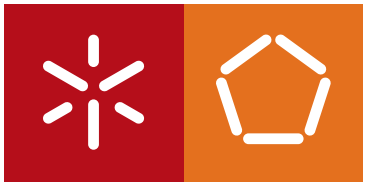
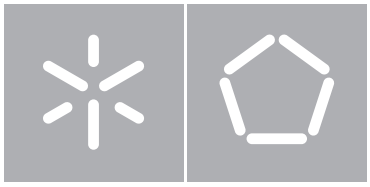


Universidade do Minho
Escola de Engenharia

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

Glossary

Event head-on collision between two particles at the LHC

LHC Large Hardron Collider particle accelerator

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

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

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

HEP High Energy Physics

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

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

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

GPU Graphics Processing Unit

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

DSP Digital Signal Processor

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

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. Introduction

1.1. Motivation

2. `ttH_dilep` Application

The CERN computational resources, the Worldwide LHC Computing Grid, are organized in a hierarchy divided in 4 tiers. Each tier is made by one or more computing centers and each tier has a set of specific tasks and services to perform. The objective of the whole grid is to store, filter 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. This tier distributes the raw data and the reconstructed output by the 11 Tier-1 computational centers, spread among the different countries that are part 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 that is then transferred to the Tier-2 computational centers. 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.

The LIP research group developed the `ttH_dilep` application to solve the problem presented in section 1.1, and it fits in the Tier-3 hierarchy level of event reconstruction and analysis applications. Its name derived from the problem it was design to solve: the tt is relative to the kinematical reconstruction of the two Top Quarks, the $t\bar{t}$ system, resultant from a head-on particle collision; the H is relative to the Higgs boson reconstruction; the dilep is the name of the routine responsable for the kinematical reconstruction, and it needs two leptons (di-lep) as input.

The application has two main dependencies. The first, and most important, is on the ROOT **??** object oriented framework, developed at CERN, only available in C++. This framework provides a set of funcionalities oriented for handling, analyzing and displaying results for large amounts of data. It has capabilities of reading and storing data in the standard formats accepted by all the tiers centers, classes for representing physics information, mathematical routines, pseudorandom number generators, histograming, curve fitting minimization and visualization methods. It was originally designed and currently developed mostly by physicists with little knowledge on computer science. This results in a framework that has much room for improvement through a code restructuration in several routines, mostly related to auxiliary functionality, rather than visualization and data storage. Some of the mathematical routines implemented could be replaced by dependencies on other much more stable and faster libraries, such as BLAS **??** or MKL **??**. There is an extension to ROOT, the Parallel ROOT Facility (PROOF) **??**, for parallelization of the work in distributed memory systems, which is not the focus of this thesis work. There is no support nor existing routines parallelized for shared memory systems, which could be made but would require restructuration of portions of the framework code.

The second dependency is on the LipMiniAnalysis library. It is a strip-down version of LipCbrAnalysis, a library developed LIP for in-house use, which provides a skeleton for creating an analysis application. It has the functionality usually necessary in most analysis developed by LIP, and is also prepared for reading a data format different from what is provided by Tier-2, which suffers a filter of the events most likely to

provide relevant information after the reconstruction. This library is also not designed for parallelization in shared and distributed memory systems.

2.1. Application Flow

This section describes the workflow of the `ttH_dilep` analysis. The callgrind tool from Valgrind **??** was used to obtain the callgraph of the application, also providing some insight on the time that was being spent in each of its routines. Further analysis of the code itself was necessary to get a better understanding of the application behaviour. In figure 2.1 is presented the callgraph of `ttH_dilep` for 128 variations per combination.

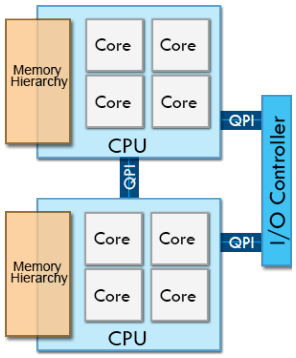


Figure 2.1.: Schematic representation of a NUMA system with QPI interface.

