Analysis Acceleration on GPUs for the ATLAS Experiment at CERN

Maria Rovatsou

Thursday 25th March 2010

Abstract

The ATLAS detector is one of the two biggest detectors of the Large Hadron Collider (LHC) at the European Center of Nuclear Physics (CERN) dedicated for research of the origin of mass, extra dimensions, and dark matter. The detector at design performance works at luminosity $10^{34} cm^{-2} s^{-2}$ with beam bunch rate of 40MHz. The rate of the collisions taking place in the accelerator is the input rate of the software that stores and processes them, the ATLAS Trigger. The Trigger is divided into three levels, the first level is a custom electronics trigger and the other two levels compose the High-Level trigger that is responsible for combining the data from the detector and reconstructing the tracks of the particles in order to store interesting data. The track reconstruction is achieved with the utilization of the Kalman filter technique. The current implementation of the filter is a distributed sequential implementation. An acceleration of the filter would boost the whole process and would allow even more significant data to be available for processing. Apart from that, the proposed upgrade of the LHC in order to increase its performance, results in luminosity increase of one order of magnitude higher and would require a more efficient implementation of the Kalman filter. A possible way for acceleration of the filter is by porting it on a Graphics Processing Unit (GPU), which has immense computational power that can be exploited through programming models as CUDA. The proposed approach is a parallelization on thread level of the Kalman filter for execution on an NVIDIA GPU. The proposed programming model is CUDA and the analysis of its performance will be done with the CUDA profiling tools and with benchmarking tools.

The ATLAS experiment

ATLAS (A Toroidal LHC Apparatus) [1], [2] is one of the four multi-purpose detectors of the Large Hadron Collider (LHC) at the European Center of Nuclear Research (CERN) . The ATLAS experiment pursues to answer the missing parts of the Standard Model. These sum up to the origins of mass and dark matter, the existence of more than four dimensions and whether all the fundamental forces can be unified. In order for these questions to be answered the ATLAS experiment has to play its part storing and processing a massive amount of data from the detector while the LHC produces approximately 15 Petabytes of data annually [3].

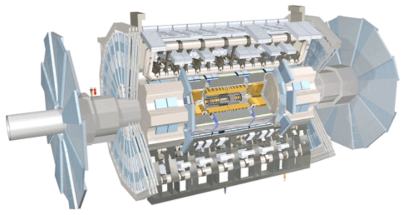


Figure 1: A 3D view of the ATLAS detector, where at the center the Inner Detector is colored yellow

The ATLAS on-line software of the experiment aims to collect information on the identity, the energy and the path of particles passing through the detector resulting from the collision of two high energy protons. The proton-proton collisions happen in a center of mass energy of 14 TeV with design luminosity of $10^{34} cm^{-2} s^{-1}$ where the beam bunch crossing rate of the LHC is 40 MHz. However, from the events taking place in this rate, the interesting physics events are several magnitudes smaller than the physics pile-up events -which are events uninteresting as far as the Standard Model is concerned [4], [5].

The LHC has started accelerating protons with highest luminosity is 6.8 10^{26} $cm{-2}$ $s{-1}$ [1] and in the following years there is a proposed upgrade of the LHC where luminosity will be one order of magnitude higher. In that case from 25 proton-proton collisions per beam bunch crossing it will reach up to 400 proton-proton collisions per beam bunch crossing. This upgrade will cause significant computational challenges [6].

ATLAS Trigger

The Atlas Trigger [7], [8], [9], [5] is the software responsible for selecting and storing the physics events that take place in the ATLAS detector. It is subdivided in three level triggers, Level 1, Level 2 and Event Builder. The Level 1 trigger is a custom-electronics pipelined hardware trigger that uses low granularity data from the calorimeters and especially from the muon trigger chambers. Its input rate is the beam bunch rate, 40 MHz, which causes a latency of 2.5 μs and its output rate is 75MHz, upgradeable to 100MHz. The Level 1 trigger splits the events space into Regions of Interest (RoIs), which are regions of the detector where there is greater possibility for interesting physics events. There

