Acceleration Analysis on GPUs for the ATLAS experiment at CERN

SIMT design of the High Level Trigger Kalman Fitter

Maria Rovatsou

Master of Science
School of Informatics
University of Edinburgh

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 into the origin of mass, extra dimensions, and dark matter. The LHC has design performance that can achieve proton-proton collisions at luminosity $10^{34} cm^{-2} s^{-2}$ with beam bunch rate of 40 MHz. The rate of the collisions taking place in the accelerator is the input rate of the hardware and software system 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, which 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 sequential implementation. An acceleration of the filter would boost the whole process and would allow even more significant data to be available for processing. A possible way of acceleration is porting it to a Graphics Processing Unit (GPU). The proposed approach is a parallelization of the Kalman filter for execution on an NVIDIA GPU utilizing CUDA. The parallelization of the Kalman Fitter suggested is to have a candidate track reconstruction per GPU thread. This approach aims at increasing the throughput of the overall fitting process rather than accelerating the fitting process of a single candidate track. The implementation of the Kalman Fitter is not straightforward, as many complications have arose, first of all the existing C++ implementation cannot be integrated with CUDA at the time being, as it needs full C++ support. The new implementation of the Kalman Fitter in C has made even clearer that the complexity of the code dictates re-engineering in view of the CUDA integration and parallelization. The complexity of the original code did not lead to final implementation of the Kalman Fitter with CUDA, although suggestions for overcoming them have been implemented to a new C source code.
Acknowledgements

I would like to express my appreciation to my supervisor for his aid, and support during this dissertation. His guidance and understanding helped me cope with the difficulties that arose in this project. I would also like to thank Efi Mouzeli and Andrew Manousakas for offering me their precious help and encouragement throughout the year. More than that, all my Greek, Romanian, Maltese, Spanish friends have drawn a colorful web of memories to an otherwise colorless year. Last but not least, I would like to thank my parents that have done everything in their powers to support me in everything that I decided to pursue and always surrounded me with care even though I have been miles away.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Maria Rovatsou)
To everyone that supported me in this endeavour.
# Table of Contents

1 Introduction ............................................................ 1
   1.1 Thesis Overview .................................................. 2

2 Background ............................................................ 4
   2.1 The ATLAS experiment ........................................... 4
   2.2 The ATLAS Trigger ............................................... 6
   2.3 The Kalman Filter ............................................... 8
      2.3.1 The Standard and the Extended Kalman Filter ........... 8
      2.3.2 The Kalman Filter for Track Fitting .................... 11
   2.4 Acceleration Utilizing Graphics Processor Units ............ 14
      2.4.1 Graphics Processing Units (GPUs) ..................... 14
      2.4.2 Compute Unified Device Architecture (CUDA) .......... 18
      2.4.3 CUDA C++ .............................................. 21

3 Track Fitting in the ATLAS detector ............................... 22
   3.1 Track Fitter Algorithm ....................................... 23

4 Our Approach on Acceleration of the Kalman Fitter ............. 30
   4.1 SIMT design approach of the Kalman Filter .................. 30
   4.2 Related Work .................................................. 31

5 C++ SIMT Design ................................................... 33
   5.1 Constraints of integrating the C++ Kalman Fitter with CUDA . 33
   5.2 Libraries for integration of CUDA with C++ .................. 35

6 Implementation of the Kalman Fitter in C ......................... 37
   6.1 SIMT design of the C version of the Kalman Fitter ........... 43
List of Figures

2.1 Overview of the ATLAS Inner Detector ........................................ 5
2.2 The LHC ring ............................................................................. 6
2.3 Particle track reconstruction utilizing the Kalman Filter. ............. 8
2.4 Overview of the process of the Kalman Filter ............................. 10
2.5 The track fitting process ............................................................. 12
2.6 Comparison of CPUs and GPUs ................................................. 14
2.7 “Fermi” GPU architecture .......................................................... 15
2.8 The design block diagram of a Streaming Multiprocessor (SM) and of a Core................................................................. 16
2.9 GPU Memory hierarchy ............................................................... 17
2.10 Unified Address Space supported by PTX 2.0 ......................... 18
2.11 The grid of thread blocks ........................................................... 19
2.12 Control flow of parallelized program execution on CPU as the Host and GPU as the device ......................................................... 20
3.1 Overview of the Level 2 Trigger Algorithms ............................... 22
3.2 UML diagram of the class hierarchy of SiCluster ....................... 25
3.5 Extrapolation process ................................................................. 26
3.3 The filtering nodes inheritance scheme ....................................... 28
3.4 UML class diagram of the classes providing all the information for the fitting process .......................................................... 29
## List of Tables

2.1 The system of equations of the combined Kalman Filter - Smoother algorithm. . . 13

6.1 Results of the same operations computed in C and C++ programs . . . 38
6.2 Modifications of the structs for less memory consumption I . . . . . . 41
6.3 Modifications of the structs for less memory consumption II . . . . . . 42
6.4 Modifications of the structs for less memory consumption III . . . . . . 43
6.5 The arrays of data structures to be transferred at the GPU device memory 45
Listings

6.1 Allocation and memory transfers from and to the device of 2D pitch in CUDA for the array dkfTrackArray ........................................ 45

6.2 Memory allocation and transfers from host to device and vice versa of a 3D array allocated as pitch of primitive type double and of struct TrkPlanarSurface .......................................................... 46
Chapter 1

Introduction

ATLAS (A Toroidal LHC Apparatus) is one of the largest multi-purpose detectors of the Large Hadron Collider (LHC) based at the European Center of Nuclear Physics (CERN) in Geneva, Switzerland. The ATLAS experiment is concerned with the origins of mass, the existence of dark matter and extra dimensions. The LHC is accelerates protons with maximum rate of 40 MHz. The product of collisions is a large number of events, where only a number of them are interesting physics events. The rest of the events are considered noise and have to be filtered away from the interesting events. The filtering and the first stage of reconstruction of events occurring in the ATLAS detector are executed by the ATLAS Trigger software. The Trigger processes the data from the detector with the rates of occurring events, but a possible upgrade of the LHC accelerator will pose even greater challenges.

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, which is responsible for combining the data from the detector and reconstructing the physics interesting events. The events’ track reconstruction is achieved with the utilization of the Kalman filter technique. The current implementation of the filter is a sequential implementation run on CPU farms. An acceleration of the filter would boost the whole process and would allow a greater number of significant data to be available for processing.

A possible way for acceleration of the filter is by utilizing programmable Graphics Processing Units (GPUs). Their immense computational power supersedes other alternatives, such as CPUs or superscalar processors. The proposed approach is a parallelization following the Single Instruction Multiple Threads (SIMT) paradigm that GPU
programming imposes for programmable GPU devices. The programming model proposed is CUDA as the programmable GPUs used are NVIDIA GPU devices for which the CUDA programming model is widely used especially for similar scientific tasks. The specific parallelization approach for the Kalman Fitter is to have a candidate track fit per GPU thread. This approach aims at increasing the throughput of the overall fitting process rather than accelerating the fitting process of a single candidate track. The latter suffers from strong dependencies amongst intermediate computations which hinder parallelization.

1.1 Thesis Overview

This thesis is concerned with the optimization of the candidate track reconstruction of events detected by the ATLAS inner detector. The scope of this project is not to gain deep understanding of the underlying physics theory or present it, but the attempt to optimize Kalman Fitter’s performance by porting it to a GPU device. The physics phenomena are described as a context of the routine, which is a standalone version of the routine used in the ATLAS Trigger. Any further explanation of the physics terminology, equations and phenomena can be found in the technical reports of the ATLAS experiment [15], [16]. These reports also include further details of the technical characteristics of the ATLAS detector whose environment is modeled in this routine.

The rest of this document is organized as follows. The presentation of the thesis overview starts with Chapter 2, which contains a concise description of the context in which the Kalman Fitter routine works in and the background theory of the Kalman filter mathematical model. It introduces the ATLAS experiment and the overview of the ATLAS Trigger to create an understanding of the context of the algorithm, as the Kalman Fitter is one of the routines of the ATLAS software Trigger. An analysis of the Kalman filter and its optimization over non-linear processes, the extended Kalman filter, follows in order to present the application of the general approach of the extended Kalman filtering to particle physics. The Kalman Filter and Smoother are utilized for track reconstruction from data gathered by accelerator’s detectors. The mathematical tools for this kind of filtering are presented in section 2.3.2. These mathematical tools, especially in a complex environment such as the Inner Detector of the LHC, contain dense computations, whose execution can be accelerated by using Graphics Processing Units (GPUs). Section 2.4 presents some notion on performance optimization by
exploiting GPUs instead of solely CPUs, and gives a brief presentation of the GPU architecture and the suggested programming model, CUDA. In the last section of this chapter, the advances of GPU architecture as far as support of C++ are discussed.

Chapter 3 contains a brief description the ATLAS level 2 software routines, to provide the reader a better view of the role of the Kalman Fitter in the High Level Trigger. In this chapter, the Kalman Fitter algorithm and implementation are presented in a detailed manner.

Chapter 4 presents the approach proposed for the acceleration of the Kalman Fitter routine. This proposal suggests the use of the GPU architecture and describes the characteristics of the SIMT parallelization of the routine.

Chapter 5 analyses the issues that arose when the C++ implementation of the routine was attempted to be ported to the device and why this integration is not currently feasible.

Chapter 6 describes the implementation of the Kalman Fitter in C and the iterative approach of re-engineering and refactoring of it to reach an implementation which can be integrated with CUDA.

