



Departamento de Ingeniería de Computadores

Facultad de Informática de A Coruña

UNIVERSIDADE DA CORUÑA

MASTER'S THESIS

INTERUNIVERSITY MASTER'S DEGREE IN

HIGH PERFORMANCE COMPUTING

Parametrized Kalman filter for downstream tracks

Student: Alejandro Perez Casas

Directors: Diego Martínez Santos
Jorge González Domínguez
Veronika Georgieva Chobanova
Arthur Hennequin

Zumaia, September 6, 2025.

Acknowledgments

I would like to thank Jorge, Diego, and Veronika for giving me the opportunity to be part of this project and for their assistance throughout its development. I am also very grateful to Arthur and Lennart Uecker, who patiently, eagerly, and expertly taught me everything I know about LHCb and Allen. Their professionalism made this project possible. Last but not least, I want to thank my family, my little sister Laia, and my partner Bianca for their constant encouragement and support.

Abstract

Este Trabajo de Fin de Máster presenta el desarrollo necesario para implementar un filtro de Kalman parametrizado destinado a la reconstrucción de trayectorias downstream en el High-Level Trigger 1 (HLT1) del experimento LHCb, implementado en GPUs dentro del framework Allen. El objetivo principal fue dividir el kernel existente del filtro de Kalman en subkernels modulares, permitiendo su reorganización para extender el filtrado de Kalman a las trayectorias downstream. Las trayectorias downstream son trayectorias de partículas que se originan fuera del Localizador de Vértices (VELO). Esta mejora busca aumentar la eficiencia en la selección de eventos para la desintegración de partículas con mayor esperanza de vida.

Un objetivo secundario fue la exploración del filtro de raíz cuadrada de Carlson como una alternativa numéricamente más robusta al filtro de Kalman tradicional, con el fin de abordar posibles problemas de inestabilidad derivados de la aritmética de precisión simple. Los resultados iniciales muestran mejoras prometedoras en la estabilidad numérica, aunque se requiere un análisis más profundo para resolver anomalías en pasos específicos de la predicción.

Además, se identificaron y corrigieron errores críticos en las funciones de similitud del filtro de Kalman parametrizado, lo que condujo a mejoras medibles en la resolución del momento. También se modernizaron estructuras de código obsoletas, reduciendo la deuda técnica y mejorando la mantenibilidad.

Abstract

This master's thesis presents the developments necessary for a parametrized Kalman filter for downstream track reconstruction in the LHCb experiment's High-Level Trigger 1 (HLT1) system, implemented on GPUs within the Allen framework. The primary objective was to split the existing Kalman filter kernel into modular sub-kernels, allowing their reorganization to extend Kalman filtering to downstream tracks. Downstream tracks are particle trajectories originating outside the Vertex Locator (VELO). This enhancement aims to improve the efficiency of event selection for long-lived particle decays.

A secondary focus was the exploration of Carlson's square-root filter as a more numerically robust alternative to the traditional Kalman filter, addressing potential instability issues arising from single-precision arithmetic. Initial results indicate promising improvements in numerical stability, although more research is required to resolve anomalies in specific prediction steps.

Additionally, critical bugs in the parametrized Kalman filter similarity functions were identified and fixed, leading to measurable gains in momentum resolution. Deprecated code structures were also modernized, reducing technical debt and improving maintainability.

Palabras clave:

- Computación de Altas Prestaciones
- GPU
- CUDA
- Física de Partículas
- Filtro Kalman
- LHCb
- Trayectorias Downstream
- Estabilidad Numérica

Keywords:

- High Performance Computing
- GPU
- CUDA
- Particle Physics
- Kalman filter
- LHCb
- Downstream Tracks
- Numerical stability

Contents

1	Introduction	1
1.1	Methodology	2
2	Theoretical Background	3
2.1	Detector	5
2.2	Kalman filter	7
2.3	Objectives	9
3	Main developments	11
3.1	Kalman filter split	11
3.1.1	Fit kernel split	11
3.1.2	Adding Downstream Kalman filter	13
3.1.3	Declaring an algorithm	15
3.1.4	Validating the algorithm	18
3.2	Carlson's sqrt filter	19
3.2.1	New filter formulation	19
3.2.2	Implementation	20
4	Other optimizations	25
4.1	Original code: kalman_filer kernel	25
4.2	Bugs in ParKF	27
4.3	Deprecated code	29
5	Results	35
5.1	Running conditions	35
5.2	Downstream Kalman filter	36
5.3	Carlson's square-Root Filter	38
5.4	Similarity Bug Fix	43