The `Loop` is the main method of the application. Its purpose is to iterate through all events in the input file and perform 21 filtering and processing tasks (known as cuts) to each event. The most evident problem with the application, inherited from the LipMiniAnalysis library, is the non existence of a data structure on memory holding the events to process. Each input file has around 1 GByte that makes perfectly possible to be stored on RAM memory. However, for each event its information is read from the file and loaded to hundreds of global variables and then it is submited through the cuts. If all the events were read at once, it would be possible to take advantage of the higher bandwidth of sequential reads to the hard drive. Even the overhead of creating such data structure for the events would be compensated by the faster acesses and possibility of easier parallelization of the analysis at the event level, since the events have no dependencies between them.

Every method starting with a capital T refers to the ROOT framework. They are only being used for reading the input file and writing the results. The `DoCuts` method performs the 21 cuts referred above. It is on cut 20 that `ttDilepKinFit` is called, which becomes the most time consuming task as the number of variations per combination increases, as seen from table 2.1, therefore, the efforts on improving the performance must be focused on this routine.

of variations/combination	1	2	4	8	16	32	64	128	256	512	1024
% of time	-	-	-	-	-	-	-	-	-	-	-

Table 2.1.: Percentage of the total execution time spent on the `ttDilepKinFit` routine for various numbers of variations per combination.

2.2. ttDilepKinFit Routine

This routine has a main loop that calculates and iterates through all the possible combinations of jets and leptons for a given event. It is possible to define the number of variations to perform per combination, resulting in an inner loop. Some of the variables varied are local to the routine but most are global to the application. The kinematical reconstruction is performed inside the loops, for each variation of each combination and its results, as well as the bottom quarks not used, are used later in the Higgs boson reconstruction. A context is created when the combination to process is calculated, and its specific for the respective combination, which is altered by the variations, kinematical and the Higgs boson reconstructions. Each one of the solutions is stored in a vector and after all the combinations are computed, the said vector is iterated and only the best solution is choosen and its relevant data is copied to global variables.

The quality of a solution is dependent on two factors. The first is the accuracy of the kinematical reconstruction, which is measured by the probability of the reconstruction being properly performed, compared to theoretical models. The second is the accuracy of the Higgs boson reconstruction and its calculation is similar to the former. Since the Higgs boson reconstruction uses results of the kinematical reconstruction, namely the neutrinos and remaining bottom quarks, if the latter is faulty, i.e., performed with a low resultant accuracy, the Higgs boson will be poorly reconstructed too. The accuracy of the overall reconstruction of the event, for a given variance of a combination, is given by the probability of the kinematical reconstruction times probability of the Higgs boson reconstruction.

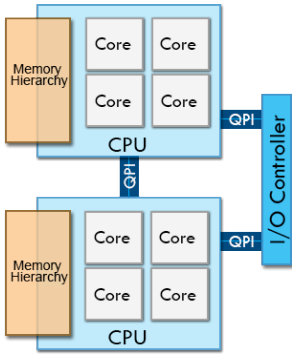


Figure 2.2.: Callgraph of the `ttDilepKinFit` routine.

Note that the flow of `ttDilepKinFit` may change between combinations, or even variations, within an event. The responsible of that behaviour is the `dilep` routine. A certain combination may be capable of being reconstructed, both the $t\bar{t}$ system and the Higgs boson, but with others the $t\bar{t}$ system may not be reconstructable, which causes the `ttDilepKinFit` flow to stop and process the next combination. This irregularity of the time that takes to process a combination can be problematic to the load balancing of the parallel tasks, as explained in section 3.

The callgraph for the `ttDilepKinFit` for 128 variations per combination is presented in figure 2.2. The sections of the routine that are most relevant, the variation of the events parameters, the kinematical and Higgs boson reconstructions, are explained in depth in the next subsections.