In Chapter 7 some proposals for future work are suggested in the short-term, but also a view of the acceleration of the ATLAS Trigger in the long-term. To conclude with, the approach of this thesis is summarized along with the issues encountered in the span of this thesis.
Chapter 2

Background

2.1 The ATLAS experiment

The European Organization of Nuclear Research [3], CERN, is the European Institute of fundamental physics research, based in Geneva, Switzerland. The construction of LHC (Large Hadron Collider) [8] [7] was CERN’s biggest and most ambitious project. It is the particle accelerator, with a 27km circumference spanning across the Swiss-French border. The collision energy of the protons is by design 14 TeV [19]. The highest luminosity\(^1\) achieved until the time of the writting was about \(8 \times 10^{29} \text{cm}^{-2} \text{s}^{-1}\), reported on the [4] and it is double the luminosity achieved since the beginning of this series of experiments on March. The design luminosity of the accelerator is \(10 \times 10^{34} \text{cm}^{-2} \text{s}^{-1}\) and an upgrade is proposed that will augment the luminosity one order of magnitude higher. Instead of having about 25 proton-proton collisions there will be up to 400 proton-proton collisions per beam crossing [24].

The largest of the two multi-purpose detectors of the LHC is the ATLAS (A Toroidal LHC Apparatus) [1] [5]. The ATLAS experiment is dedicated to the missing blocks of the Standard Model (SM), with the most striking example being the Higgs boson [2] [6], the last unobserved particle of the SM, that may give light to the origins of mass. The project wants to unveil the possibility for the Unified Force and surpass the frontiers of the SM observing super-symmetric particles or other unobserved physics phenomena.

\(^1\)Luminosity [3] is the number of particles per square-centimeter per second. It is used to measure the performance of a particle accelerator, as the higher the density of particles per area the greatest the number of events.
The experiments running on the ATLAS detector necessitate to gather as much information as possible about the events occurring in the environmental conditions created by the LHC. The ATLAS Inner Detector’s [1] [5] [35] structure consists of three detector systems: the Pixel detector, the Semiconductor Tracker (SCT) and the Transition Radiation Tracker (TRT). The Pixel and the SCT detectors consist of a different number of pixel detectors placed in layers in the barrel and at the end cap wheels, as they are shown in figure 2.1:

Figure 2.1: The ATLAS Detector. [5] On the right figure the ATLAS Inner Detector is shown with the Inner Detector in the center colored yellow. The Inner Detector [35], in greater detail is shown on the left figure. The Pixel detector is in the innermost of the detector’s structure, the Semiconductor Tracker on an outer level and the Transition Radiation Tracker at the outermost level. The barrel layers are shown in the center of the detector and the end cap layers (wheels), are placed at the front and rear of the barrel layers.

The beam passing through the LHC ring shown in figure 2.2 defines the z-direction of the three-dimensional system of the ATLAS detector. Transverse to the beam the planes formed, denoting an area of the detector, are defined by the x and y axis. The x-axis positive direction is the direction defined from an interaction point of the beam to the center of the ring and the y-axis positive direction is the upward direction. The azimuthal angle $\phi$ of an interaction point or a particle’s position is calculated by projecting a vector beginning from the center of the ring to the point of interest on the beam line plane and computing the difference in angle of the projection to a reference vector with the same origin but with apex at the center of the detector. The angle $\theta$ is the angle in which two vectors originating from the center of the ring and pointing one to the point of interest and the other to the beam axis are differing. Both angles $\phi$ and $\theta$ are calculated in spherical coordinates. The pseudorapidity, $\eta$, is defined as $\eta = -\ln(\tan(\frac{\theta}{2}))$ and the transverse momentum of a particle $p_T$ are defined in the x-y plane. These parameters are used throughout the dissertation and for greater detail the
ATLAS Technical design report Volume I is proposed \cite{15}.

Figure 2.2: The LHC ring and an indication of the location of the ATLAS detector.

\section*{2.2 The ATLAS Trigger}

The ATLAS Trigger \cite{19} \cite{35} is responsible for processing the events of the ATLAS detector, storing the identity, energy and tracks of the particles contained in these events. However, not all events are of interest for the ongoing research. Specifically the uninteresting physics events, often referred to as pile-up events, outnumber the interesting events by seven orders of magnitude \cite{35}. As a result the ATLAS Trigger filters the data from the detector keeps only the interesting parts of them for processing. The input rate of event processing is the same as the beam crossing rate of 40 MHz.

To address the filtering and reconstruction of the physics interesting events in this large volume of data, the structure of the Trigger adopts a 3-tier architecture. The first tier, the Level 1 Trigger is mostly based on hardware that is explicitly designed for processing the input data with as less descriptive information as possible from the calorimeters and muon chambers so that in a fixed latency of $2.5 \mu s$ an event is accepted or filtered. An effective way that the Level 1 Trigger exploits in order to reject pile-up events are the Regions of Interest (RoI), which are areas of the detector during an event that the trigger has detected some interesting hits. The RoIs are around the 2% of a total event, minimizing the input to the next level by a factor of about 500. The RoIs are transferred to ReadOut Buffers (ROBs) for further filtering and processing by the High Level Trigger. The latter is composed by the next two software based levels, the Level 2 Trigger and the Event Filter.
The Level 2 Trigger [34] [17] has as input the RoIs and full information data from the detector corresponding to the events of the RoIs so that better performance in particle identification and path reconstruction can be achieved. The latency in this level is increased to 10 ms, having an input rate of about 75 kHz, but upgradeable to 100 kHz and maximum output rate of about 3 kHz. It takes as input positions of the detector layers -Space Points- constrained into a RoI of the Pixel or the SCT Detectors and processes them with a number of on-line algorithms. The Track Fitter is the last routine of the processing pipeline of the tracks passing through the input Space Points. The Track Fitter utilizes a Kalman Filter for the track fitting process. The output of level 2 Trigger algorithms is passed to another routine for full event reconstruction. The output of the latter is then passed to the third tier, where the Event Filter routine dedicated to interesting physics events can have access to full information about the calibration and alignment of the data. This information along with the higher latency permit the algorithm to finalize the on-line event selection. The High Level Trigger on the overall has a rejection factor close to the Level 1 Trigger value, so that the off-line software algorithms can then process the selected events.
2.3 The Kalman Filter

In order for physics the events to be reconstructed, the High-Level trigger utilizes the Kalman filter methods. They are used to reject the tracks belonging to pile-up events and reconstruct the accepted event tracks [35], [17], [34], [23]. The fast reconstruction of these tracks in the Track Fitter relies on the performance of the Kalman filter.

![Particle track reconstruction utilizing the Kalman Filter](image)

Figure 2.3: Particle track reconstruction utilizing the Kalman Filter. The bars represent the detector layers from which the particles pass through. The positions of the particles are an estimation of the predicted position, which are represented as stars and are colored green. The measured positions which include noise from the environment are represented as red stars. The predicted track is represented as a line joining the predicted measurements on the upper figure and the final track is colored red and represented as a line joining the several red stars, which are the noisy position measurements. [23]

2.3.1 The Standard and the Extended Kalman Filter

The standard Kalman Filter [31], [20], [30], [33] 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 of the
The Kalman Filter models the unknown process as a discrete linear stochastic process. A discrete stochastic process is a set of random variables ordered by time, where time has discrete values. The state of the Kalman filter is a random variable of the stochastic process. The system is described by a linear stochastic equation:

\[ x_k = Ax_{k-1} + w_{k-1}, \quad (2.1) \]

where \( x_k \) denotes the state, where \( x \in \mathbb{R}^n \), \( A \) denotes an \( n \times n \) matrix relating the previous states in time step \( k-1 \) with the current state at time step \( k \), and \( w_{k-1} \) is the process noise at the previous time step with covariance matrix \( Q \). The covariance matrix is the matrix of the covariances of the form \( Q_{ij} = \text{cov}(w_i, w_j) = E[(w_i - \mu_i)(w_j - \mu_j)] \) and \( \mu_i = E[w_i] \) is the expected or mean value of the random variable \( w_i \). The measurement at time step \( k \) is represented by \( z \in \mathbb{R}^m \), which is defined as

\[ z_k = Hx_k + v_k, \quad (2.2) \]

where \( H \) is an \( m \times n \) matrix relating the current measurement to the current state and \( v_k \) is the measurement noise with covariance matrix \( R \). Although \( A \) and \( H \) might be changing by we proceed under the assumption that they remain constant.

We define as \( \hat{x}_k^- \) the \( \alpha \) priori estimate of the current state, i.e. the estimated value of the state at time step \( k \) with the knowledge provided till the time step \( k-1 \) and \( \hat{x}_k \) the \( \alpha \) posteriori estimate of the current state, meaning that the measurement noise \( z_k \) has been included to the estimate. We define the \( \alpha \) priori estimation error as

\[ e_k^- \equiv x_k - \hat{x}_k, \quad (2.3) \]

with covariance \( P_k^- = E[e_k^- e_k^-^T] \) \( (2.4) \)

and the \( \alpha \) posteriori estimate error

\[ e_k \equiv x_k - \hat{x}_k, \quad (2.5) \]

with covariance \( P_k = E[e_k e_k^T] \). \( (2.6) \)

The equation that relates the \( \alpha \) posteriori with the \( \alpha \) priori state estimate is

\[ \hat{x}_k = \hat{x}_k^- + K(z_k - H \hat{x}_k^-), \quad (2.7) \]

\footnote{The process and measurement stochastic processes are assumed to be independent, white, and Gaussian.}
where $K$ is the $n \cdot m$ matrix defined as the Kalman Gain.

The filtering process can be summarized as follows:

1. Initial estimation of the process by providing initial estimates for $\hat{x}_{k-1}$ and $P_{k-1}$.
2. Prediction of the estimation of the process for the next time step:
   \begin{align*}
   \hat{x}_k &= A \hat{x}_{k-1}, \quad (2.8) \\
   P_k^- &= AP_{k-1}A^T + Q \quad (2.9)
   \end{align*}
3. Computation of the Kalman gain
   $$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \quad (2.10)$$
4. The filtering step, where the estimation of the process is updated with the measurements and the estimation of the process:
   \begin{align*}
   \hat{x}_k &= \hat{x}_k^- + K_k (z_k - H \hat{x}_k^-) \quad (2.11) \\
   P_k &= (I - K_k H) P_k^- \quad (2.12)
   \end{align*}

Figure 2.4: Overview of the process of the Kalman Filter, where $x_0$ and $P_0$ are the initial values of the unknown process and its covariance matrix, $\hat{x}_k$, $\hat{P}_k$ are the predicted estimates of the process per time step $k$, $x_k$, $P_k$ are the estimates produced from the filtering at time step $k$, and $x_n$, $P_n$ are the final estimates for the whole process.