6	Conclusions	45
6.1	Personal conclusions	45
7	Future work	47
8	Glossary of Terms	51
	Bibliography	53

List of Figures

2.1	LHCb detector's diagram	4
2.2	LCHb diagram with a focus on VELO, UT, MAGNET and SciFi	5
2.3	VELO Top view including tracks	6
2.4	Track types visualized. TT=UT; T stations=SciFi. Taken from https://lhcb.github.io/glossary/glossary/T.html#Track	7
5.1	first_qop for downstream Kalman filter showing the initial qop of downstream tracks	36
5.2	best_qop for downstream Kalman filter showing the final qop estimate of downstream tracks	37
5.3	All UpdateState kernels in Carlson's sqrt filter formulation	39
5.4	PredictState functions replaced by Carlson formulation, one by one	39
5.5	UpdateState functions plus PredictState functions replaced by Carlson formulation, one by one	40
5.6	Having only one PredictState in carlson versus having one plus all the previous ones. From top to bottom, from left to right: PredictStateVUT, PredictStateUT, PredictStateUTT, PredictStateT	41
5.7	Comparing the effects of storing S throughout the code versus conversion in every step	42
5.8	Various cases showcasing how PredictStateTFT in Carlson implementation severely degrades quality	42
5.9	Effects in momentum resolution quality of the two bug fixes	43

List of Tables

3.1	Line rate change as a consequence of splitting the fit function	13
5.1	hlt1_pp_forward_then_matching_with_parkf Performance	44

Introduction

CERN, The European Organisation for Nuclear Research, is an intergovernmental body founded in 1954 that runs the world's largest particle physics laboratory. Located in Meyrin, near Geneva on the France–Switzerland border. This organisation has made multiple discoveries (the birth of the world-wide web in 1989[1], arguably the most famous Higgs boson discovery 13 years ago[2], or the more recent new measurements of the Z boson's mass[3]). In addition to contributions to the standard model and particle physics, CERN's Knowledge Transfer Group brings innovation to other fields, such as healthcare, computing, environmental applications, aerospace, quantum technologies, and more[4].

At the core of the CERN experimental programme lies the challenge of handling enormous volumes of data generated by the detectors of the Large Hadron Collider (LHC). After the first upgrade of the LHCb detector, one of the four major experiments, its sub-detectors produce a total of 40 Tbit/s output data. Storing all this data for further offline analysis is not feasible both for technical and economical reasons. A real-time analysis system is necessary to select only the most interesting events.

Detecting which events are worth keeping is not a trivial task, moreover, keeping in mind the high volume of data. Several algorithms contribute to this process, including decoding, tracking, and clustering. Among these, the Kalman filter helps to improve particle trajectory estimates.

The goal of this master thesis is to split the current Kalman filter GPU kernel into smaller sub-kernels, in such a way that allows the reorganisation of said sub-kernels to enable Kalman filtering on particle trajectories other than long tracks. By doing so, we aim to improve event selection for particles that have this trajectory.

Given the high-performance focus of this project and the collaboration with numerous experts from across disciplines, the development process faced several challenges. Understanding the basics of fundamental physics and familiarising with the detector and its configuration took a meaningful amount of time, as well as becoming familiar with the Allen

software framework and the underlying architecture. Additionally, on large-scale projects with as many collaborators as this, developments must follow a strict methodology: submit merge requests, present changes to developers, physicists, and maintainers, and pass throughput and efficiency pipeline tests before integration into the framework.

1.1 Methodology

The initial definition of this project was established around mid-January, marking the starting point of this master thesis. Shortly thereafter, at the end of January, I attended LHCb's computing workshop in A Coruña, where I had the opportunity to meet my supervisors and the people who helped me during the thesis: Diego, Jorge, Veronika, Arthur, and Lennart. There I could start to familiarise myself with both the LHCb detector and the software infrastructure. During this event, a one-month long on-site development stay at CERN was scheduled, in Geneva, to allow for deeper understanding and accelerating development.