2.2.1. Variations Routine

The purpose of the variation is to overcome the experimental resolution of the ATLAS detector, as explained in section 1.1.The variation of the parameters of an event, for a given combination of jets and leptons, consists on applying an offset of a given magnitude to the said values. The offset is randomly

obtained, using the TRandom3 pseudo-random number generator from ROOT, following a gaussian distribution. The mean value used is 0 and the standard deviation is 2%, as it is the resolution error of the detector associated to every measurement. The values varied are the momentums of the 2 jets and 2 leptons that make the combination, and consequently their energy is recomputed.

The pseudo-random number generator used by the TRandom3 class is the Mersenne Twister **??**, 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, change it must be done sequentially, sequentializing the number generation among the threads. In this case, the number generated by one thread will affect the number generated by the remaining.

NVidia offers a parallel implementation of the Mersenne Twister for GPUs in the cuRand library **??**. 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, up to 256 threads sharing the same state structure, for each block. Two different blocks can safely operate concurrently.

TRandom uses the Acceptance-Complement Ratio algorithm **??** for transforming the pseudo-random numbers from an uniform to a gaussian transformation. It is allegely 66% faster than the Box-Muller transformation **??** and similar to the Ziggurat method **??**. 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.

2.2.2. `dilep` Routine

The kinematical reconstruction is performed in the `dilep` routine. The $t\bar{t}$ system obeys a set of properties of an theoretical expected 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 the properties of 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 routine is dependent on only one class from ROOT, TLorentzVector, making it easy to port to GPU, aside from the process of marshaling and unmarshaling the data, namely the TLorentzVector classes from ROOT, into a format that is usable in CUDA. Executions of `dilep` on different inputs are completely independent, since this function does not alter the global state of the `ttH_dilep` analysis.

2.2.3. `ttDilepKinFit` Routine Computational Analysis

Coisa

3. Parallelization Approaches

The section of code which takes more time is the `ttDilepKinFit` function. The main objective is to run as many kinematical reconstructions per event, with a slight variation to the particle characteristics, and as they increase the `ttDilepKinFit` execution time also increases. So, since it is the critical section of the application, the efforts on performance optimizations will be focused on this portion of the code.

The `ttDilepKinFit` workflow can be divided in three main stages. Each event can have an arbitrary number of jets and leptons associated, requiring a minimum of two of each to perform the kinematical reconstruction of the $t\bar{t}$ system. Events that not fulfill this requirement are discarded in the previous cuts. If there is more than the minimum number of jets and leptons it is necessary to combine them, in pairs of two jets with two leptons, and perform the kinematical reconstruction for every combination possible. Note that their order on the combination is not relevant, reducing the total amount of possible combinations. Then, it is possible to apply a variation to the jets and leptons characteristics, motivated by the reasons explained in section 1.1. The variation has a magnitude equivalent to the experimental resolution of the ATLAS detector (which is 5%) and it is applied to the three momentums and energy of the particles, and causing the need to re-compute other auxiliary parameters to the rest of `ttDilepKinFit`. The number of variations to apply to each jet/lepton combination is arbitrary and defined by the user.

`ttDilepKinFit` has a main loop, for each jet/lepton combination and for each variation of each combination, where the most intensive computation occurs, which will be explained next, and a final section of code that iterates through all the reconstructions and picks only the best, discarding all the others computed.

Inside the main loop of `ttDilepKinFit` is possible to identify three distinct stages of the reconstruction. The first is the variation of the jets and leptons momentums, as explained before. The second stage is the kinematical reconstruction of the $t\bar{t}$ system, using the varied combinations. It attempts to reconstruct the $t\bar{t}$ system, presented in section 1.1 and produces a result (the Top Quarks), which has an computed probability associated to its accuracy. This probability determines the quality of the reconstruction. The third stage is the reconstruction of the Higgs boson, using the results of the kinematical reconstruction. This reconstruction also has a probability associated, and the final quality of the overall reconstruction of the event is given by the multiplication of this probability with the one of the kinematical reconstruction of the $t\bar{t}$ system. Figure 3.1 (left) presents the explained workflow of `ttDilepKinFit`.

