which will be addressed as pointer version, is compared against the previous implementation 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 of evaluating performance will be used, such as speedup and throughput, and the results will be analyzed and discussed. In this section the test system used is the compute-711 node if no other information is given.

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 4.8 to compare 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 affect slightly

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

the performance, specially for a small number of variations.

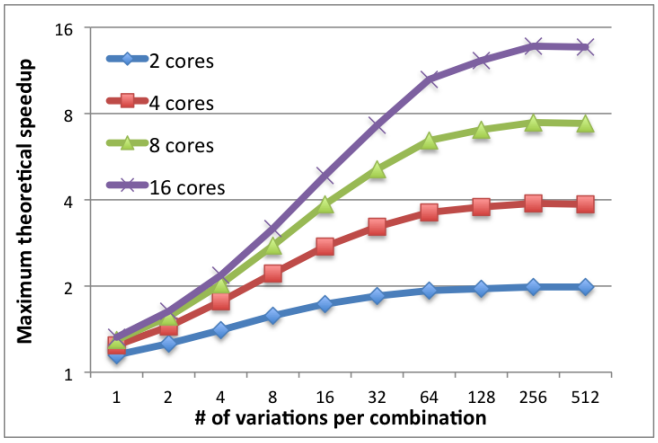


Figure 4.5.: Theoretical speedup (Amdahl's Law) for various number of cores.

The maximum theoretical speedup calculation, presented in figure 4.8, is based on the Amdahl's Law [45] which states the maximum attainable speedup based on the number of processors and the percentage of the execution time spent on the parallelizable region of the code. The efficiency of the parallelization of an application can be measured by the distance of the speedup curve to the 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 resource usage.

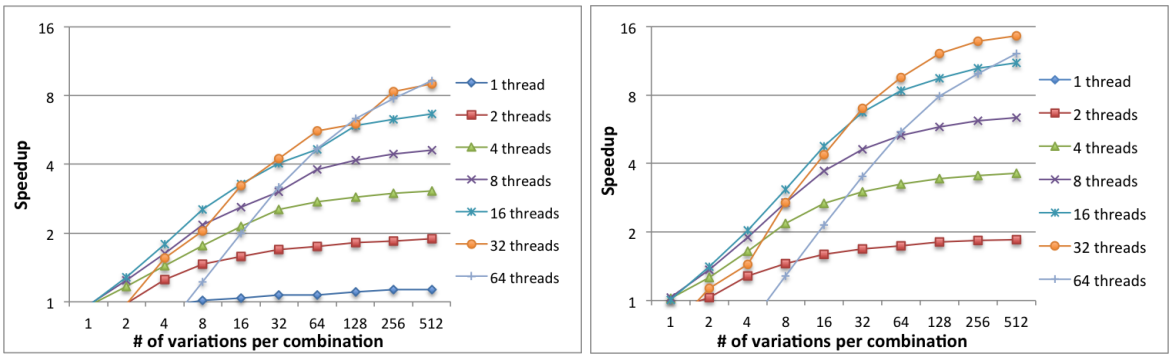


Figure 4.6.: Speedup for the parallel non-pointer version of `ttH_dilep` application with static (left) and dynamic (right) scheduling.

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 only 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) offers a speedup of 6.7 and 9.2, for the static and dynamic scheduler versions respectively, in 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 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.