The **Extended Kalman Filter (EKF)** [20], [30] is 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. In this technique although the
process is modeled as non-linear the equations attempt to linearize the estimation. The equations describing the system now, corresponding to (2.1) and (2.2) are

\[ x_k \approx \tilde{x} + A_k (x_{k-1} - \hat{x}_{k-1}) + W_k w_{k-1}. \]  
(2.13)

\[ z_k \approx z + H_k (x_k - \tilde{x}_k) + V_k v_k. \]  
(2.14)

In this case the matrix \( A \) is the Jacobian matrix\(^3\) of the partial derivatives of \( f \) with respect to the state \( x \), \( H \) is the Jacobian matrix of the partial derivatives of \( f \) with respect to \( w \), \( W \) is the Jacobian matrix of the partial derivatives of \( h \) with respect to \( x \), and \( V \) is the Jacobian matrix of partial derivatives of \( h \) with respect to \( v \). Given the above we can now have the EKF equations for prediction:

\[ \hat{x}_k = f(\hat{x}_{k-1}, 0) \]  
(2.15)

\[ P_k^- = A_k P_{k-1} A_k^T + W_k Q_{k-1} W_k^T \]  
(2.16)

and the equations for filtering:

\[ K_k = P_k^- H_k^T (H_k P_k^- H_k^T + V_k R_k V_k^T)^{-1} \]  
(2.17)

\[ \hat{x}_k = \hat{x}_k^- + K_k (z_k - h(\hat{x}_k^-, 0)) \]  
(2.18)

\[ P_k = (I - K_k H_k) P_k^- \]  
(2.19)

### 2.3.2 The Kalman Filter for Track Fitting

The Kalman Filter is utilized in the track fitting process, where the track of a particle is reconstructed by measurements from each sub-detector\(^3\). The stochastic process estimated is a track of particles intersecting the detector’s layers, and consequently the states of the process represent the space points of the track. The prediction step estimates the next state of the track given the previous states, the filtering step estimates the current state in respect to the predicted state and verifies that this state is a part of track and not a falsely included state introduced due to noise in the measurements. These steps alternate until all the space points identified as part of the track are filtered. When the prediction and filtering steps finish, the smoothing operation follows so that the estimations are refined. The smoother uses the sequence of states generated by the filter as input so that “past” states can be estimated in respect to the “future” ones. This backward propagation of the parameters of the track states,

\(^3\)The Jacobian matrix is the matrix of the first-order partial derivatives of a function \( f \) with respect to a vector \( v \), \( J_{[i,j]} = \frac{\partial f_i}{\partial v_j} \).
from the final track state to the initial has the effect of bettering the estimation, making the trajectory “smoother”.

![Diagram](image.png)

**Figure 2.5:** The track fitting process: Track states are processed by a filter so that the estimation of the complete track is formed before the estimation is refined by a smoother this estimation.

The combination of Kalman filtering and smoothing processing produce optimal track state estimations. This feature is exploited in order to merge track measurements of track hits from different detectors or from several layers of the same detector. It a vital feature for filtering of falsely included detector space points as parts of the track, which may have been accepted by a previous stage as part of the track due to the noise of the environment. The metric used for false space points detection is the $\chi^2$. The $\chi^2$ measure can be computed via the residuals of the filtering for every state using the formula:

$$\chi^2 = r_k^T (R_k^{-1}) r_k - 1.$$  \hspace{1cm} (2.3)

If $\chi^2$ is larger that a bound $c$ chosen to be close enough the expected value of the track, then the state is rejected as an outlier. The global $\chi^2$ is the total $\chi^2$ of all the states of the track. Although this measure may not be able to reject the outliers with values very close to the track’s values, it does not have a great impact on the overall algorithm, as the latter ones do not impact significantly the track fitting process. This measure apart from detecting outliers is also a metric for the quality of the estimation of the filtering algorithm.

The presence of a magnetic field introduces large non-linearities to the system and the use of the Extended Kalman Filter is required. The representation of the system is as follows: the state is a vector $x_k$ consists of 5 parameters describing the track at each space point, where $C_k$ is its corresponding covariance matrix, a function $f_k$ represents the non-linear transition from state to state, $w_k$ represents random noise involved in the transition from one space point to another, $m_k$ is the noisy measurement from the detector, and $\varepsilon_k$ represents the noise in the measurements. The Kalman gain is denoted by $K_k$, the Smoother Gain is denoted by $A_k$ and the residuals of every operation are
denoted by $r_k$. The equations of the system are shown in table 2.1.

<table>
<thead>
<tr>
<th>State and Measurement Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State of the process:</strong></td>
</tr>
<tr>
<td>$x_{k,t} = F_{k-1}x_{k-1,t} + w_{k-1}$</td>
</tr>
<tr>
<td><strong>Measurement:</strong></td>
</tr>
<tr>
<td>$m_k = H_k x_{k,t} + \varepsilon_k$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prediction Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Predicted state:</strong></td>
</tr>
<tr>
<td>$x_{k}^{k-1} = F_{k-1}x_{k-1}$</td>
</tr>
<tr>
<td><strong>Residuals of the prediction:</strong></td>
</tr>
<tr>
<td>$r_{k}^{k-1} = m_k - H_k x_{k}^{k-1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Filtering Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Filtered state:</strong></td>
</tr>
<tr>
<td>$x_k = x_{k}^{k-1} + K_k (m_k - H_k x_{k}^{k-1})$</td>
</tr>
<tr>
<td><strong>Residuals of the filtering:</strong></td>
</tr>
<tr>
<td>$r_k = m_k - H_k x_{k}$</td>
</tr>
<tr>
<td><strong>Kalman Gain:</strong></td>
</tr>
<tr>
<td>$K_k = C_k H_k^T G_k$, where $V_k = G_k^{-1}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Smoothing Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Smoothed state:</strong></td>
</tr>
<tr>
<td>$x_{n}^{k} = x_k + A_k (x_{n+1}^{k} - x_{k+1}^{k})$</td>
</tr>
<tr>
<td><strong>Smoother gain matrix:</strong></td>
</tr>
<tr>
<td>$A_k = C_k F_k^T (C_{k+1})^{-1}$</td>
</tr>
<tr>
<td><strong>Residuals of smoothing:</strong></td>
</tr>
<tr>
<td>$r_{n}^{k} = r_k - H_k (x_{n}^{k} - x_k)$</td>
</tr>
</tbody>
</table>

Table 2.1: The system of equations of the combined Kalman Filter - Smoother algorithm.

---

4The notation $x_j^i$ defines that the state vector at space point $j$ is predicted using information from space points till $i$
2.4 Acceleration Utilizing Graphics Processor Units

The proposed 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 2.6, so they have more capacity of computations in relation to the CPUs.

![Figure 2.6: Comparison of the CPU and GPU floating-point operation capabilities and memory bandwidth. The figure on the left exhibits the increase in floating operations capability of the CPUs in comparison to the GPUs from 2003 to 2009. The figure on the right shows the growth of memory bandwidth of CPUs and GPUs from 2003 to 2010. [28]](image)

The Kalman Fitter routine of the High-Level Trigger involves heavy calculations applied on thousands on tracks and demands very low latency in the throughput of the application which will augment if the proposed LHC upgrade is realized. The GPUs due their superiority over CPUs on floating point calculations are a perfect candidate for exploring new potential speedup [14]. However, the utilization of GPUs for high performance computing applications, such as these, is not straightforward.

2.4.1 Graphics Processing Units (GPUs)

Graphic Processing Units (GPUs) [28] originate from the need of specialized processing units for graphics rendering and their most common use is in gaming industry and graphics cards for personal computers.
Their origin has defined their architecture, as they are optimized over floating-point computations throughput. This feature of GPU architecture is the one that makes it so attractive to general purpose programming applications with heavy computations. As shown above, GPU surpasses the capabilities of CPUs on single and double precision computations maximizing throughput of computations over vast volumes of data. However, the specialization of the GPU architecture to the heavy computational demands has also some drawbacks for general use.

The GPU architecture imposes some restrictions on the programs that are suitable for executing on them. Firstly, the applications should be able to be parallelized in a manner where a set of calculations are executed on several independent and large sets of data. Moreover, the data must have high arithmetic density so that the flow of the program needs a small number of memory accesses relative to the computations of the data transferred, 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.

![Figure 2.7: “Fermi” GPU architecture. The Fermi card has 6 memory partitions supporting the DRAM memory interface, a GigaThread scheduler and a host interface for GPU-CPU connection via PCI-Express situated on the sides of the card (blue). The SMs are drawn as rectangular strips containing the processor cores (light green), their scheduler and dispatch unit (orange), and their shared memory and register file (light blue) situated at the bottom and at the top of the card. In the middle of the card is a Level 2 cache memory shared by the SM’s (light blue).](image-url)
In 2006, NVIDIA \[13\] introduced a new generation of GPU technology, the G80 architecture. The key innovation in this series was the introduction of general purpose programmability to a previously specially designed architecture for graphics rendering. The evolution of this architecture lead to the, recently released, third generation of GPU architecture, “Fermi”.

The “Fermi” architecture includes 16 Streaming Multiprocessors (SMs), a GigaThread scheduler that distributes the instructions to the SM’s schedulers, and one L2 cache memory.