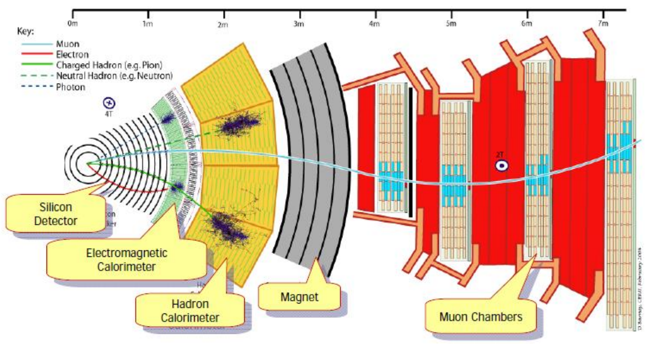


Figure 2: Overview of the ATLAS detector chambers

are about 2 RoIs per accepted event and the average size of a RoI is approximately 2% of the total event. When Level 1 accepts an event it stores the data and the readout into Readout Buffers (ROBs) for access by the Level 2 trigger.

Level 2 trigger and Event Builder form the High-Level Trigger. The input rate of Level 2 is the output rate of the Level 1 trigger and the maximum output rate is 3KHz, where the processing time per event on average is 10ms. The Level 2 trigger is the first stage of the Trigger that has access to the Inner Detector, where the collisions take place. As a result of this access, Level 2 can combine data from different detector chambers. It can combine the Inner Detectors data with the calorimeter or the muon system and improve the particle reconstruction. After Level 2 accepts an event, Event Builder takes these data and reconstructs the event in full. This reconstruction is vital for the identification of physics signatures which can exhibit new or rare physics events, such as high transverse energy electrons, photon and muons or final states containing b-quarks.

Kalman Filter

In order for physics the events to be reconstructed, the High-Level trigger utilizes the Kalman Filter or the extended Kalman Filter depending on the deployed algorithm. It is used for the selection of track candidates from select parts of the detector, as some particles that have been accepted from the first level trigger are from pile-up events [7], [8], [9], [10]. The fast reconstruction of these tracks in the Track Fitter relies on the performance of the Kalman filter.

The standard Kalman Filter [11], [12], [13], [14] is a set of mathematical tools that address the problem of prediction and filtering of an unknown process. The filter forms an initial estimation of this process and utilizes noisy measurements in order to refine its estimation. Prediction is the projection of the current estimation to the next time step and filtering is the incorporation of the measurements for the formulation of the final estimation of this time step. The Kalman filter approximates the unknown process optimally in respect to the estimation error. The filter is a closed loop, where prediction and estimation follow one another, with the significant advantage that in each time step it does not need to store or process all the prior measurements and estimations

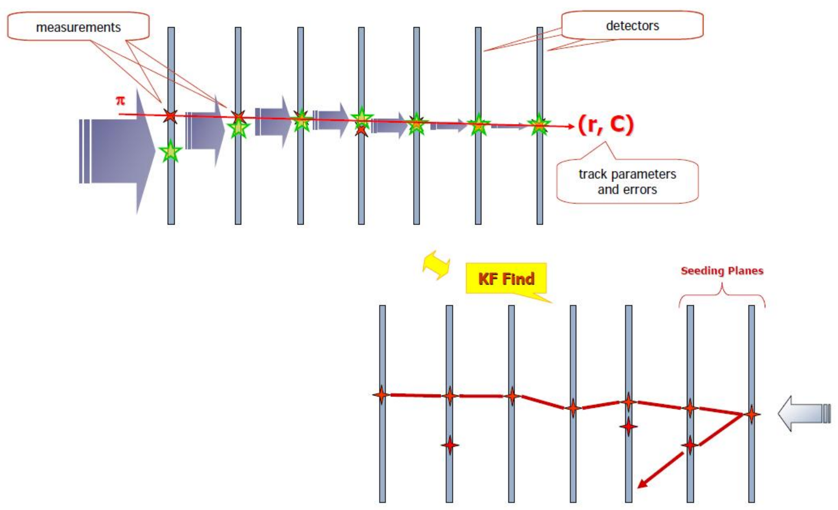
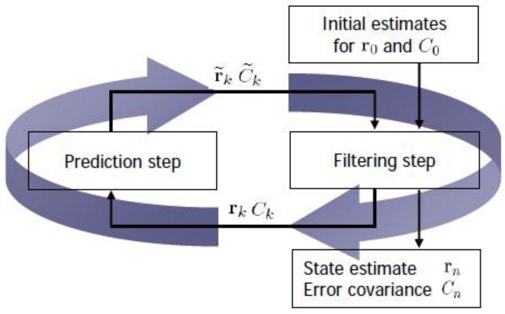