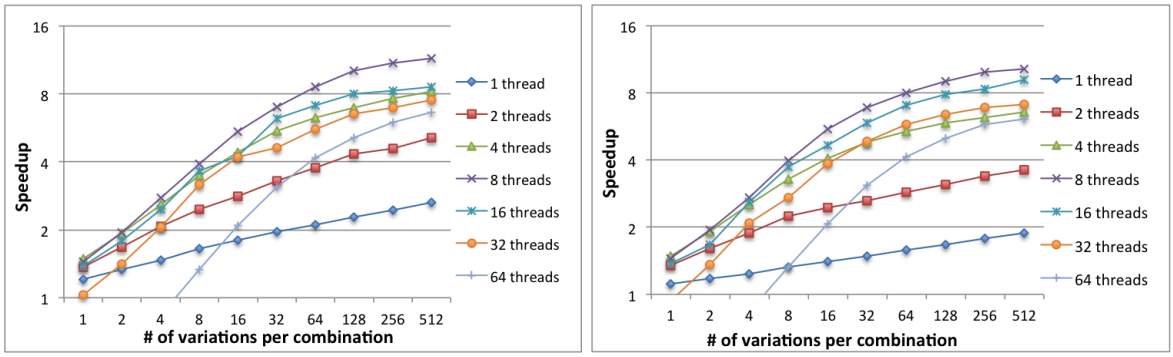


Figure 4.7.: Speedup for the parallel pointer version of `ttH_dilep` application with static (left) and dynamic (right) scheduling.

Similar to the non-pointer version, the dynamic scheduling provides the best performance for high number of threads. 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 result is 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, as opposed to the higher overhead of the non-pointer version.

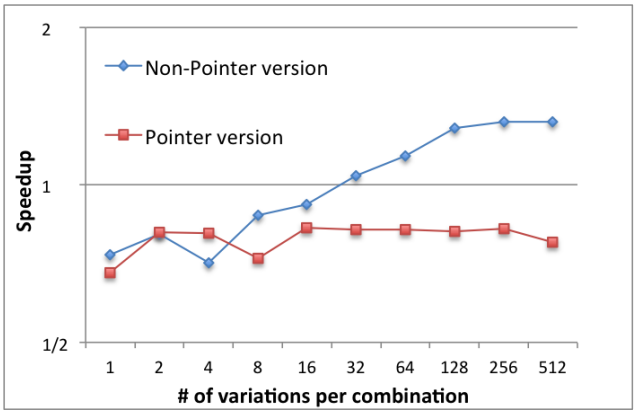


Figure 4.8.: Theoretical speedup (Amdahl's Law) for various number of cores.

The speedup provided by the use of hardware multithreading, which helps to improve the CPU resource usage by scheduling by hardware two threads in each 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, as explained before. In the non-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 increase of threads used, as long as more than one CPU is used.

Considering only the dynamic scheduler non-pointer and the static scheduler pointer implementations, which present the best performance, 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, as explained in section 4.1.1. 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

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

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 said data is only in one memory bank, forcing non-unified memory accesses that degrade the performance of the threads in one of the CPUs, and the data cannot be properly stored in the cache as the use of pointers prevents a vast set of memory management optimizations. Chosing the best implementation depends on the system, as 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 possible bottlenecks and the optimization process stoped as the results are very close to the theoretical maximum.