An SM has 32 cores, with one integer arithmetic unit (ALU) and one floating point unit (FPU) optimized over 64-bit operations and precision crucial calculations. Apart from these units, there are 4 special function units (SFUs) on an SM specifically designed for complex operations, e.g. the trigonometrical equation sine, that execute “asynchronously” relatively to the cores. An SM scheduler subdivides the threads executing on the SM to warps, which are sets of 32 parallel threads. There are 2 warp schedulers on-chip with 2 corresponding dispatch units permitting 2 concurrent warps executing on the SM, excluding the case that there are double precision instructions scheduled, in which case the execution is serialized. This concurrency is achieved by assigning one instruction of each warp per half of the cores or on some of the 16 load/store units or on some of the SFUs.

Figure 2.8: The design block diagram of a Streaming Multiprocessor (SM) and of a Core.\[13\]

The are six types of memory for the GPU, global, texture, local, constant, shared mem-
ory, and register file. The on-chip available memory is the register file and an available memory of 64 KB that can be configured as 48 KB of shared memory, used by all threads executing on the same SM, and 16 KB of L1 cache or 16 KB of shared memory and 48 KB of L1 cache depending on what are the needs of the application. If the application needs more shared memory for storing data that are used within the same warp and have constrained bandwidth, the configuration used would be the first one. Otherwise the configuration could be the use of the memory as a L1 cache, decreasing significantly the latency of using global, local, and texture memory. The constant memory is read-only for the GPU and provides a memory space for each multiprocessor where all its threads can read simultaneously. Another cached read-only memory is the texture memory, which belongs to the texture unit, an enhanced cache optimized for two dimensional spatial locality between threads. The global memory is external to the device but L2 cache shared by all SMs and L1 cache on-chip can be utilized in order to decrease significantly the high latency penalty of accessing it 2.9 [28][13].

Parallel Thread Execution (PTX) [29] is the virtual machine and instruction set architecture (ISA) that defines the GPU as a programmable compute device. The PTX version reflects the compute capability of the GPU architecture, for example the compute capability of the “Fermi” architecture is 2.0 as the PTX version.

PTX 2.0 maps the address space of global, shared, and local memories to one virtual address space, enabling load and store instructions to operate on it. The memory
manipulating instructions seem to work on one unified address space, whereas they work on all three address spaces. This is achieved by mapping one set of load and store instructions to a set of load and store instruction to the three separate address spaces.

![Unified Address Space supported by PTX 2.0](image)

2.4.2 Compute Unified Device Architecture (CUDA)

NVIDIA has introduced the Compute Unified Device Architecture (CUDA) programming model for scalable GPU programming [28], [11]. 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. CUDA offers a programming interface for general purpose programming languages as C, C++, and Fortran.

In CUDA, a program can be partitioned to sub-tasks executed in parallel by using kernels. A kernel is a C function executed by device threads in parallel. CUDA threads are organized in a structure reflecting the GPU architecture. They are first grouped into blocks of threads that are going to execute on and share the available memory of the processor cores. They execute the same operations on different data and are synchronized by a barrier. These threads cannot exchange data with threads that belong to other blocks. The number of threads that can be assigned to a block cannot
exceed the 1024 on the current architectures. Blocks are also grouped in another set, grids, which can be one or two dimensional and correspond to SM’s. The number of blocks in a grid 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. The structure described is represented in the diagram below 2.11. A kernel is invoked by the host using a configuration syntax defining the number of blocks per grid and the number of threads per block, for example `mykernel«<numberOfBlocks, numberOfThreads>>`.

![Diagram of thread blocks and grids](image)

**Figure 2.11:** The grid of thread blocks that is specified at compile time and runs on kernel invocation by the host to the device. [28]

CUDA 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 2.12. SIMT is similar to Single Instruction Multiple Data (SIMD) adopted by vector machines such as Cell processors where the same set of operations are executed on different sets of data. However, the parallelization in SIMD spans over the whole of the application whereas in SIMT it is done on thread level. The operations run in parallel until there is a branching point, where they are serialized until the next synchronization point. One of the most important aspects of using GPUs is that context-switching is almost “free” since that thread scheduling is done on the device by the GigaThread scheduler. There
are two alternatives for the use of on-chip memory by blocks of threads. They can either use the shared memory, registers and low-cost hardware-based atomic functions for applications that need to follow shared memory paradigms, or utilize the on-chip memory as a second level cache.

In a CUDA application the host invokes a kernel that executes synchronously or asynchronously on the device. If it is synchronously the host waits for the results of the kernel to proceed, whereas in asynchronous mode denoted by “Async” in memory transfer instructions, the memory transfers are done via streams and the host continues executing. The programming paradigm is such, that the host is responsible for the application’s flow-control and only the heavy-weight calculations are assigned to the device, as it can be seen in the following figure 2.12.

![Diagram showing control flow of parallelized program execution on CPU as the Host and GPU as the Device][28]

The advantage of such a structure is that the programming model is scalable, as the
number of blocks, a kernel is going to be executed by, is defined on run time. A drawback, of this paradigm, is the importance of high data locality for acquiring high performance. Threads share only the on-chip memory corresponding to their block and further data exchanges are serialized degrading performance and increase the required memory bandwidth.

### 2.4.3 CUDA C++

The most significant innovation of the “Fermi” architecture, is the increase of GPU programmability. This is achieved by the unification of the address space resulting to full C++ support. “Fermi and the PTX 2.0 ISA also add support for C++ virtual functions, function pointers, and “new” and “delete” operators for dynamic object allocation and de-allocation. C++ exception handling operations “try” and “catch” are also supported.” [13] This is enabled by overcoming the constraints imposed by the different types of memory address space used in the GPU architecture, constant, global, texture, shared, and register file have imposed. The consequence of having separate address spaces, where the data may reside, was that the memory allocation should be done at compile time. It hinders dynamic allocations of memory space at run time. As a result, the pointers used in C and C++ should be allocated at compile time, whereas in a large number of general purpose applications written in C or C++ dynamic allocation via the use of pointers is the norm, making those applications very difficult to implement in CUDA. Focusing on C++, the operators “new” and “delete” could not be supported by the architecture, as the address space where the data would reside would be ambiguous. This constraint is overthrown by the unification of address space achieved in the last PTX ISA and it is explained more thoroughly in section 2.4.1.

In the most recent CUDA releases some features of C++ are also supported, although the full support of C++, that PTX 2.0 is offering, is not yet supported by CUDA. The features of C++ in CUDA to this release are polymorphism, or alternatively function overloading, the definition of default parameters, operator overloading, namespaces, function templates and classes. However, classes in CUDA do not have full functionality, virtual functions are not supported, so there cannot be any abstract classes or more generally any inheritance implementation. The types of classes supported are the functors, classes with elements and operator definitions for these.
Chapter 3

Track Fitting in the ATLAS detector

The High Level Trigger aims to distinguish interesting physics events from pile-up events and reconstruct the tracks of particles taking part in these events (section 2.2). The Level 2 trigger software is comprised of four routines: the zFinder, the HitFilter, the GroupCleaner and the Track Fitter. The sequence of these routines are shown in the following diagram 3.1. The focus of this project is on the Track Fitter, but for completeness a short description of the algorithms preceding the Track Fitter is given, for greater detail refer to the corresponding publications [12] [34] [17] [35] and the technical report of the ATLAS detector [15] [16].

The zFinder [12] takes as input a selected area of the inner detector, where level 1 trigger has detected some interesting physics events. The cylindrical area of the RoI
is partitioned into small slices of angle $\phi$, in order to refine the processing of the helix trajectories of the particles stemming from an interesting interaction. Pairs of three-dimensional space points of the Pixel or SCT detector, belonging to one $\phi$ slice and its neighbor slices, are combined using linear-extrapolation with the beam line. The beam line lies on the $z$-direction with $z=0$ being the center of the detector, to determine the pair’s $z$-position. This is the position where they intercept with the beam line. The $z$-positions of all pairs are accumulated to a histogram, where the peak of the histogram, which denotes that the most hit pairs specified this $z$-position, is recognized as the $z$-coordinate of the proton-proton collision, and the rest are ignored as pile-up event hits.

The algorithm, that follows in the software pipeline, is the HitFilter that uses the $z$ coordinate in order to compute the pseudorapidity $\eta^1$ of all the hits and create another two dimensional histogram ($\eta, \phi$). All the hits that contributed to the computed $z$-position will be in a small solid angle with its apex at the $z$-position of the physic’s event. All the space points, from each track that contributed to this $z$-position value, form a cluster of neighboring bins in the histogram. For each of these bins, the number of different detector layers that contain hits belonging to them are counted and if they are not at least four out of the seven expected layers, the bin is rejected as a fake track candidate. The accepted bins form groups from neighboring clusters of hits. The groups that were formed by the HitFilter algorithm are processed by the GroupCleaner, so that the space points included in each group are mapped to a candidate track.

### 3.1 Track Fitter Algorithm

The candidate tracks formed from the HitFilter are passed to the Track Fitter in order to be reconstructed before the event is passed to the Event Builder for full event reconstruction. The Track Fitter algorithm is implemented according to an abstract interface meant for track fitting tools of the ATLAS Inner Detector. The specific implementation is described below.

Recorded tracks of the accepted events are streamed into the Track Fitter, consisting of the hits on the Pixel and SCT detectors’ layers. The structure of the classes corresponding to the input of the algorithm is shown in the Figure 3.2. The recorded tracks

---

1The pseudorapidity is a variable characterizing a particle when the only available measurement is the angle of the particle in relation to the beam axis, $\theta$, and it can be defined as $\eta = -\ln(tan(\frac{\theta}{2}))$ [18]
are instances of the RecTrack class, which includes all the parameters computed by
the previous algorithms. The space points, of the SpacePoint class, are decomposed
into clusters depending to the layers of the detector that the hits correspond to. They
can either be from the Pixel or the SCT wafers for this algorithm. The clusters from
the Pixel wafers are instances of the PixelCluster class and the ones from the SCT
wafers are either instances of SCTBarrelCluster class, if the hits are from the barrel
layers, or of SCT_EndCapCluster class in the case the hits are from the end cap layers
of the SCT detector. For every cluster there is a corresponding Surface that is formed
from the position of the hit in three-dimensional coordinates and momentum direction
of the particle for the particular cluster.