Note that there is data dependencies between loop iterations, since that for chosing the next jet/lepton combination it is necessary to know which combinations were already picked, and between the three stages within the loop. The kinematical reconstruction needs the combination already varied, and the Higgs boson reconstruction needs the result from the kinematical reconstruction in order to chose different particles (since one particle cannot belong to the $t\bar{t}$ system and the Higgs boson decay). The current workflow is not suitable for parallelization.

A generalist workflow suited for parallelization is presented in figure 3.1 (right). The combinations must be computed, and also all the variations, and their information stored in a data structure, so that the main loop of `ttDilepKinFit` is eliminated. Now, each stage of the workflow can have its own loop, which is not represented in the figure since it is considered to be part of the stage itself. By having their own loop, the stages can now be executed in parallel, but maintaining the same data dependencies between stages.

Within the same event, different combinations (or even variations) may be reconstructable, both the $t\bar{t}$ system and the Higgs boson, while others may not. If the kinematical reconstruction produces no results the Higgs boson reconstruction will not be performed and the next combination or variation will be

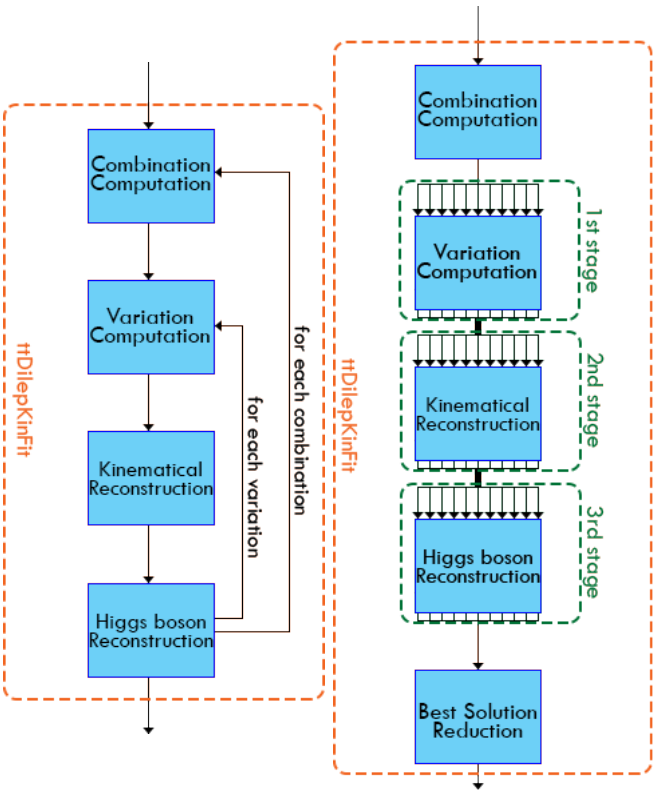


Figure 3.1.: Schematic representation of the `ttDilepKinFit` sequential (left) and the new parallel (right) workflows.

processed. If the reconstructions are parallelized, some tasks may take more time to process than others due to this irregularity of the reconstructions, which will cause load balancing issues. If the heuristic used for load balancing is not suitable for the type of the tasks (regular vs irregular) the computational resources will be underused, limiting the possible performance increase of the parallelization.

3.1. Shared Memory Parallelization

The parallel workflow model proposed for this implementation is similar to the generalist previously presented. One challenge to the implementation is that most of the variables are global to the application, and there is no data structure holding the information needed during `ttDilepKinFit`, such as the combinations. The computation of the jet/lepton combinations must be kept sequential, since choosing one combination is always dependent on all the previous combinations made and, therefore, cannot be parallelized. The final iteration through all the results, in which the best solution is chose and is "uploaded" to global variables, cannot also be parallelized in the current implementation (there is, however, a way to overcome part of this problem that will be explained later). During this chapter a concurrent task is considered to be the subset of a parallel region. The aggregation of all these tasks is the whole parallel region. Figure 3.2 presents the workflow used for this implementation, and is explained next.