Figure 3: Particle track reconstruction utilizing the Kalman Filter, where the position of the particles when they are passing through the different chambers of the detector leads to the reconstruction of their whole track

of the process as all the information it needs is included to the estimation of the previous time step. The filtering process can be summarized in five steps, i) initial estimation of the process, ii) prediction of the estimation of the process for the next time step , iii) incorporation of noise, iv) the filtering step where the estimation of the process is updated with the measurements and v) the estimation of the process. The extended Kalman Filter (EKF) [12], [13] is



Initial estimates for $\mathbf{r}_0$ and C_0
$\tilde{\mathbf{r}}_k$ $\tilde{C}_k$
Prediction step
Filtering step
$\mathbf{r}_k$ C_k
State estimate $\mathbf{r}_n$
Error covariance C_n

Figure 4: Overview of the process of the Kalman Filter, where r_0 and C_0 are the initial values of the unknown process r and its covariance matrix, $\tilde{\mathrm{r}}_k$,$\tilde{\mathrm{C}}_k$ are the predicted estimates of the process per time step k, r_k, C_k are the estimates produced from the filtering at time step k, and r_n, C_n are the final estimates for the whole process

a variation of the Kalman filter for a non-linear process. The difference is that the unknown process estimated is modeled as a non-linear process and the measurements are related to the process by a non-linear function. This variance of the Kalman filter is approximately optimal and is used for better approximation and computational optimization.

Performance Optimization

The High Level Trigger in view of the upgrade of the LHC experiment poses new computational challenges to track fitting by the Kalman filter. One alternative for addressing the challenges of the upgrade of the LHC is the use of General Purpose Graphic Processing Units (GPGPUs). Due to their development origins GPUs are focused toward massive high parallel computations on data rather than data caching and flow control, shown in figure , so they have more capacity of computations in relation to the CPUs.

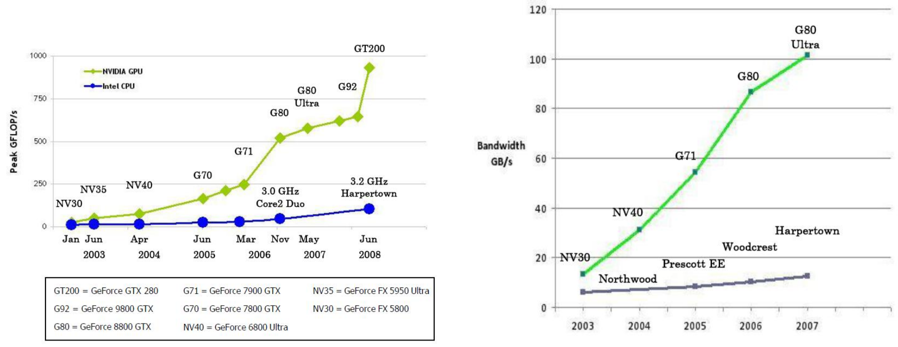


Figure 5: Comparison of the CPU and GPU floating-point operation capabilities and memory bandwidth

One of the applications that could be accelerated by the use of GPUs is the Kalman filter routine of the High-Level Trigger. However, the utilization of GPUs is not straightforward as the applications cannot be simply ported from a CPU to a GPU due to the fact that GPUs architecture does not support high memory caching, and the computations must run in parallel and independently. It is a possibility though that should be explored as in the case of particle track reconstruction, with the thousands of physics events taking place, a process speedup would be translated into an increase of the output rate of the trigger [4].

Graphic Processing Units (GPUs) [15] originate from the need of specialized processing units for graphics processing and their most common use is in gaming industry, e.g. Playstation 3, and graphics cards for personal computers. GPUs are focused on the optimization of floating-point computations throughput, so their architecture imposes some restrictions on the programs that are suitable for executing on them. Firstly, the computations on the data have to be able to be parallelized in the form of executing the same operations on a large amount of data. Apart from that, the data must have high arithmetic density so that the flow of the program needs a small number of memory accesses relatively to the computations of the data transfered, in order for the high memory latency to be hidden by the computational load. Lastly, the problem must have data locality, where data are needed only for current computations in the execution of the algorithm. In order to provide more specific information about the hardware that is proposed to be used in this project we are going to focus on NVIDIA GPUs.