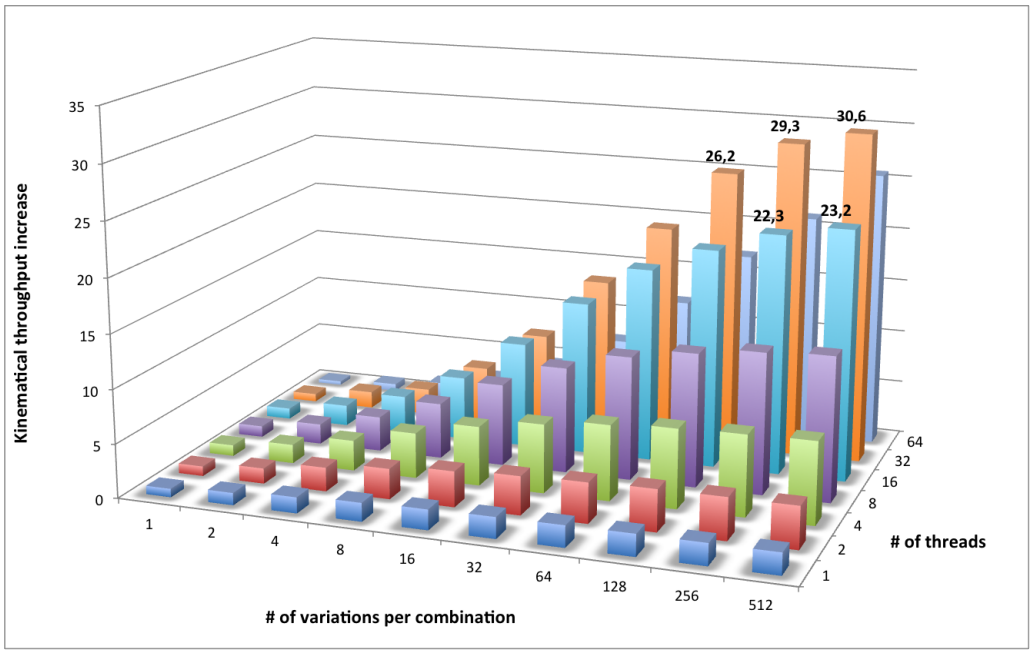


Figure 4.9.: Theoretical speedup (Amdahl's Law) for various number of cores.

One possible metric to measure the increase of throughput in the `ttDilepKinFit` is the increase of kinematical reconstructions performed relatively to the sequential application. Only this part of the function is suitable for this purpose as it is always executed for every variation of any combination, opposed to the Higgs Boson reconstruction which is only performed if the $t\bar{t}$ system is reconstructed. The speedup of the kinematical reconstruction versus 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, irregular `ttDilepKinFit`.

One of the concerns for research groups is the amount of events that their applications are able to process. Figure 4.10 presents the event throughput of the non-pointer implementation with the dynamic scheduler, with the purpose of identifying the maximum throughput possible due to parallelization. An event processing is considered to be all events in the input data file, as well as each individual variation reconstructions of those who reach cut number 20 of `ttH_dilep`. The sequential, already optimized, application is presented in the ̈I threadl̈ine on the graph. throughput is For 512 variations and 32 threads the event throughput is 14300 events per second, which is 15 times higher than the original for the same number of variations.

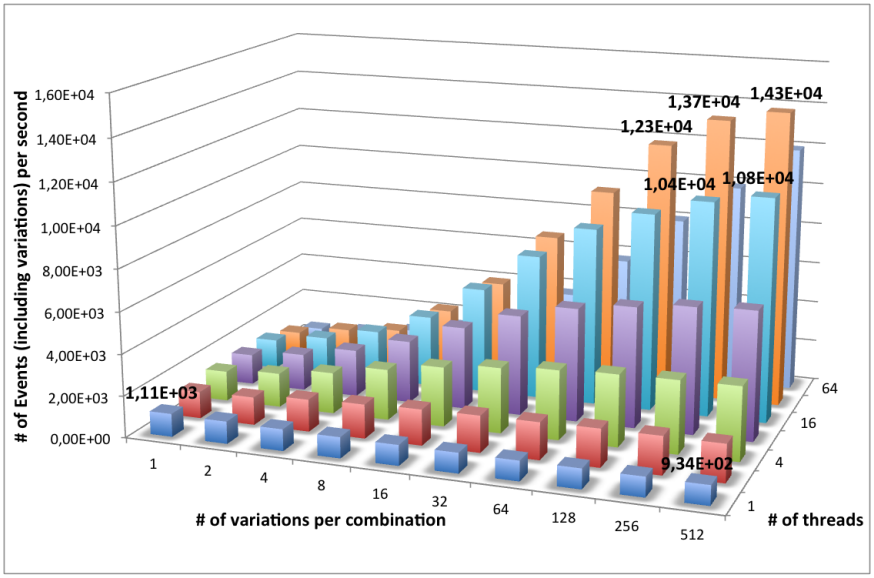


Figure 4.10.: Theoretical speedup (Amdahl's Law) for various number of cores.

Performance analysis on various computational systems

Scientific computer clusters are not always made of high end computing nodes, such as the compute-701 test system used. 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 for the compute-701 system in section 4.1.2.

Only the two best implementations, non-pointer with dynamic scheduler and pointer with static scheduler, 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 4.11.: Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-401 node.

4.2. GPU Parallelization

An early analysis of the code was made before designing the the workflow for the GPU parallelization. The implementation of this version will be restricted by the dependencies that `ttDilepKinFit` has on ROOT classes, namely on its third stage of the generalist parallelizable workflow (figure 4.2, right). It uses several functions and classes from ROOT, which can possibly be adapted to the GPU but the amount of time and work necessary to do so makes it unviable. The kinematic reconstruction also uses ROOT classes, namely TLorentzVectors, however it is read-only so it can be transformed in a data structure fit to be used on GPU. Note that this transformation will have a cost associated, which can slightly affect the performance.