In this implementation it does not make any sense to use different tasks for different stages that can be parallel; two of the three steps presented next that can be parallel will be aggregated in the same task, increasing its granularity and reducing the number of synchronizations necessary between tasks. The first step of the new workflow is to create a data structure holding all the combinations and other information associated with them. Each task will pick the respective combination and perform a loop over the subset of the total number of variations. The grain size of the parallel work of each task will be dependent on the total number of combinations times the number of variations per combination (the higher this value

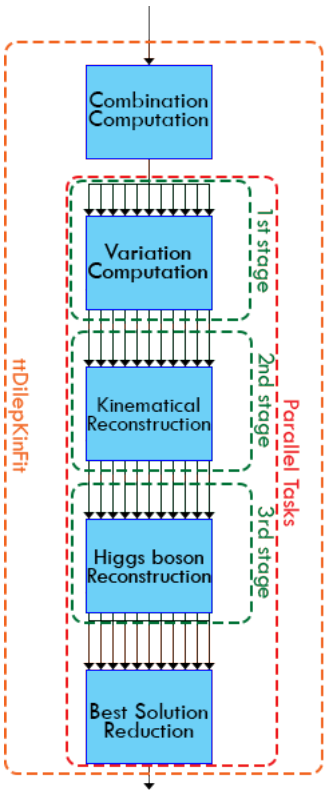


Figure 3.2.: Schematic representation of the parallel `ttDilepKinFit` workflow for the shared memory implementation.

the coarser the grains size) and the number of parallel tasks (the higher the number of tasks the thinner the grain size).

The second step, the kinematical reconstruction of the $t\bar{t}$ system, will be performed within the same task, meaning that there is no synchronization between this and the previous steps among different tasks. The same happens between this and the third step, the Higgs boson reconstruction. By aggregating these three steps the grain size of the tasks is increased, compared to the generalist parallelization model presented before, which benefits their execution on CPU (a lesser number of coarser tasks means that the CPU spends more time performing computations relevant to the application and less time switching context, which is very slow).

As explained before, after these three stages, on the original workflow, there is an iteration through all the solutions and the best is chosen. Instead of saving all the solutions, after each iteration of the loop of the current workflow the solution is compared to the previous and only the best of the two is save. When all tasks finish all their respective iterations, each will have the best solution for the subset of combinations and variations that they processed. A parallel reduction must be made so that the best solution from all the tasks is found. However, another data structure must be created, since the best solution is a set of variables and TLorentzVectorWFlags (from the LipMiniAnalysis library) class instances. The final "uploading" of the best solution to global variables is only made by the task with the best solution.

This workflow will have two limitations to the performance scalling, the creation of the data structure holding the combinations and the creation of the data structure for the best solutions and respective merging. Its time increases with the number of combinations and variations and the number of parallel tasks to use.

3.2. 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 3.1, 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.

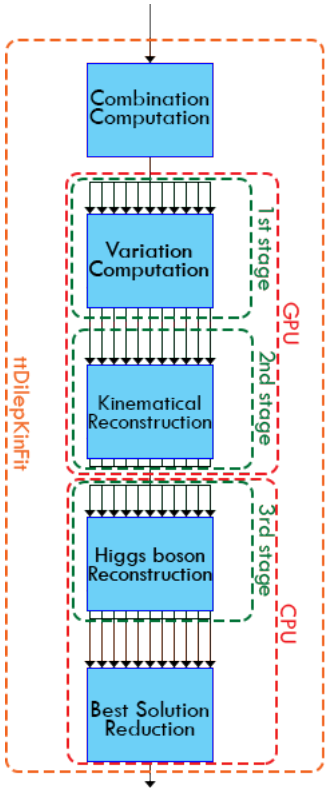


Figure 3.3.: Schematic representation of the `ttDilepKinFit` workflow.

The first two stages of the the workflow presented in figure 3.1 (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 3.3.