NVIDIA GPUs [15] consist of 16 multiprocessors with eight scalar processors. In a top-down view, as shown in figure 6 [15], threads are first divided into blocks per multiprocessor and then into warps, blocks of 32 threads, for execution on the scalar processors.

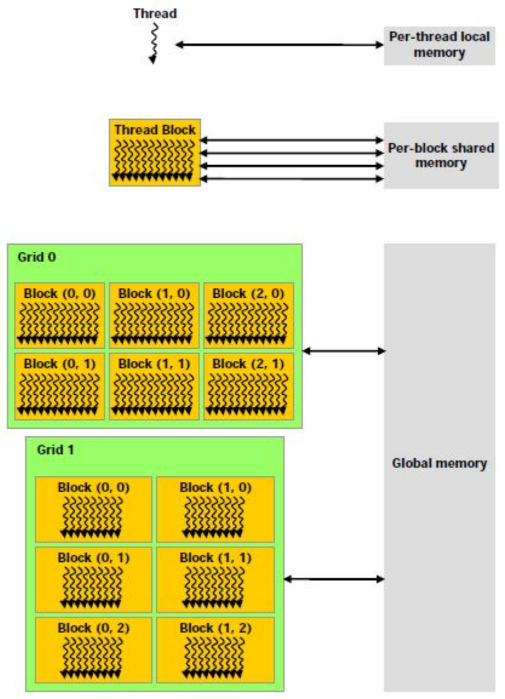


Figure 6: GPU Memory hierarchy

The are several types of memory for the GPU, global, texture, local, constant, shared memory, and register file. The global memory is external to the GPU and it serves as main memory. The problem that arises with memory accesses is due to the fact that this memory is not cached so it has the latency of the DRAM memory which is from 400 to 600 clock cycles. The local memory is a part of the global memory. Two cached memories are the read-only constant memory and the texture memory.The last two memory types, shared memory and register file are embedded to the multiprocessors. Shared memory is the local address space of the blocks of threads of the multiprocessors. The register file is dynamically allocated by the threads of the multiprocessor. The overall hardware model of a GPU is shown in figure 7 [15].

NVIDIA has introduced the Compute Unified Device Architecture (CUDA) programming model for scalable GPU programming [15], [16]. CUDA is designed in order to exploit the computational power of NVIDIA GPUs as CPUs co-processors and its software architecture structure follows the hardware structure of the GPU. CPUs are the *hosts* that are responsible for the flow control and GPUs function as compute *devices*. The parallel program is partitioned

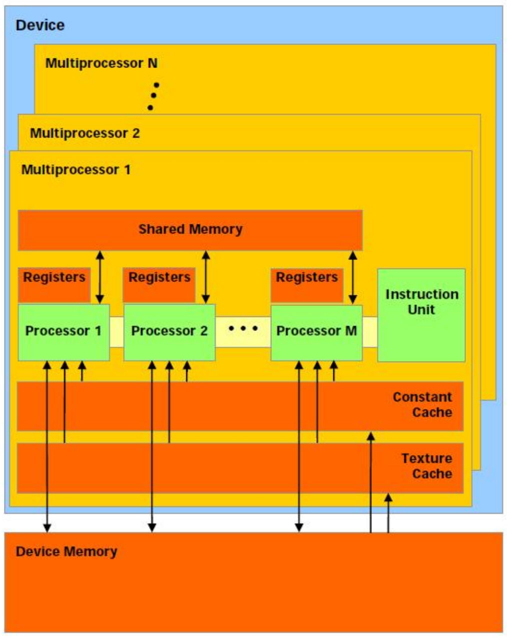


Figure 7: GPU Hardware Model

to sub-tasks that execute in parallel and are referred to as *kernels* to which correspond large blocks of data. Kernels execute in parallel on grids and have access to the grids register file, to the global memory and to shared memory. They cannot read or write on the global memory as it is written only by the host. The shared memory of grids is divided in blocks where multiple threads are executing in each block. Threads that belong to the same block execute the same operation on their set of data, can share data through their shared memory, and they are synchronized by a barrier. These threads cannot exchange data with other threads belonging to different blocks. The number of blocks that can be executed on a multiprocessor depends on the resources needed for their execution, i.e. the number of registers needed in the register file and the shared memory that is partitioned for each block. This structure adopts the Single Instruction Multiple Threads (SIMT) architecture, which executes the same operation to different blocks of data exploiting the data locality of the problem. The outline of how a program is executed is shown in figure 8.