Throughout the month of February, I dedicated time to gaining a more comprehensive understanding of the fundamental concepts behind particle physics (an area of academic interest for this thesis but also of big personal interest), as well as diving into the theoretical and practical aspects of Kalman filtering and the computational architecture employed by CERN. This preparatory phase was valuable for more effective participation during my time at CERN.

In March, I carried out the on-site development work, which included a visit to the LHCb detector itself and a close interaction with Arthur, which allowed me to accelerate learning and development. These experiences showed me some of the specific challenges faced in the Allen framework, and helped shape the concrete development goals of this thesis. From that point onward, work has continued remotely, focusing on three distinct areas, which will be detailed in the following chapters.

Theoretical Background

CERN was first proposed by a handful of European and North American scientists with a twofold goal: to mitigate the migration of scientific talent to the USA, and to promote unity across Europe after WW2. Among the first experiments built we can mention the Synchrocyclotron¹ (CERN's first accelerator, which provided beams for CERN's first experiments in particle and nuclear physics) and the Proton Synchrotron² (proton accelerator, still working today feeding particles to other accelerators). Today, the world's largest and most potent particle accelerator is the Large Hadron Collider (LHC³). This particle accelerator consists of a 27km ring buried 100m underground, and speeds up ions and protons to near the speed of light in opposing directions in two separate ultrahigh vacuum tubes that cross at certain points along the ring to make particle beams collide.

The Large Hadron Collider beauty[5] (LHCb) experiment is one of the four main detectors at the LHC, dedicated to exploring the subtle differences between matter and antimatter by studying the 'beauty quark' (or b quark). To capture the forward-going particles produced in each collision, Figure 2.1 shows how LHCb uses a series of sub-detectors arranged along a 20-meter length: the first is mounted very close to the interaction point, with the others aligned one after another.

Physicists perform the most precise analysis possible offline, but the huge data volume generated by LHCb's sub-detectors makes it economically and technically impossible to store and distribute every event. To address this, a fully software-based real-time analysis system selects only the most interesting events as they occur. Note that each run differs not only in collision type, but also in the small calibration shifts of the detector (sometimes intentional, sometimes caused by heating during operation), so alignment and calibration data are impor-

¹The 600-MeV Synchrocyclotron - Last checked: 20-07-2025 <https://home.cern/science/accelerators/synchrocyclotron>

²The Proton Synchrotron - Last checked: 20-07-2025 <https://home.cern/science/accelerators/proton-synchrotron>

³The Large Hadron Collider (LHC) - Last checked: 20-07-2025 <https://home.cern/science/accelerators/large-hadron-collider>

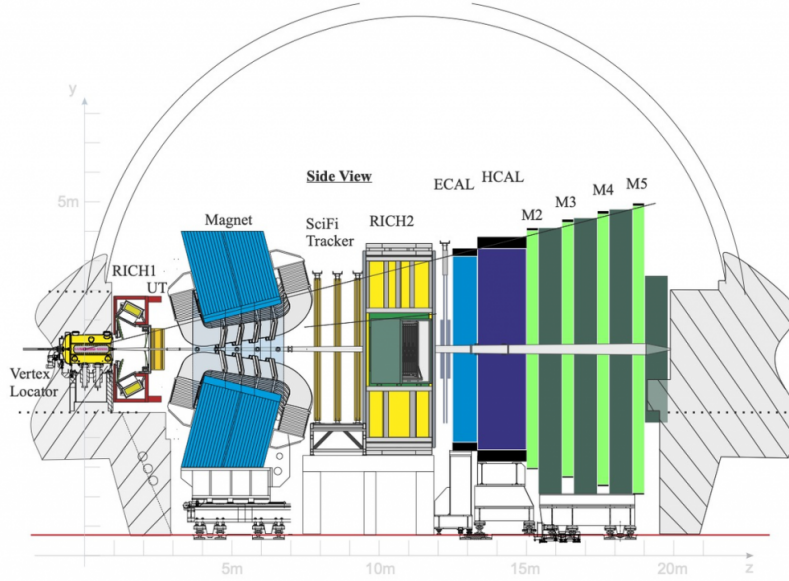


Figure 2.1: LHCb detector's diagram

tant.