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

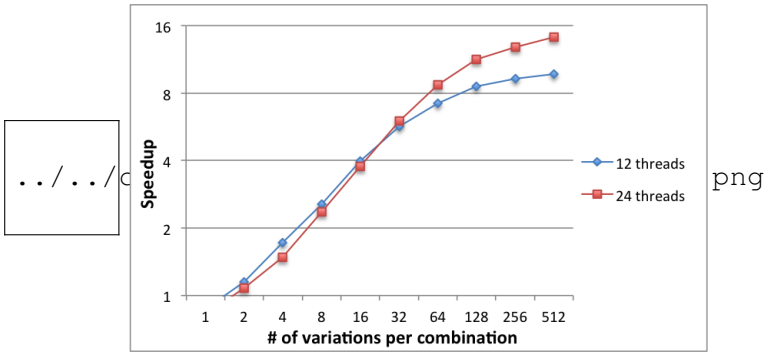


Figure 4.12.: Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-511 node.

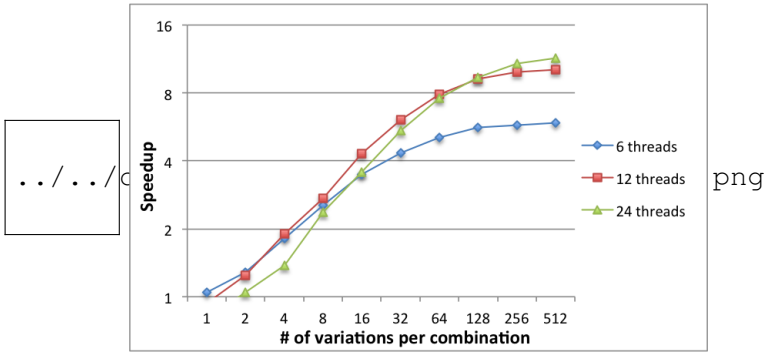


Figure 4.13.: Speedup of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations in the compute-601.

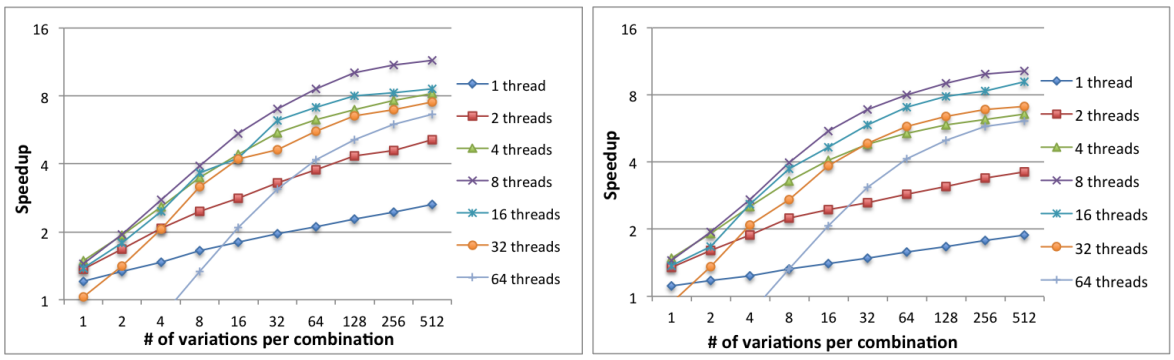


Figure 4.14.: Execution times of the `ttH_dilep` application for pointer static (left) and non-pointer dynamic (right) scheduler implementations.

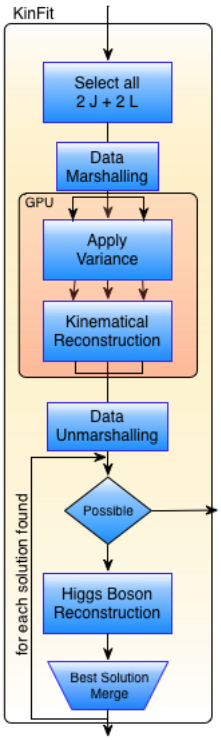


Figure 4.15.: Schematic representation of the `ttDilepKinFit` workflow.