The candidate tracks from the previous algorithms are the recorded tracks of the trigger
and the tracks that are going to processed and possibly accepted by the filter encap-
slate the information from the recorded tracks and acquire some additional from the
process. These tracks are of type DkfTrack which at the end of the algorithm will be
the reconstructed tracks. The clusters have position information, and the description
of the magnetic field at the point of the hit which are passed on to the filtering nodes.
These nodes are specialized to the area of the hit as the clusters are and consequently
a similar to the cluster inheritance hierarchy is produced, with the difference that now
this filtering node objects also encapsulate the algebraic equations of the filtering pro-
cess. These equations differ for each detector, as for the Pixel detector the filtering
process is based on a two-dimensional measuring system, in contrast to the SCT de-
tector where a one-dimensional measuring system is followed. Apart from that there
different features for barrel hits and end cap hit as far as the filtering process is con-
cerned. These specializations are done for better representation and results of the track
fitting in different detectors and areas of detectors. The filtering nodes inheritance
structure is shown in the Figure 3.3, where the base class is the TrkBaseNode, the
next level of hierarchy defines the measurement model used and there are two classes
representing them, TrkFilteringNode1D and TrkFilteringNode2D, and at the last
level the clusters of the Pixel detector are filtered by object of the TrkPixelNode, as
the clusters from the SCT are filtered corresponding to the two kinds of layers, the
TrkClusterNode for the barrel and TrkEndCapClusterNode for the end cap layers.

The Kalman Filter algorithm for track fitting (section 2.3), is a mathematical tool for
optimal estimations of particle tracks. The tracks consist of three-dimensional space
points in the detectors’ area and as the track progresses in space, the track state retains
both track information and estimation of the filter for the current space point of the track. The parameters that uniquely describe the track at a given state are the following, as they are described in 2.1 and measured in the global coordinate system:
- local x and y, which define the transverse plane to the beam line,
- the azimuthal angle $\phi$
- the angle from the beam direction, $\theta$
- the track inverse momentum $Q$,
- and the corresponding covariance matrix.

These parameters are the elements of TrkTrackState and their initial values are the parameters computed by the previous algorithms.

A cluster with the particle’s momentum direction define a surface which is modeled by Surface. When the cluster parameters are passed to the filtering node object, the associated surface passes its parameters to a new class object of type TrkPlanarSurface, as shown in the diagram 3.5. The planar surface object retains further information such as the radiation thickness. The classes providing all the information for the fitting process are shown in the UML class diagram, the DkfTrack, the TrkTrackState and the classes belonging to the filtering nodes’ inheritance scheme with base clase TrkBaseNode. These classes are the ones we are focusing on, since they interact in the context of this routine.

The Kalman Fitter algorithm is modeled as a class TrackFitter, controlling the track fitting process. For every candidate track the fitting function of the Track Fitter is called. The prediction step of the kalman filter is the initial one, where given the initial values of the track states and the planar surfaces, the predicted estimates of the track states and planar surfaces are produced as shown in figure 3.5. These extrapolated track states are validated as part of the track or rejected as outlier using the $\chi^2$ distance as a measure.

![Figure 3.5: The extrapolation process of a track state, given the measurement of the position of the cluster, the associated planar surface, and the updated state, and the extrapolation of the next state via the previous state](image)
The next step is the filtering step, where the validated extrapolated track state estimates are incorporated with the noisy measurements of the detectors, which are the cluster parameters stored in the filtering nodes. The filtering is done according to the $\chi^2$ distance and the resulting estimates have to be refined so the particle track to be “smoother”. All the information from the forward extrapolation and the filtering is used for a backward extrapolation technique, the smoothing process (section 2.3.2). The smoother takes as initial input the final track state of the track candidate, predicts the previous state given the current state and proceeds. The outcome of this process optimizes the filtering procedure while the final $\chi^2$ estimates of the track states are added together in order to produce the global $\chi^2$ estimate of the track filtering as a metric of estimation quality.
Figure 3.3: The filtering nodes inheritance scheme represented as a UML 2.0 class diagram with the use of Umbrello [9]
Figure 3.4: UML class diagram presenting the interconnection of the classes providing all the information for every step of the fitting process. DkfTrack represents the candidate track, TrkTrackState class represents a track state, TrkPlanarSurface the surface associated with a cluster, and the inheritance scheme with base class TrkBaseNode encapsulates the description of a cluster and the fitting elements and equations for this cluster. This is a UML 2.0 class diagram generated using Umbrello [9].
Chapter 4

Our Approach on Acceleration of the Kalman Fitter

4.1 SIMT design approach of the Kalman Filter

The Kalman Fitter algorithm implementation is based on three functions the fitting function, the extrapolation function and the smoothing function, as described in section 3.1. The fitting function is the one that encapsulates the fitting process. For every cluster of an event, a function is called to produce the filtering node object for the current track state, the planar surface and the object of the track state.

This whole process involves some more functions devoted to Kalman filter equation computations, but they are of a smaller scale. The extrapolation and smoothing functions dominate the application’s run time. Parallelizing the computations of these two functions, would introduce a major drawback, since both these functions have a closed loop behavior, which creates strong input-output dependencies. In other words, when the output of one step is the input of the next step, parallelization on this level is deterred.

A proposed way of parallelizing this algorithm is by having one candidate track fit per GPU thread. This approach bears many advantages, since there are no dependencies among the tracks. Even if the tracks belong to the same event, all the information that is needed in order for the fitting process to complete are encapsulated to every track. The clusters, their corresponding surfaces and the track states processed by the filtering nodes are only dependent in the tracks context, but independent from other
tracks. Additionally, the magnetic field description is specialized for every track, as it depends on the space points that the candidate track consists of.

Adopting such an approach can be beneficial as it can potentially fully exploit the capabilities of the GPU architecture. The GPU architecture is optimized over throughput, meaning that apart from its capabilities of single and double precision computations, which are also indicated by the graph of CPU and GPU computational comparison 2.6, it is designed to perform dense mathematical computations in parallel. This kind of parallelism aims at accelerating the whole process, by having as many tracks as possible in parallel. The threads of the GPU architecture are spawned and controlled by the architecture’s schedulers and even context switching is done on hardware level, accelerating the process of creating and maintaining a large number of threads. The number of threads that can be executing in parallel this routine depends on the capabilities of the machine.

However, this approach has also some drawbacks on the implementation level. First of all the code has many objects with a large amount of data originating for the needs of highly optimized estimation precision in a complex environment such as the ATLAS Inner Detector. Moreover, all the data from intermediate steps of extrapolation are retained for outlier rejection and smoothing. Thus, the memory allocation and manipulation is not straightforward.

The mathematical optimizations have lead to hierarchies such as the one of the filtering nodes, which lead to having different implementations depending on the instance. This causes threads to branch according to the class functions they are instances of, and when they are executing in warps, this change in the sequence of instructions causes serialization of their execution and degradation of the parallelized application’s performance.

### 4.2 Related Work

S. Gorbunov et al., in 2007, ported the Kalman filter of the CBM experiment to Single Instruction Multiple Data (SIMD) architecture, SSE and Cell SPE with a speedup of 10000. CBM is a heavy-ion experiment at the Facility of Antiproton and Ion Research (FAIR) in Darmstadt, Germany. Two optimizations lead to this speedup, the first was memory optimization specialized for the CBM experiment and
the second was the numerical stability of the square root Kalman filter. As in this project we are not dealing with the CBM experiment, thus 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., [26], [25] 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 [33]. 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.
Chapter 5

C++ SIMT Design

The approach of parallelizing the fitting process at thread level can be implemented via transforming the fitting C++ function into a CUDA kernel. This has the consequence of porting all C++ source code to the device, which involves not only the implementation of the mathematical equations, but also the creation and manipulation of a large number of class objects. The implication of executing all the functions of the Kalman Fitter to the GPU machine is hindered by the fact that C++ is not fully supported by CUDA yet, as this is an on-going research, some of the results of which are presented in the next section.

5.1 Constraints of integrating the C++ Kalman Fitter with CUDA

The parallelization of the Kalman Fitter via executing one candidate track fit per GPU thread is problematic given the current releases of CUDA. The issues that arise make the specific implementation of the Kalman Fitter at the time infeasible for porting it on the GPU device.

First of all, the structure of the code is such that the track fitting function, \texttt{m_fit()} has to be re-implemented as a kernel. A CUDA kernel cannot dynamically allocate or deallocate memory; all the memory allocations and transfers from the CPU must be done before the invocation of the kernel. The problem arising is that the functionality of this function is dominated by dynamically allocated objects. The first function called by
m_fit(), m_createDkfTrack(), creates the instance of candidate track to be processed, the filtering node objects corresponding to track’s clusters, and the associated surface objects, and the instances of the track state class, TrkTrackState. The operators “new” and “delete” are vital for retaining the full functionality of these functions. The density of the dynamic allocations and the memory management manipulation is exhibited in the following pseudocode.

