**Abstract**

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

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

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

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

An experimental framework, GAMA, will be used to automate (i) the workload distribution among the available resources and (ii) the transparent data management across the physical distributed memory environment between the shared multi-core memory and the many-core device memory. It will be compared against a similar concurrent framework, OpenACC, in terms of performance, development time and usability.

**Intro**

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

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

The information gathered passes through a set of computational tiers, where the data is refined and scattered to among the many research groups until it is ready to be used in simulations, where it is analyzed.

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

The focus of this dissertation work will be on tuning and parallelizing the kinematical reconstruction of events, using as case study a specific analysis, ttH\_dilep, developed by LIP, which is very important for their research and, consequently, is widely used within the group. Using a case study will lead to analyzing and improving the performance of other application specific tasks, in order to (i) get the maximum performance from the tuned kinematical reconstruction, and (ii) improve the overall performance of the application.

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

To ease this burden on the programmers there are several frameworks that try to create a level of abstraction between the architectural details of the heterogeneous environments, which affect the programing paradigm, and the programmer. The GAMA and OpenACC frameworks will be tested in the context of this problem, and the implementations using those frameworks will be compared to the fine tuned implementation, in terms of performance, usability and development time.

**Problem**

From the head-on collision between two particles result a series of decaying particles that generate other particles, in a limited chain reaction. Only some of the latter particles react with the ATLAS detector and, therefore, their information is gathered. A schematic representation of the head-on collision is presented in figure X1. ATLAS only detects the bottom quarks (which are detected as a jet of particles) and leptons, while the neutrinos do not react with the detector. In order to reconstruct the collision, the characteristics of the neutrinos must be determined. Since this system obeys a set of properties, as well as the calibrated model expected from the collision, it is possible to analytically determine the neutrinos characteristics and reconstruct the event (kinematical reconstruction), and then associate a degree of certainty.

Figure X1 – desenho ttbar

The particle collisions never happen one at a time. Instead, clouds of particles are collided, which leads to many events being recorded at the same time. This can cause bottom quark jets and leptons from a certain collision to be identified as part of other collision, and making these events unable to be properly reconstructed. Also, there are other particles that can be detected but do not belong to the ttbar system and, therefore, lead to bad kinematical reconstructions.

To overcome this problem is possible to pick all the bottom quark jets and leptons that can be associated with the collision, perform the kinematical reconstruction for each combination of two jets with two leptons, and then evaluate the probability of that reconstruction being accurate. For each event only the best reconstruction, i.e., the most probable, is chosen.

Another factor that can affect the accuracy of the reconstruction is the experimental resolution associated with the ATLAS detector. The detected values for the particles (bottom quark jets and leptons) characteristics are not 100% accurate. In fact, the measurements made by ATLAS can have a 2% inaccuracy to the real values. After the kinematical reconstruction, this error affects the probability of the reconstructed event. In order to have more accurate reconstructions, the experimental resolution must be compensated. This can be achieved by varying the bottom quark jets and leptons characteristics, such as mass or momentum, and use them in the kinematical reconstruction. However, this cannot be performed only once; the search space must be covered a certain amount of times in order to get higher probability of finding a great reconstruction. This means running the kinematical reconstruction as many times as possible.

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

In the application used as case study, the ttH\_dilep analysis, the importance of the kinematical reconstruction (dilep) is even greater. This analysis takes into account the two jets that result from the Higgs boson decay and also tries to reconstruct it. Figure X2 schematically represents the ttbar system with the Higgs boson decay and respective jets, resultant form a head-on collision. After performing the ttbar system reconstruction (tt), based on the kinematical reconstruction, and with the jets used in the best reconstruction, it uses the remaining jets to reconstruct the Higgs boson (H). If an event ttbar system is badly reconstructed, the Higgs boson reconstruction will not be accurate. Now, the final solution is given by the probability of the best kinematical reconstruction and the respective Higgs boson reconstruction.

Figure X2 – ttbar com higgs

By increasing the kinematical reconstruction performance it is possible to perform more reconstructions per event, leading to better and more accurate results. However, it is not possible to narrow the scope of the work to the reconstruction; to get the most efficiency from it, will be necessary to look to the jet combination, variance appliance and Higgs reconstruction as other important tasks to improve. The LIP research group has a big interest on improving the kinematical reconstruction, as well as the overall ttH\_dilep analysis, performance, which would give them an advantage over the other research groups.