The advantage of such a structure is that the programming model is scalable, meaning that only on runtime the number of cores on which is going to be executed is essential, but this comes at the cost of utilizing such a model only on when the problem has the property of data locality, as data exchange is constrained at the level of thread blocks through their shared memory.

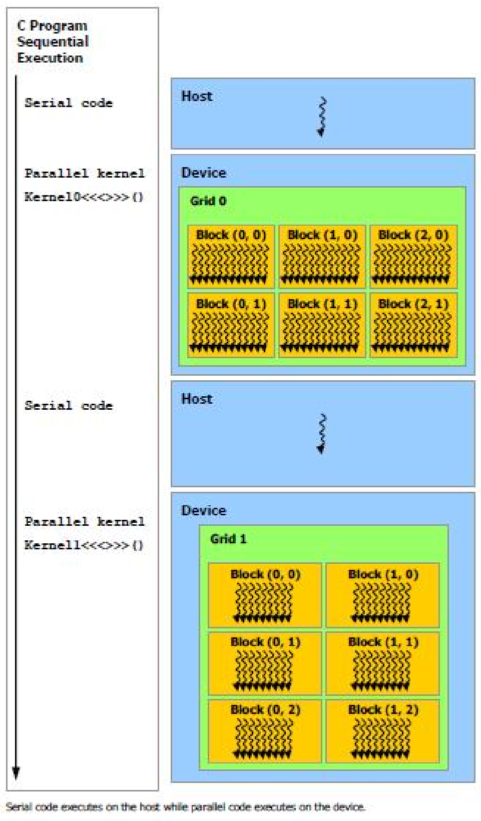


Figure 8: Control flow of parallelized program execution on CPU as the Host and GPU as the devise

Related Work

S. Gorbunov et al., [14] in 2007, ported the Kalman filter of the CBM experiment to Single Instruction Multiple Data (SIMD) architecture to SSE and Cell SPE with speedup of 10000. CBM [17] is a heavy-ion experiment at the Facility of Antiproton and Ion Research (FAIR) in Damstardt, Germany. Two optimizations lead to this speedup, one was memory optimization specialized for the CBM experiment and the other was the numerical stability of the square root Kalman filter. As in this project we are not dealing with the CBM experiment we are going to focus on the second optimization. In this research, they have utilized single precision data and achieved the same levels of accuracy with the standard Kalman filter and numerical stability by rejecting any initial estimation errors that are four times larger than the mean measurement errors. Although, in the first estimation there is numerical instability due to this rejection the instability is not propagated to later estimations and the outcome is numerically stable. This optimization boosts performance as it uses single precision data which are processed at a greater speed than double precision data.

M. Bach et al., [18], [19] in 2008, ported the Kalman filter of the CBM experiment to NVIDIA's GPU GT200 using the optimizations referenced in the SIMDized version of the algorithm [14]. In this implementation the difficulties that were addressed was that the sequential Kalman Filter should be parallelized at thread level adopting the SIMT architecture needed for porting it to the GPU. This had the implication that the code does not branch to different threads lead by the different data but from the code of the threads. Apart from this, it used the data primitive type of *float* instead of a vector type. This implementation also profited by the memory optimizations done in the SIMDized version of this Kalman filter. The result was that for the same latency at 44μs it can calculate 960 tracks instead of 4 tracks in parallel, which shows how well the track fitting algorithm can exploit the GPUs computational power.

Porting Kalman Filter of the High-Level Trigger to GPU

In this project the approach that we are proposing is to port the Kalman filter of the High-Level Trigger of the ATLAS experiment to an NVIDIA GPU using CUDA Programming Model. The need for accelerating the Kalman Filter springs from the high input rate of data into the High-Level trigger that is 75KHz leaving for the Event Builder a latency of approximately 1ms for track fitting. In this input rate any acceleration of the algorithm will have significant benefits for the track reconstruction. Even if the output rate of the filter remains the same there may be an increase of the number of reconstructed tracks. This hypothesis is based on the related work done for the CBM experiment [18]. The context of the Kalman filter, however, is different as the ATLAS experiment deals with different input data and higher input rate in comparison to the CBM experiment, which may lead to different performance results. The GPUs, though, are candidates for better performance of the algorithm which justifies the exploration of such a solution.