Algorithm 1 Creation of the class instances needed for the track fitting process

DkfTrack m_createDkfTrack (RecTrack recordedTrack)

inputs: RecTrack recordedTrack : The recorded candidate track formed by the previous algorithm of the Level 2 Trigger

outputs: DkfTrack candidateTrack : The candidate track instance for the fitting process.

magneticFieldIndex=1;
create a vector magneticFieldVector;
trackState=new TrkTrackState;
candidateTrack=new DkfTrack;

for every space point of the recordedTrack do
  for every cluster of the space point do
    if cluster instanceof SCTBarrelCluster then
      filteringNode = new TrkClusterNode;
    else if cluster instanceof SCTEndCapCluster then
      filteringNode = new TrkEndCapClusterNode;
    else if cluster instanceof PixelCluster then
      filteringNode = new PixelNode;
    end if
    add filteringNode in the list of nodes of candidateTrack;
    planarSurface = new TrkPlanarSurface; {including the parameters from the cluster surface}
    add the fieldIndex to magneticFieldVector;
  end for
  increment magneticFieldIndex;
end for

Pseudocode 1 presents three elements of the code. The first one is the interconnection
of the classes from the previous Level 2 Trigger routines, RecTrack, Surface and the inheritance structure with base class SiCluster shown in UML diagram 3.2, with the classes taking part in the filtering process, TrkTrackState, TrkPlanarSurface, and the inheritance structure with base class TrkBaseNode as shown in UML diagram 3.3. Additionally, it is shown that the information passed to this routine is encapsulated to the instances of its classes. The instances of filtering nodes correspond to the clusters and the instances of TrkPlanarSurface absorb the parameters from the instances of the Surface class.

An important feature of this code is that every track consists from many space points, which are dissolved into clusters and for each of these clusters the following objects are created, a filtering node and a planar surface. Each step of the algorithm needs a filtering node object, a planar surface object, and a track state object in order to proceed. As a consequence, the memory resources are challenged by the domination of these objects at the address space. Their main characteristic is to point to each other’s address space, which follows from their class interconnections as shown in the Figure 3.4. This adds to the complexity of the code especially when these objects have to be created in the GPU device memory.

Even in the case when this function is not called by the fitting function, all the instances of the above mentioned classes have to be allocated in advance for all the candidate tracks. CUDA supports small aggregate classes or functors, but this is not the case for the classes that we have in this source code. The filtering nodes are represented by the multiple inheritance structure with base class TrkBaseNode. Apart from that, all these objects have references to each other something that is beyond the functor functionality.

5.2 Libraries for integration of CUDA with C++

CUDA for C++ is not the only choice for porting an application to a GPU machine, CuPP and Thrust are two popular alternatives that enhance the programmability of CUDA applications. Thrust[21] is a library for CUDA integration of C++ template classes and generic algorithms similar to those provided by the Standard Template Library. These features do not find application in this case, as there are no members of this library or of similar algorithms in this source code.
Another alternative is the CuPP framework\cite{22} especially developed for increasing the programmability of CUDA regarding C++. It eases the integration of C++ source code with CUDA by offering libraries that encapsulate the functionality of memory management, kernel invocation, and device management. Memory allocation and deallocation via instances of functor classes is transparent to the programmer and thus the complexity of the CUDA C++ code is reduced. Even though it offers more functionality than CUDA, it is based on CUDA and it cannot surpass it in a way that surpasses the capabilities of the PTX ISA or the current research of the C++ support. It offers functionality in allocating and deallocating containers, but does not support the kinds of functions used in the routine of the Kalman Fitter, or the custom-defined data structures, which is the only kind of structures used as inputs in this source code.

CuPP and Thrust are useful when the typical heterogeneous programming paradigm is followed, where the flow control is conducted by the CPU and only the functions of high computational demands are executed by the device.

The difficulties addressed with C++ source code lead to the implementation of the Kalman Filter in the C programming language, which CUDA is based on, in order to have full CUDA support.
Chapter 6

Implementation of the Kalman Fitter in C

The C++ implementation of the Kalman Fitter in combination with the programmability of CUDA for C++ is such that hinders the integration of the original source code with CUDA. This gave rise to the need of re-writing the code in C in view of integrating this code with CUDA following the initial SIMT approach. This was not a straightforward task due to the complexity of the code. As a consequence, the aim of the first C implementation is to achieve the same results with the C++ implementation.

The first C implementation followed the original design of the Kalman Fitter code and retained its structure as closely as possible. The elements of the classes are now stored in C structs with their corresponding names and the class functions are now C functions that retain their functionality and names, even though the latter is done only for clarity. They work on a by-reference copy of the struct that contains all the elements of the corresponding class object. Another set of classes that had to be re-written were those belonging to an inheritance hierarchy, in order to eliminate the inheritance scheme, which is not supported in C. The hierarchy structure of the filtering nodes, shown in the UML diagram 3.3, is re-organized in the form that all the elements of the three hierarchy levels are included into the third level, resulting to the structs `ClusterNode`, `EncCapClusterNode`, and `PixelNode`. The functions that operate on these structs previously belonged to the corresponding classes of the inheritance hierarchy. The next step, was the implementation of the vectors as lists specialized for each of the structs they were going to store, as templates or abstraction is also not supported in C.
Table 6.1: Results of the same operations computed in C and C++ programs: both programs have the same operations using the `<cmath>` library but differences in precision lead to noticeable differences.

Additionally, the TrkBaseNode struct has now three pointers of type TrkPixelNode, TrkClusterNode, and TrkEndCapClusterNode, from which only one is not NULL at a time, containing the location of the corresponding structure.

The implementation of the routine to C from C++ has a number of implications. The first one is that the C++ and the C programs produce different results for the same operations. This is caused due to different rounding methods in the floating point operations, especially when dealing with very small numbers, which is the case for this application. These differences in precision are initially very small. However, due to the repetitive behavior of the algorithm, since the output of one step is the input to the next, they are accumulated producing comparatively very different results. An example of how these variations from negligible differences in decimal points lead to great differences is shown at the table 6.1, where the final results for the assessment of the quality of the estimation are computed using the trigonometrical function sine of `<cmath>` library in both programs and both being compiled by the g++ 4.1.2 compiler.

Another implication of the C implementation is the increase in volume of the source code. The complexity of the code, though, is retained at the same levels. The relations of the different structures involved in this process are complex and this becomes more obvious when dealing with memory management on a lower level. Most of the structs of the source code consist of pointers to other structs or arrays of primitive and custom-defined data types. Allocating such kind of structs on the device memory is an error-prone procedure, which in this scale of complexity becomes problematic. Apart from that, the volume of data to be allocated on the device is extensive due to the fact that dynamic allocation is not supported by the GPU device, which leads to an altered version of this code, where all the memory address space allocation has to be done prior to the invocation of the kernel for track fitting. In the new version of the code all the structs needed for the fitting process are allocated in advance, forming arrays with
outer dimension equal to the number of tracks to be processed. Another implication of this modification is that the routine’s speed is decreased due to an additional iteration over events for the initialization of the structs.

The design of this routine involves the utilization and updating of the following data structures for each fitting step, TrkPlanarSurface, TrkTrackState, TrkBaseNode, TrkClusterNode, TrkEndCapClusterNode, TrkEndCapClusterNode, and TrkPixelNode. Each of those provides some data for the process but this is done in a manner that forms a chain of pointers from one to another rather than directly providing the data to the corresponding stage of the process.

An alternative approach is to eliminate or at least decrease the chain of pointers, which complicates the transfer of these structs into device memory and decreases the possibility of accelerating the algorithm. The first step is the use of dynamic arrays instead of lists. Even though the reallocation of arrays may be the best option as far as C is concerned, CUDA does not support it and the alternative adopted is to allocate the dimensions of the arrays as soon as they are available.

The candidate track structs are stored in a one dimensional array type DkfTrack equal to the number of tracks, $t$. Similarly, the nodes are stored into a two dimensional array of type TrkBaseNode relative to the number of filtering nodes, $n$, with exact dimensions $t \times n$. The track states of the candidate tracks form a $t \times (n + 1)$ array of type TrkTrackState, same as the array of planar surfaces. Two one dimensional arrays, must also be added in this straightforward conversion, with as many variables of TrackFitter and RecTrack as there are tracks.

Furthermore, the use of arrays rather than lists has eliminated the need of having pointers from one struct to another. The DkfTrack struct does not need to keep the pointers of the list of filtering nodes and planar surfaces, neither do the TrkTrackState and the TrkBaseNode need to do bookkeeping of the correspondence of filtering nodes to track states and planar surfaces, as this correspondence is already maintained by the array data structure. This happens because all the arrays differ in only one dimension, so if the common dimensions are set to the same value, the variable acquired is the one corresponding to the rest. For example, the filtering nodes’ array and the planar surfaces’ array have the first dimension equal to the byte size of their structs in memory, the second dimension is equal to the number of nodes, and the third dimension equals with the number of tracks to be processed. Setting the second and third dimension of
these two arrays respectively to the same value, leads to acquiring the values that corre-
respond to each other without the need of storing each other’s pointers into the structs
themselves.

These changes were not enough, since the reduction of the required memory address
space is small. The next step was to evaluate which of the attributes of the tracks
were useful for the process after these changes and isolate these as much as possible,
in order to decrease the number of multi-dimensional arrays of custom-defined data
types. In this way, the type of TrackFitter was deleted, as its attributes were either
used locally, and there was no reason for globally assigning space for them, or the
one array that needed to be retained, was of primitive type int, so it easier to handle
as CUDA array. The same applied to RecTrack data type. After some changes to rest
of the data types, eliminating attributes that had only local use and not globally used,
the simplified structures were much easier to manipulate, the detailed description of
the alterations are in the tables 6.2, 6.3, and 6.4. The resulting code is used for the
proposed implementation of the device code of the Kalman Fitter.
### Table 6.2: Modifications of the structs DkfTrack, TrackFitter for less memory consumption.