**State of the art**

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

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

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

These devices are constituted from smaller processing units, focused on achieving the most performance possible on specific domains, opposed to common CPUs. Usually, their architecture is oriented for massive data parallelism processing, offloading the CPU from such data intensive operations. Several many-core accelerating devices are available, ranging from the general purpose GPUs, the Intel Many Integrated Core line, currently known as Intel Xeon Phi, and Digital Signal Processors. A heterogeneous system may have one or more accelerating devices of the same or different types.

**- Hardware**

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

The main similarity between different accelerating devices architectures is their orientation for the Single Instruction Multiple Data parallelism model. It is specialized to get the most performance from applying the same instruction to massive amounts of independent data. Taking for example GPUs, each pixel that must be rendered is independent from all other pixels, thus their processing is embarrassingly parallel. For achieving maximum performance, one common characteristic of the code that is to be run on these devices is this massive parallelism between the data handled. Other device specific properties, with interest for the programmer, will be discussed later.

These heterogeneous architectures open the possibility of running parallel tasks on both CPU and accelerators simultaneously. However, due to their technical differences, the same task can take different amount of time to complete. This raises the issue of managing the amount of work to be processed on the CPU and on the accelerating device, in such a way that neither of them become stalled waiting for the other to complete, and thus not wasting computational resources. It also depends on the nature of the problem; regular problems are easier to balance than irregular problems, which usually require dynamic load balancing at runtime. This is one of the most important issues to deal when tuning a parallel algorithm for hybrid systems.

**--GPUS**

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

As mentioned earlier, they are specialized for massive data parallelism, where the same operation can be applied to large amounts of data. One example of this problem domain is the multiplication of matrices, very common in scientific applications. However, as GPGPUs evolved, the support to specific demands of other than image rendering problems was added, such as support for double precision float point arithmetic and compliance to all IEEE arithmetic rules.

More recently, Nvidia launched a line of GPUs designed for scientific computation, rather than image processing. These devices, the Tesla, come with more GDDR ram, more processing units and designs suitable for use in cluster computational nodes. In this dissertation two different Nvidia GPUs will be used, the Nvidia Tesla C2070 (Fermi architecture) and the Nvidia Tesla XASD (Kepler architecture).

Nvidia GPUs are organized in a set of Streaming Multiprocessors (SM) and GDDR ram, each SM containing CUDA cores that are processing units to perform both integer and float point arithmetic (additions, multiplications and divisions). These SMs also have some specialized processing units for only square roots, sins and cosines, as well as a warp scheduler (warps will later be explained), to match CUDA threads to CUDA cores, load and store units, register files and a 2 level cache.

Nvidia considers that a parallel task is represented by a set of CUDA Threads, in which these threads will execute the same instructions (however, there is the special case of conditional jumps, which will be explained next) but on different data. A simple way to visualize this concept is by the problem of multiplying a scalar with a vector. In this case, a single thread will handle the multiplication of the scalar by an element of the vector, using as many CUDA Threads as vector elements.

A block is a set of CUDA threads set by the global scheduler to run on a specific SM. It has a limited number of CUDA threads. A grid is a set of blocks, representing the whole task to run in parallel. A warp is a set of a given number of CUDA threads (usually the same as the number of CUDA cores per SM), scheduled by the SM scheduler to run on its SM at a given time. **REVER**

When programming for these devices, conditional jumps must be avoided at all costs. Within an SM it is not possible to have 2 threads executing different instructions at the same time. So, if there is a divergence between the threads within the same warp, the two conditional branches will be executed separately, doubling the warp execution time.

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

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

**---Fermi**

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

In this model, each SM has 32 CUDA cores, with a total of 14 SM, making a total of 448 CUDA cores. Theoretically, it is possible to have 448 CUDA threads running at the same time. In each SM there is 4 Special Functional Units (SFU) to process square roots, sins and cosines.

Figure Fermi arch

Memory wise, these devices have a slightly different memory hierarchy than the CPUs, where the faster, and smaller, memory is closer to the CUDA cores. Each CUDA thread can have up to 63 registers, but when using large amounts of threads this value diminishes, which can, in some cases, cause register spilling (when there is not enough registers to hold the variables values and they must be stored in the cache).