The first two stages of the the workflow presented in figure 4.2 (right), the computation of the variances and the kinematical reconstruction, can be adapted to run on the GPU. After computing the data structure holding the jets and leptons combinations, it must be transferred to the GPU (device) memory and then launch the kernel, where each task (which is refers to one combination) is assigned to one thread. The variation of the combinations is done by each thread, on the assigned combination, the kinematical reconstruction is computed and the results are transferred back to the CPU (host) memory. Then, the third stage of the workflow, the Higgs boson reconstruction, is performed on the host. Note that this process, copying the memory to the device and back to the host is done one time for each event processed. The schematic representation of the workflow for this implementation is presented in figure 4.15.

This implementation has two factors which will restrict the performance. The first is the overhead associated to the transformation of the data (ROOT classes) to a suitable data structure to be used by the device. This happens with the input and output of the kernel. Even though this process can be tuned, which will be explained in section **??**, there is no alternatives to study, algorithm-wise. The second factor is the synchronization and data transfer between host and device. The transfer time is affected by the amount of data to transfer, but it cannot be reduced since it is always necessary to transfer the jet/lepton combinations. Note that they are only transferred once per event, where the kernel threads copy the information so they can change it. However, the synchronization can be removed. The kernel can be launched and the threads are blocked while waiting for work, and then each time the one thread of the host computes a combination it is transferred to the device memory and the respective threads start the computation. Meanwhile, there is another thread in the host waiting for the results and starting the parallel computation of the Higgs boson reconstruction each time a group of kernel threads finish. If this asynchronous communication can be correctly implemented it might offer significant better performance than the synchronous version.

4.3. MIC Parallelization

4.4. Scheduler Parallelization

throughput scheduler

Appendix A. Test Environment

This appendix focuses on fully 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 used in small physics research group clusters. The first, named compute-711 node, has two Intel Xeon E5-2670 (Sandy Bridge architecture) **??**, 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 **??**.

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, have 8 GB RAM with a maximum bandwidth of 18.4 GB/s. The CPUs are Intel Xeon E5520 (Nehalem architecture). The two CPUs are interconnected by a QPI interface with the same bandwidth as the previous system, 24 GB/s.

The fourth system features two AMDOpteron 6174, being the system with more 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 Appendix A.1.

The compiler used was the GNU compiler version 4.8, using the -O3 optimizations and the AVX/SSE 4.2/SSE 4a (depending on the CPU architecture) instruction set on the code regions that 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 **??** 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`.

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

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

Appendix A. Theoretical Performance Models

Appendix A.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 [45] 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.

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

Equation Appendix A.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 A.2. Roofline Model

The Roofline model [38] 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 maximum attainable performance. The classic Roofline uses float point computation as the peak CPU performance metric, which is usually advertised as peak performance by CPU manufacturers. It may be a good metric for heavy computational algorithms, such as matrix multiplication, but the type operations on the critical region (`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 Appendix A.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.

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

The tilted ceiling 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 Appendix A.

Figure 3.8 illustrates the Roofline model for the four systems used.

Roofl

Appendix A. Test Methodology

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

All performance measurements, of both the original and parallel algorithms, were made on binaries compiled with the same compiler and same flags, presented in section Appendix A. All tests used the same input, 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 $x \in \{1, ..., 9\}$.

For the shared memory implementation was used different number threads, depending on the system. The test using 1 thread has the purpose of evaluating the overhead of the creation and access to the data structures. 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 when accessing memory, specially on NUMA accesses. The number of threads equal to the number of cores in one CPU, with one thread per core, is to test 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 is so that both CPUs are used, meaning that the memory accesses are now NUMA, but still not using hardware multithreading. The number of threads equal to the available hardware threads is to test both CPUs with hardware multithreading active.

In the GPU the number of threads used was the number of variations times the number of combinations, so that each thread computes a variation of a combination. This way there is a high number of threads to hide the memory access latency of the GPU.