<table>
<thead>
<tr>
<th>Struct</th>
<th>Elements</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DkfTrack</td>
<td>double m_dChi2, int m_nNDOF</td>
<td>Retained</td>
<td>Variable storing global $\chi^2$</td>
</tr>
<tr>
<td>TrkTrackState</td>
<td>* m_pTrackState</td>
<td>Deprecated</td>
<td>The array index of the DkfTrack structs array suffices</td>
</tr>
<tr>
<td>TrkPlanarSurface</td>
<td>** m_vpSurfaces</td>
<td>Deprecated</td>
<td>The list of surfaces is stored into an array defined for all tracks</td>
</tr>
<tr>
<td>TrkTrackState</td>
<td>** m_vpTrackStates</td>
<td>Deprecated</td>
<td>The list of track states is stored into an array defined for all tracks</td>
</tr>
<tr>
<td>TrackFitter</td>
<td>double m_startB[3], double m_endB[3], MapList, * m_reMapList</td>
<td>Deprecated</td>
<td>The struct storing all the global variables of the fitting process</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Used only locally</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The elements of the array are stored into an array for all tracks</td>
</tr>
<tr>
<td>Struct</td>
<td>Elements</td>
<td>Status</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------</td>
<td>----------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TrkPixelNode</td>
<td>int m_nodeState</td>
<td>Retained</td>
<td>Struct of the filtering node encapsulating a cluster from the Pixel detector</td>
</tr>
<tr>
<td></td>
<td>double m_chi2Cut</td>
<td>Retained</td>
<td>Threshold initially set for outlier detection</td>
</tr>
<tr>
<td></td>
<td>double m_dChi2</td>
<td>Retained</td>
<td>Local $\chi^2$ distance</td>
</tr>
<tr>
<td></td>
<td>int m_ndof</td>
<td>Retained</td>
<td>Degrees of freedom</td>
</tr>
<tr>
<td>TrkTrackState</td>
<td>* m_pTrackState</td>
<td>Deprecated</td>
<td>The filtered track state</td>
</tr>
<tr>
<td></td>
<td>double m_B[5][2],</td>
<td>Deprecated</td>
<td>Locally utilized</td>
</tr>
<tr>
<td></td>
<td>double m_D[2][2],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>double m_H[2][5],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>double m_K[5][2],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>double m_resid[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>double m_m[2]</td>
<td>Retained</td>
<td>Measurement</td>
</tr>
<tr>
<td></td>
<td>double m_V[2][2]</td>
<td>Retained</td>
<td>Input from previous routines</td>
</tr>
</tbody>
</table>

Table 6.3: Modifications of the structs TrkPixelNode for less memory consumption.
### Table 6.4: Modifications of the structs TrkClusterNode and TrkEndCapClusterNode for less memory consumption.

<table>
<thead>
<tr>
<th>Struct</th>
<th>Elements</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrkClusterNode</td>
<td>int m_nodeState</td>
<td>Retained</td>
<td>Structures of the filtering node</td>
</tr>
<tr>
<td></td>
<td>double m_chi2Cut</td>
<td>Retained</td>
<td>from the SCT detector</td>
</tr>
<tr>
<td></td>
<td>double m_dChi2</td>
<td>Retained</td>
<td>Threshold initially set for outlier detection</td>
</tr>
<tr>
<td></td>
<td>int m_ndof</td>
<td>Retained</td>
<td>Local $\chi^2$ distance</td>
</tr>
<tr>
<td>TrkEndCapClusterNode</td>
<td>double m_B[5],</td>
<td>Deprecated</td>
<td>Degrees of freedom</td>
</tr>
<tr>
<td></td>
<td>double m_D,</td>
<td></td>
<td>The filtered track state</td>
</tr>
<tr>
<td></td>
<td>double m_H[5],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>double m_K[5],</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>double m_resid</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>double m_m</td>
<td>Retained</td>
<td>Measurement</td>
</tr>
<tr>
<td></td>
<td>double m_V</td>
<td>Retained</td>
<td>Input from previous routines</td>
</tr>
<tr>
<td></td>
<td>double m_Rc</td>
<td>Retained</td>
<td>Input only for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>TrkEndCapClusterNode</td>
</tr>
</tbody>
</table>

#### 6.1 SIMT design of the C version of the Kalman Fitter

The SIMT version of the C code adopts the general approach we have introduced in Chapter 4, where every thread is responsible for executing the fitting process for one candidate track. This could potentially produce significant speedup even though the track reconstruction process is not accelerated for one track, as there can be an increase in the throughput of the routine. The implementation of such an approach necessitates that the fitting function has to be converted to a CUDA kernel.

The implementation of the SIMT design approach was hindered by the problems discussed above, regarding memory allocation. All the data structures that are going to
be used in the filtering process have to be allocated in advance and transferred from
the host to device memory. A proposed way of achieving this with the less possible
problematic structure and device management is by using array structures in CUDA.

There are two alternatives for allocation of arrays in CUDA, either by utilizing CUDA
arrays, which are structures allocated in texture memory or allocate the arrays as
pitches that are going to be allocated in global memory. Although, CUDA arrays
are a very attractive alternative as they gather many advantages, such as read and
write optimization as far as transfer time is concerned and programmer defined pat-
terns with which the elements can be accessed, they reside in the read-only texture
memory, which is not desirable in this case.

The proposed option is the allocation of the arrays in the linear memory with use
of cudaMallocPitch() for two-dimensional arrays and cudaMalloc3D() for three-
dimensional arrays. These functions patch the memory allocation so that it satisfies
the memory alignment requirements gaining in transfer time and optimizing band-
width utilization. This kind of memory allocations are optimized for read and write
interleaving when threads are not trying to access the same location of an array. This
can be exploited in this case as the data of the candidate tracks bear no dependencies
among them and consequently the threads do no cause any memory race conditions.
For example, if the candidate track structs were stored in a list, all the threads in the
worst case scenario would have to pass through all the nodes of the list until the target
node. Even if a more optimized way of traversing the list was implemented, threads
would still need to access the same address spaces simultaneously, causing memory
race conditions. This would bandwidth utilization downgrade and unreasonable mem-
ory management of thousands of pointers.

Moreover, the use of arrays simplifies the structure of the code and it is easier to do
optimizations such as selectively copying from global memory to shared memory the
pitch that a thread is using so that the penalty of accessing global memory would be
paid only twice, at the time of the fetching from global memory to shared memory
and back to global memory. Of course, for such an optimization to be feasible the
data volume of the pitch should not exceed the shared memory of the thread block.
Another, optimization that could be feasible is to partition the volume of the input data
and transfer asynchronously through streams the data from host to device and vice
versa.
The refactoring of the code has resulted in an implementation that could be integrated with CUDA, as its complexity and required memory resources have decreased. This decrease, however, was not enough for the routine to be ported on the device. The volume of data that the routine has to create and manage still exceeds the memory resources of the device. To be more precise, there are six arrays that have to be stored in global memory so that the device can proceed with the fitting process, shown in Table 6.5. 3034 tracks of muons were included in the dataset of this study, consisting of twelve clusters each in the worst case. The problem arising is that apart from the theoretical size shown below, there are also some bytes of pathing added to the first dimension, or “width” of the allocated arrays, which increases for larger data-structures. Unfortunately, the CUDA source code for allocating these arrays runs out of memory with the first non-primitive data type allocation. The sample code for allocating a two-dimensional pitch and a three-dimensional pitch are shown at listings 6.1 and 6.2 respectively.

<table>
<thead>
<tr>
<th>Array</th>
<th>width</th>
<th>height</th>
<th>depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>dkfTrackArray</td>
<td>sizeof(DkfTrack)</td>
<td>number of tracks</td>
<td>-</td>
</tr>
<tr>
<td>BFieldPointArray</td>
<td>3 sizeof(double)</td>
<td>max(number of nodes)</td>
<td>number of tracks</td>
</tr>
<tr>
<td>reMapArray</td>
<td>sizeof(int)</td>
<td>max(number of nodes)</td>
<td>number of tracks</td>
</tr>
<tr>
<td>plSurfacesArray</td>
<td>sizeof(TrkPlanarSurface)</td>
<td>max(number of nodes)</td>
<td>number of tracks</td>
</tr>
<tr>
<td>trackStateArray</td>
<td>sizeof(TrkTrackState)</td>
<td>max(number of nodes) + 1</td>
<td>number of tracks</td>
</tr>
<tr>
<td>trackNodeArray</td>
<td>sizeof(TrkBaseNode)</td>
<td>max(number of nodes)</td>
<td>number of tracks</td>
</tr>
</tbody>
</table>

Table 6.5: The arrays of data structures to be transferred at the GPU device memory: the dkfTrackArray is the array of candidate tracks that stores the estimation measures of the process, the BFieldPointArray array includes the parameters of the different B field points of the magnetic field depending on the initial space points of the recorded tracks, the reMapArray stores the correspondence of B field points with clusters, the plSurfacesArray stores all the planar surfaces for the extrapolation process, the trackStateArray the track states for the track reconstruction, and the trackNodeArray the filtering nodes for the filtering process.

Listing 6.1: Allocation and memory transfers from and to the device of 2D pitch in CUDA for the array dkfTrackArray

```c
1  cudaError status;
2  size_t dkftrack_pitch;
3  status = cudaSuccess;
4  __global__ DkfTrack * _dkfTrackArray;
5  status = cudaMallocPitch(& _dkfTrackArray, & dkftrack_pitch, sizeof(DkfTrack), sizeof(DkfTrack) * tracks_no);
6  if (status == cudaSuccess) { printf("Successfully allocated _dkfTrackArray\n");}
7  else { printf("_dkfTrackArray_MallocPitch\n", cudaGetErrorString(status));}
```
In the source code above, in lines 1 to 9, the `dkfTrackArray` is allocated in device memory as a pitch, utilizing `cudaMallocPitch()`, which takes as parameters the size of the pitch in bytes and returns the pointer of the pitch in the device address space and the actual size of the array in device memory. The `cudaError` variable `status` retains the return flag of the allocation command, whether it is `cudaSuccess` or an error code. The same variable is used for the call of the memory transfer command for the two-dimensional array, `cudaMemcpy2D`, which takes as parameters the source and destination array, the byte size of the pitch to be transfered and the parameter `cudaMemcpyHostToDevice` defining the direction of the transfer from the CPU to the GPU. The opposite memory transfer is done in lines 16 to 19, using the same memory but swapping the source and destination pointers and changing the last parameter to `cudaMemcpyDeviceToHost`. The allocation and transfer were both successful. In the following source code the allocation of three-dimensional arrays is exhibited.