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

The L2 cache is slower but larger, with the size of 768 KB. It is shared between the SM, opposed to the L1 cache. The global memory is the last level of on device memory. The Tesla C2070 has a total of 6 GB GDDR5 ram, with a bandwidth of 192.4 GB/s.

One important detail to extract the most performance possible from memory accesses is to use coalesced memory accesses. Since the load units get memory in blocks of 128 bits, it is possible to reduce the amount of loads by guaranteeing that all the threads that need to load, preferably continuous data on the address space (such as elements of an array), do it at the same time.

Finally, on the Fermi architecture it is only possible to run one kernel (piece of CUDA code designed to be ran by each CUDA thread) at a time on the GPU.

**---Kepler**

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

The Streaming Multiprocessor present in the Fermi architecture was changed to hold more (now 192), but smaller, CUDA cores, working at half the speed of the previous CUDA cores, and it is now known as SMX. This allows having up to 2880 CUDA cores in only one chip.

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

Figure Kepler architecture

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

**--MICS**

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

Figure MIC arch

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

Unlike the CPUs, the MIC cores do not share any cache, therefore cache consistency and coherence is not assured. If needed, data must be explicitly passed between cores.

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

**- Software**

Developing applications for a parallel environment is not as recent as one may think. There are some libraries that attempt to abstract the programmer from specific architectural and implementation details, providing an easy API, as similar as possible to current sequential programming paradigms.

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

Through the next subsections, frameworks that attempt to ease the programmer’s job, while providing scalable and flexible solutions, will be presented, that will be used during the dissertation will be presented.

**-- OpenMP**

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

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

**-- OpenACC**

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

While it was originally designed only for CPU/GPU systems, they are currently working on the support for the new Intel Xeon Phi [100]. Also, they are working alongside with the members of OpenMP to create a new specification supporting accelerating devices in future OpenMP releases [101].

**-- GAMA**

**-- Debugging**

Debug applications in shared memory systems is a complex task, where the error is sometimes hard to replicate. 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, 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 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. 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 (consider that CUDA parallel regions can reach to the thousands of threads).

**Application**

The computing resources related to the ATLAS project are organized in a series of tiers. The first is the CERN Tier-0 computing clusters and from there is distributed to the 10 Tier-1 data centers, spread by different countries, which are used for central processing and reconstruction of data events and simulation of Monte Carlo events. Tier-2 sites are dedicated to further processing and reconstruction of data and Monte Carlo events, while Tier-3 sites are used to perform data analysis and simulation.

It is in this Tier-3 that the ttH\_dilep is an analysis application fits. It was developed by the LIP researchers to solve the problem explained in the section 2. The application depends mainly on two frameworks: the ROOT and the LipMiniAnalysis.

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

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

Figure Callgraph

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

The dilep function analytically determines the neutrinos characteristics for each jets and leptons combination, reconstructing the ttbar system. It can produce two to four possible result particles. These results are processed in the remaining of the ttDilepKinFit to determine their probability, as well as reconstruct the Higgs bosons. The final probability of the reconstruction is determined by combining the probability of the ttbar reconstruction combined with the probability of the Higgs reconstruction.

As seen from the callgraph, most of the application execution time is spent in the Loop method, so there is the place where most efforts of optimization must be focused. The rest are auxiliary and I/O functions.

**Strategy**

The workflow of the Loop method, explained in the previous section, needs to be changed so that it can be easier and more effective to parallelize its execution. Currently, for each jet/lepton combination it applies the variation, executes dilep, iterates through the results and reconstructs the Higgs boson.

The best approach is to create a data set with all the jet/lepton combinations and all the respective amount of variations. Then, execute dilep with all the elements on the data set and store all the results with the associated element of the data set. Finally, iterate through all of the results, reconstruct each Higgs boson and calculate the respective probability for each element of the first data set.

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

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

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

Finally, an implementation using the OpenACC and GAMA frameworks will be tested, relatively to the previous optimized implementations, in terms of performance but also considering the development time and usability of these tools.

Refs

3 - http://public.web.cern.ch/public/en/research/DataAnalysis-en.html

100 - <http://www.hpcwire.com/hpcwire/2012-06-20/openacc_group_reports_expanding_support_for_accelerator_programming_standard.html>

101 - http://www.openacc-standard.org/node/49