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. Implementation and Performance Analysis

In this chapter the implementation process, based on the models on section 3 of the different approaches will be presented and discussed. After explaining all the details of the implementation for a given platform an analysis from the computational point of view will be presented, along side with the performance comparison of the said implementations. Finally, a comparative analysis of all the implementation will be presented.

4.1. Shared Memory Implementation

The implementation of the shared memory parallelization follows the workflow presented in section 3.1. The first goal was to have a working naïve implementation that could be used as a starting point so that it could be profiled and the bottlenecks identified.

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 three dual-socket multicore systems. The first has two Intel Xeon E5-2650 (Sandy Bridhe 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. Figure Appendix A.1 illustrates the architectural model of this system. The system features 64 GB of DDR3 RAM with a speed of 1333 MHz, for a maximum bandwidth of 52.7 GB/s, measured with the STREAM benchmark **??**.

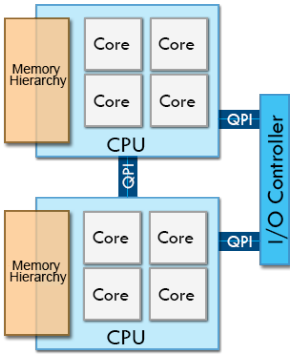


Figure Appendix A.1.: Schematic representation of a NUMA system with QPI interface.

The second system 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 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 zz GB/s. AMDuses Hyper-Transport (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 features 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. All libraries/frameworks used were compiled with compliance to the C++ 11 specifications to ensure, among other things, thread safety on memory allocations. Was used the Performance API version 5.0 for measuring the hardware counters of the different CPUs for the characterization of `ttDilepKinFit`.

CPU	Intel Xeon E5-2650	Intel Xeon X5650	AMDOpteron 6174
Architecture	Sandy Bridge	Nehalem	MagnyCours
Clock Freq.	2.0 GHz	2.66 GHz	2.2 GHz
# of Cores	8	6	12
# of Threads	16	12	12
L1 Cache	32 KB I. + 32 KB D. per Core	32 KB I. + 32 KB D. per Core	64 KB I. + 64 KB D. per Core
L2 Cache	256 KB per Core	256 KB per Core	512 KB per Core
L3 Cache	20 MB shared	12 MB shared	
CPU Interconnection	QPI @4.0 GHz	QPI @3.2 GHz	HT @3.2 GHz
ISE	AVX	SSE 4.2	SSE 4a

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 **??** 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.

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

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 **??** 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. 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 in the instruction mix presented in section 2.2.3. Instead, the computational intensity was used for measuring the CPU peak performance, as it considers all types of instructions.

Figure **??** illustrates the Roofline model for the three systems.

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, since the number of combinations varies between events but remains constant overall as the same input is used. The number of variations tested were 2^x, where $x \in \{1, ..., 10\}$.

For the shared memory implementation was used 1, 2, 4, 8, 16, 32 and 64 threads. The test using 1 thread has the purpose of evaluating the overhead of the creation and access to the data structures. The test using 64 threads is used to check if the software multithreading (managed by the operating system) has benefits, which can expose problems when accessing memory, specially to the memory bank of the other CPU, where the thread becomes stalled waiting for the data. The 8 thread test will only use one CPU, with one thread per core, running the application without the limitations of the NUMA memory accesses and the multithreading. With 16 threads both CPUs will be fully used, meaning that the memory accesses are now NUMA, but still without using hardware multithreading. The 32 threads test will use both CPUs with hardware multithreading.

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.

The tests on the Intel Xeon Phi were conducted on its two different operating modes: native and offloading. In the native mode all the application is executed on the device, as it is possible to use the ROOT and LipCbrAnalysis libraries since the device uses x86 code, even the single threaded portion of the code. Only the accesses to read the data from the hard drive pass by the CPU. In the offloading mode the Xeon Phi acts like a regular GPU, where only a portion of the application code is executed on the device.

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 task necessary for the system, which can occasionally need CPU time. The mean 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 choosen was the *k best*. It chooses 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 a different approach to ensure that the times were properly recorded and synchronization of the kernels and memory transfers are ensured. CUDA events were used for this purpose.