Listing 6.2: Memory allocation and transfers from host to device and vice versa of a 3D array allocated as pitch of primitive type double and of struct `TrkPlanarSurface`

```c
/* an array of the B field Points fields */
double *bFieldPointArray [tracks_no][refmaxfields][3];

/* reads events from input file as it is a standalone version */
tracks_no = m_readEvents (argv[1], vpEvents, refmaxfields, refmaxnodes);
size = tracknd->pTrack->m_fieldMap->size;
bFieldPointFields [track_cnt][track_cnt] = double* malloc (sizeof (double) * size);
cBFieldPoint = tracknd->pTrack->m_fieldMap->phead;
for (i = 0; i < size; i++)
    for (j = 0; j < 3; j++)
        bFieldPointArray [track_cnt][i][j] = cBFieldPoint->m_field[j];

/* allocate memory of b field points */
cudaPitchedPtr _bFieldPoints;
size_t bfp_width = 3, bfp_height = refmaxfields, bfp_depth = tracks_no;
cudaExtent bfp_extent = make_cudaExtent(bfp_width, sizeof (double), bfp_height, bfp_depth);
```
The allocation of three-dimensional arrays in CUDA is different than the two-dimensional allocation. First of all, the array has to be allocated in as a pitch in the host heap, as it is shown at line 2, for a successful transfer to the device. This results in a lower level manipulation of the allocated address space not only to the device, but also at host. The source code in lines 7 to 16 exhibits the way the address space must be manipulated in host, but it is also the same for the device. The allocation on the device is done by `cudaMalloc3D` which takes as input a set of specifications for the memory allocation, which are mainly included in `cudaExtent` which is created as shown in lines 27 and 47. The configuration of the `cudaMemcpy3DParms` is the only encapsulates all the information needed for the copy, including source and destination pointers, extent and the kind of transfer, such as the transfer from host to device.

The SIMT CUDA C version of the code proposed is to allocate all the memory needed from a number of events in arrays in host memory, by starting to transfer these data to the device while preparing some more data from the events. The data used by the threads from the same block should be transfered to shared memory, so that the
accesses to these data will cost less with respect to high spatial and temporal locality in the input and output data of the filtering and smoothing functions. The kernel function of fitting process \( \text{m_fit}() \) will asynchronously with the data transfers transfer the output data to the host device, exploiting the capability of the “Fermi” architecture to have simultaneous transfers from and to the host memory if the memory space is different. This is feasible as partitioning the data in order to avoid out of bounds memory errors can be done and it would be able to produce a more scalable approach than having more than one CUDA enabled GPU devices executing the code, although in the scale of the ATLAS framework of course the latter will be also utilized.

The refactoring of the code has lead to an implementation of the same routine which can potentially be integrated with CUDA, due to complexity decrease by simplifying data structures and their interleaving. However, there is another part of the code that has not been mentioned yet, the actual filtering functions. The step that follows the memory address space allocation is the execution of the filtering and smoothing functions. It is not trivial to port these functions to the device even though all the data may reside into the device memory, as the memory has to be addressed utilizing the thread and block identifiers and the equations that take part in the fitting process have intense use of matrices that need to be meticulously converted. Unfortunately, this task was not implemented due to the focus on the memory management issues. The proposal for the implementation of the kernel is to run one thread per block, in order to utilize the on-chip shared memory as a local cache and avoid exceeding the register file size, that would cause even greater memory performance problems. Another important reason is that having one thread per block ensures that the unavoidable branching of the code due to specialized equations of the filtering nodes is not going to cause any kind of serialization of the code, as blocks are scheduled independently. The number of blocks in the simple case can be as many as the cores of the available devices and if streaming is utilized the CUDA code could exhibit better scalability.

The ideal structure of this source code in order to ease the integration with CUDA would be a structure, where all the information needed for the extrapolation of a new track state would be encapsulated into one structure. Then if the Kalman filters and smoothers technique need only the previous state in order to predict the next state, it would be ideal if the source code would follow the same rationale. However, we should keep in mind that this source code is a part of a larger framework and this study is only done on a standalone version. Working on the framework will introduce even
greater complexity to the source code. These issues can be addressed if the memory is partitioned in a way that can be ported on the device, asynchronous CUDA streaming and multiple devices are used.
Chapter 7

Future Work and Conclusion

7.1 Future Work

The C implementation of Kalman Fitter can be fully integrated with CUDA, as the refactored source code is modified according to this approach. This integration will provide results justifying that an considerable acceleration is possible utilizing the general approach of executing a candidate track fit per thread. An extension to this work will be to measure the speedup reached by distributing the tracks on multiple GPU devices. The results of this approach can aid in deducing conclusions for the scalability of the CUDA implementation relating candidate tracks for fitting and devises used.

In the longterm, when the full C++ CUDA release is available, this approach and implementation can be used to produce another version of the integrated with CUDA Kalman Fitter. The latter, will produce interesting results regarding memory management, as the two ways memory address space allocation, “malloc” C operator and “new” C++ operator can be compared in view of the memory management techniques used, how much address space the program needs in those two cases and the reusability of these two implementations.

The CUDA Kalman Filter code can be integrated to the ATLAS framework in order to examine the results of exploiting GPU devices in the ATLAS High Level Trigger, and gain some acceleration results of the overall ATLAS level 2 trigger. The implementation of the C Kalman Fitter can be potentially provided as a library in the framework source code. The on-going research in the ATLAS and generally in the High Energy Physics community on GPU utilization for acceleration of these algorithms and routines
will bring to light more approaches and implementations of CUDA that could produce an overall estimate of the ATLAS High Level Trigger utilizing GPU devices in each step of the pipeline. This could lead to more tangible results and possibly to a significant acceleration. This approach and implementation could be also utilized as a basis or even a reusable component for other algorithms used for track reconstruction in the ATLAS project.

7.2 Conclusion

The task of accelerating the ATLAS software trigger has many extensions and potential, giving rise to a long-term research, which is combined with the on-going research for an upgrade of the LHC. A potential upgrade, with a scope of having a degree of magnitude higher luminosity, may result in an increase from 25 proton-proton collisions to 400, multiplying the occurring events into a relative scale and consequently the data and the rates of their processing will increase correspondingly. Challenging applications such as the ATLAS trigger have to explore acceleration options in order to cope with such an increase. One of the alternatives explored is the utilization of GPU devices, in a heterogeneous programming paradigm combining GPU and CPU architecture, that could boost the heavy-weight computational demands of the routines of the ATLAS trigger.

This feasibility study is concerned with the alternative of utilizing the vast capabilities that GPUs exhibit in executing high density computations. As far as the Kalman Fitter is concerned, this project proved that it is not an easy task to integrate such a routine with CUDA, at least on a two-month timetable. Re-engineering of the application with the aim of CUDA integration, is a task that might need to enforce changes that go back to the design phase of the project lifecycle. Re-modelling may be a necessity in order to make any kind of acceleration feasible, as memory management is vital for such kind of endeavours and it completely changes scope on a parallel paradigm let alone when the execution is passed from CPU to GPU devices. The changes in the related work for the SIMDized version of a kalman fitting routine for the CBM experiment [33] were not only on the basis of the design model of the source code but went beyond that, to the mathematical tools and representations of the environment of the detector. Firstly, a uniform magnetic field approximation was used instead of the previous non-homogenous representation of the magnetic field and some mathematical
instability that was caused in type conversion to float instead of double was taken into consideration in order to alter the routine. The parallelized in SIMD fashion routine of the kalman fitting technique of the CBM experiment was the starting point for further acceleration using GPUs and CUDA instead of Cell processors [25], which encapsulated the changes mentioned above. All the above show that the re-engineering part of the integration with CUDA is a strenuous part of the integration. This part can be eased if CUDA offers full support of C++ and CUDA becomes easier to integrate and debug.

The programmability of GPUs is increasing at a high pace, especially after the recent innovations on hardware level, opening new horizons in CUDA implementation of scientific computing applications such as these. The potential of utilizing GPUs, is based on both the computational capabilities of the GPUs and for their programmability level. These two factors are the ones that play a crucial role of selecting this alternative of the various perspectives for acceleration beyond CPU farms.

The ATLAS High Level Trigger can be potentially accelerated to a considerable point, if we take into account the results of the on-going research in the community of High Energy Physics. Of course, the results of High Level Trigger algorithms will give more insight and justification of the role that GPU’s can play in the ATLAS project. This however, is an open field of research that started recently, studies of each of the algorithms and progressively of the whole High Level Trigger will give sufficient results. The characteristics of the applications’ environment, the demands for high throughput and immense computational resources can justify this research and the perspective of a significant role of GPU computing in the context of the ATLAS experiment and more generally in the High Energy Physics community.
Appendix A

Technical Characteristics of the GPUs

The CUDA enabled GPU device utilized in this project is part of the Edinburgh Compute and Data Facility (ECDF) and is of compute capability 2.0. The following are its detailed technical characteristics:

Device 0: "GeForce GTX 470"
CUDA Driver Version: 3.10
CUDA Runtime Version: 3.10
CUDA Capability Major revision number: 2
CUDA Capability Minor revision number: 0
Total amount of global memory: 1341849600 bytes
Number of multiprocessors: 14
Number of cores: 448
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per block: 1024
Maximum sizes of each dimension of a block: 1024 x 1024 x 64
Maximum sizes of each dimension of a grid: 65535 x 65535 x 1
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Clock rate: 1.21 GHz
Concurrent copy and execution: Yes
Run time limit on kernels: No
Integrated: No
Support host page-locked memory mapping: Yes
Compute mode: Default (multiple host threads can use this device simultaneously)

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 134560347, CUDA Runtime Version = 3.10, NumDevs = 2, Device = GeForce GTX 470, Device = Quadro NVS 290
Bibliography