It is important to adopt a good heuristic for choosing the best measurement since it is not possible to control the operating system and other background tasks necessary for the system, which can occasionally interfere with the measurements. The mean value is very sensitive to extreme values, i.e., the cases when the system may have a spike on the workload from other OS tasks and greatly affect the measurement will have a big impact on the mean, not truly 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, even if a small subset of great measurements was made. Choosing only the best measurement, with the lower execution time, is not a solid heuristic, since it is more complex to replicate the result.

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

To measure the total execution time of the application was used the `gettimeofday` function from the C standard libraries, providing microsecond precision, which is enough considering that the fastest execution of the application with the defined inputs without any variation takes around 4 seconds. For the measurements of only the portion of the code executed on the GPU was used CUDA Events to ensure that the times were properly recorded and synchronization of the kernels and memory transfers are ensured.

References

[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. 1).

[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. 1).

[13] European Organization for Nuclear Research. *The Worldwide LHC Computing Grid*. 2013. URL: http://wlcg.web.cern.ch/ (cit. on p. 1).

[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. 8).

[16] Intel. *The Intel® Xeon PhiTM Coprocessor 5110P*. Tech. rep. 2012 (cit. on pp. 9, 12).

[17] Texas Instruments. *Digital Signal Processors*. 2012. URL: http://www.ti.com/lsds/ti/dsp/overview.page (cit. on pp. 10, 13).

[18] TOP 500. *June 2013*. 2013. URL: http://www.top500.org/lists/2013/06/ (cit. on p. 10).

[19] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architectur: Fermi*. Tech. rep. 2009 (cit. on p. 10).

[20] NVIDIA Corporation. *Tegra*. 2013. URL: http://www.nvidia.com/object/tegra.html (cit. on p. 13).

[21] Sixto Ortiz Jr. "Chipmakers ARM for Battle in Traditional Computing Market." In: *Computer, 44(4):14-17* (2011) (cit. on p. 13).

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

[23] James Reinders. *Intel Threading Building Blocks*. Tech. rep. 2007 (cit. on p. 14).

[24] Massachussets Institute of Technology. *The Cilk Project*. 2013. URL: http://supertech.csail.mit.edu/cilk/ (cit. on p. 14).

[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. 15).

[26] OpenACC Corporation. *OpenACC*. 2012. URL: http://www.openacc-standard.org/ (cit. on p. 16).

[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. 16).

[28] OpenACC Corporation. *How does the OpenACC API relate to OpenMP API*. 2013. URL: http://www.openacc-standard.org/node/49 (cit. on p. 16).

[29] João Barbosa. *GAMA framework: Hardware Aware Scheduling in Heterogeneous Environments*. Tech. rep. 2012 (cit. on p. 16).

[30] Intel. *Profiling Runtime Generated and Interpreted Code with Intel VTune Amplifier*. Tech. rep. 2013 (cit. on p. 17).

[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. 17, 22).

[32] Free Software Foundation. *GDB: The GNU Project Debugger*. 2013. URL: http://www.gnu.org/software/gdb/ (cit. on p. 17).

[33] NVIDIA. *CUDA-GDB: The NVIDIA CUDA Debugger User Manual*. Tech. rep. 2008 (cit. on p. 17).

[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 p. 18).

[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. 18).

[36] Intel Corporation. *Intel Math Kernel Library*. 2013. URL: http://software.intel.com/en-us/intel-mkl/ (cit. on p. 18).

[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. 18).

[38] 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. 23, 40).

[39] 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. 24).

[40] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". In: (2012) (cit. on p. 24).

[41] NVIDIA. *CUDA CURAND Library*. Tech. rep. 2010 (cit. on p. 25).

[42] W. Hoermann and G. Deringer. "The ACR Method for generating normal random variables". In: *OR Spektrum 12, 181-185* (1990) (cit. on p. 25).

[43] 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. 25).

[44] George Marsaglia and Wai Wan Tsang. "The Ziggurat Method for Generating Random Variables". In: *Journal of Statistical Software* (2000) (cit. on p. 25).

[45] 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. 31, 40).