The CUDA Programming Model is chosen as it is the programming model designed for NVIDIA GPUs and it can be used with only some extensions for C++, which is the programming language of the implementation of the sequential algorithm being used. Apart from that, CUDA has the advantage of supporting scalable implementations of a program on an NVIDIA GPU. A compiled version of the program can run in any GPU by providing on runtime the number of cores the underlying GPU has.

The GPU system that is going to utilized for the implementation and testing of the algorithm is Tesla C1060. It has 4GB of overall memory, 16KB of shared memory and the memory bandwidth rate is 408 GB/sec and it is clocked at 800 (GDDR3) . The single precision performance is 933 Gigaflops and 78 Gigaflops of double precision performance [20], [21].

The algorithm has to follow the SIMT parallelized model, so parallelization must be done on thread level. This can be achieved by distributing the computations of the Kalman filter to blocks of threads. The Kalman filter's steps have to be divided to tasks that are going to be divided into subtasks of simple operations for the threads to execute on small proportions of the input data. A problem that has to be addressed is how these computations are going to be divided in blocks so that these threads are the only ones that are going to share data and be synchronized in an optimal way. Moreover, the data locality of the specified problem has to be explored given the data that are going to be processed. The data distribution into blocks is crucial as the performance of the algorithm depends on it. The global memory of the GPU may be very large but the latency is also very high, at approximately 400 to 600 cycles per access, so the data that are going to be transfered to the 16KB shared memory of the threads must be optimally distributed in respect to data locality. The exploitation of these data so that the full power of computational units is utilized is the key matter in order for the high latency of the transfer form global memory to shared memory to be balanced.

Furthermore, apart from the standard Kalman filter there are some variation that also have to parallelized and tested in order to explore the potential of better performance. One such version, is the square root implementation of the Kalman filter used in the SIMDized and SIMTed version of the Kalman filter for the CBM experiment [14], [19]. The performance of the GPU for single precision data is orders of magnitude higher, more specifically instead of using the double precision units that have the performance of 346 Gigaflops we can use the single precision units with performance 4.17 Terraflops. This increase of computational power may increase the performance of the parallel algorithm under the condition that the latency for transferring data from global to shared memory is hidden by the amount of computational workload executed.

A standalone version of the algorithm will suffice for performance tests and profiling. Optionally, if the parallelization has been achieved in way that provides good performance results it can be incorporated to the framework so that testing and profiling can be done in order for its performance to be measured in relation to the whole framework.

The implementation and profiling will be done with the aid of the tool provided by CUDA for Linux platform. The debugger DDD is going to be used as a debugging environment. The profiling will be done utilizing the CUDA Visual profiler which provides Kernel Profiler Data, such as GPU and CPU Time for the various methods of the program, Memory transfer information, such as what type of transfers take place, synchronous or asynchronous, source and destination, size and which stream is used, and Data Analysis information and plots where data from multiple sessions are compared and where counters and time-lines are shown.

Benchmarking tools will also be utilized in order for the performance of the GPU for such computations to be measured. A possible benchmarking tool is GPUBench [22], [23], which is an open-source tool introduced for evaluation of GPU performance for scientific operations. This tool evaluates float-point memory bandwidth, data upload and readback, instruction rate, and numerical precision. It also produces graphs and data from all the tests it runs. Further research for packages and tools for benchmarking GPUs for scientific applications and computations relevant to the Kalman filter must be done, as most of the benchmarking tools for GPUs focus on tests for graphics processing.

Schedule

The first part of the project consists of some research and familiarization with the domain and tool sets. In the beginning there needs to be some time dedicated for familiarization with CUDA API for C++ and the debugging and profiling tools offered by NVIDIA CUDA for the Linux platform. Another important task is the understanding of the currently used implementation of the Kalman filter and the ATLAS framework. Some research of other benchmarking tools has also to be done. Then a relatively large amount of time is dedicated to the design of the parallelization of the Kalman filter, its implementation, profiling and testing. Moreover, benchmarking tools have to be used for performance analysis of the GPU in use in view of the type of computations used in the Kalman filter implementation. After the first working version of the GPU-based Kalman filter is implemented, alternatives can be tested, such as optimizations for single-precision data and memory utilization optimizations. An optional part of the project is its incorporation to the ATLAS framework, which can be done under the condition that the parallelized version of the program has achieved in standalone version some speedup and that there is enough time for such a task.

Tasks of the Project	Starting Dates
Orientation with CUDA and its extensions for C++	31/5
Familiarization with the serial implementation of the Kalman Filter in the ATLAS High Level Trigger	7/6
Evaluation of possible benchmarking tools relevant to the application	9/6
Kalman Filter parallelization design and implementation	14/6
Testing and Profiling of the first version of the parallelized Kalman filter	23/6
Benchmarking	12/7
Optimizations, Testing, and Profiling	19/7
Start writing the dissertation	26/7
Possible integration	2/8
First draft of the dissertation	9/8
Final submission of the dissertation	19/8

Bibliography

[1] "ATLAS experiment." http://atlas.ch/.

[2] "CERN-LHC Experiments: ATLAS." http://public.web.cern.ch/public/en/LHC/ATLAS-en.html.

[3] "Worldwide LHC Computing Grid." http://lcg.web.cern.ch/LCG/public/default.htm.

[4] Amir Farbin, "Emerging Computing Technologies in High Energy Physics," in *Meeting of the Division of Particles and Fields of the American Physical Society*, jul 2009.

[5] A. G.-M. et al., "Overview of the High-Level Trigger Electron and Photon Selection for the ATLAS Experiment at the LHC.," in *IEEE Transactions on Nuclear Science*, vol. 53, pp. 2839–2843, Institute of Electrical and Electronics Engineers, New York, USA, Oct. 2006.

[6] N. Konstantinidis, "ATLAS L1 trigger for super-LHC," Sept. 2009.

[7] M. R. Sutton, "Commissioning the ATLAS Inner Detector Trigger," Tech. Rep. ATL-DAQ-PROC-2009-015. ATL-COM-DAQ-2009-036, CERN, Geneva, May 2009. 15/05/2009.

[8] J. T. M. Baines, H. Drevermann, D. Emeliyanov, N. P. Konstantinidis, F. Parodi, C. Schiavi, and M. Sutton, "Fast Tracking for the ATLAS LVL2 Trigger," no. ATL-DAQ-CONF-2005-001. ATL-COM-DAQ-2004-028, p. 246, 2005.

[9] M. Sutton, "Tracking at level 2 for the ATLAS high level trigger," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 582, no. 3, pp. 761–765, 2007. VERTEX 2006 - Proceedings of the 15th International Workshop on Vertex Detectors.

[10] I. Kisel, "Online Event Reconstruction in HEP Experiments." FIAS Colloquium, Dec. 2009.

[11] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[12] G. Welch and G. Bishop, "An Introduction to the Kalman Filter," in *SIGGRAPH 2001*, ACM, Aug. 2001.

[13] P. S. Maybeck, *Stochastic Models, Estimation, and Control*, vol. 1. Academic Press, 1979.

[14] S. Gorbunov, U. Kebschull, I. Kisel, V. Lindenstruth, and W. F. J. Müller, "Fast SIMDized Kalman filter based track fit," *Computer Physics Communications*, vol. 178, no. 5, pp. 374–383, 2008.

[15] NVIDIA, *NVIDIA CUDA Programming Guide*, Aug. 2009. version 2.3.1.

[16] "CUDA Technical Training," 2008.

[17] "CBM Collaboration, Compressed Baryonic Matter Experiment," 2006. Technical Status Report. GSI.

[18] M. Bach, S. Gorbunov, I. Kisel, V. Lindenstruth, and U. Kebschull, "Porting a Kalman filter based track fit to NVIDIA CUDA." GSI Scientific Report 2008.

[19] M. Bach, "Utilization of Graphics Processing Units in Applications for High Energy Physics," Master's thesis, Faculty of Physics and Astronomy, 2009.

[20] "GPGPU Environment on Eddie." https://www.wiki.ed.ac.uk/display/ecdfwiki/GPU+and+CUDA+Quick+Start.

[21] Gernot Ziegler, "Tesla GPU Computing, A Revolution in High Performance Computing." www.industrialmath.net/CUDA09_talks/ziegler.pdf.

[22] "GPUBench." ACM Workshop on General Purpose Computing on Graphics Processor.

[23] "GPUBench: Evaluating GPU Performance for Numerical and Scientific Applications,"