This system, known as the High-Level Trigger (HLT), is divided into two stages. In HLT1, the full collision rate of 30 MHz (around 40 Tbit/s) is subjected to rapid event selection using calibration and alignment constants carried over from previous runs, reducing the data rate by a factor of 30–60. The selected events are stored in a temporary buffer, where near-real-time alignment and calibration are performed.

The next step, HLT2, involves offline quality track reconstruction, full particle identification, and precise track fitting. Here, the events are further reduced, both in quantity and in size, storing only the objects related to specific decays of interest rather than whole events. The bandwidth is thus further reduced to about 80 Gbit/s for later offline analysis.

Allen[6], HLT1 implementation on GPUs[7], is able to process the full collision rate with around 500 off-the-shelf GPUs mounted on the same nodes that are in charge of event building. Implemented in CUDA, Nvidia's GPU programming API, Allen has its own custom implementation for the scheduler and memory manager[8].

Each sub-detector registers 'hits', points where particles have passed by the sub-detector, plus some additional information. Track reconstruction is the process of identifying which series of hits belong to the same particle, as to determine its trajectory (track) and which collision point (called primary vertex) each particle belongs to[8]. By identifying all particles coming from a collision and using mass, energy, momentum, and other conservation principles, physicists are able to identify particle decays.

One of the algorithms used by Allen is the Kalman filter. By using a series of measurements obtained over time (hits, in this case) together with statistical predictions, the filter produces estimates of unknown variables that are generally more accurate than estimates based on only one measurement. This helps improve the track resolution, thus also improving decay detection efficiencies.

2.1 Detector

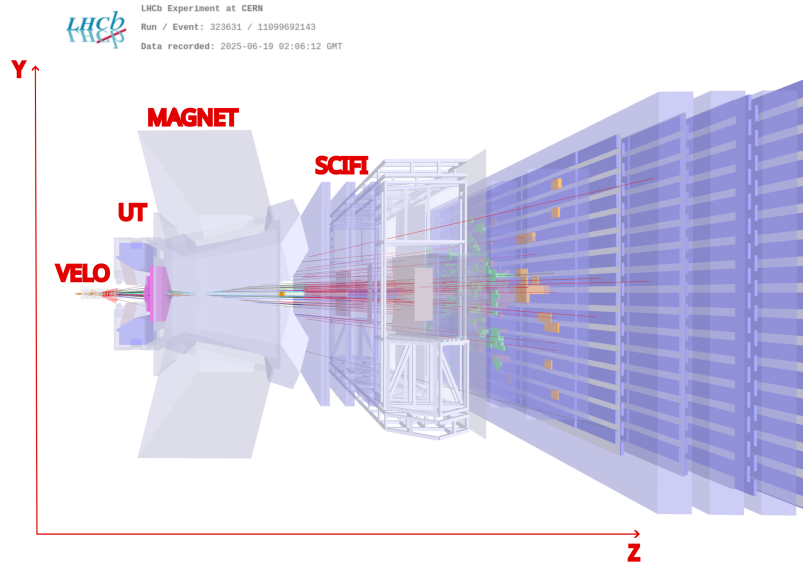


Figure 2.2: LCHb diagram with a focus on VELO, UT, MAGNET and SciFi

As seen in Figure 2.2, the LHCb detector consists of various sub-detectors. The most relevant ones for this work include the Vertex Locator (VELO), the Upstream Tracker (UT), and the Scintillating Fibers (SciFi), as well as the magnet, which is not really a sub-detector but it is relevant for this project, too.

The VELO[9] shown in Figure 2.3 surrounds the beam-crossing region, where two proton beams intersect and spawn a plethora of new particles. To pinpoint both the collision origins (primary vertices) and subsequent decay points (secondary vertices) with micrometer precision, VELO is built from 52 retractable silicon pixel modules, each module arranged in two halves around the beam. When closed around the beam, the innermost pixels sit within 5.1 mm of the beam axis. Each module has $55 \times 55 \mu\text{m}^2$ pixels, for a grand total of 41 million pixels. The spacing between each module gradually increases as the distance to the interaction point increases. Particle tracks in this stage are almost straight lines, and hits are grouped into pos-

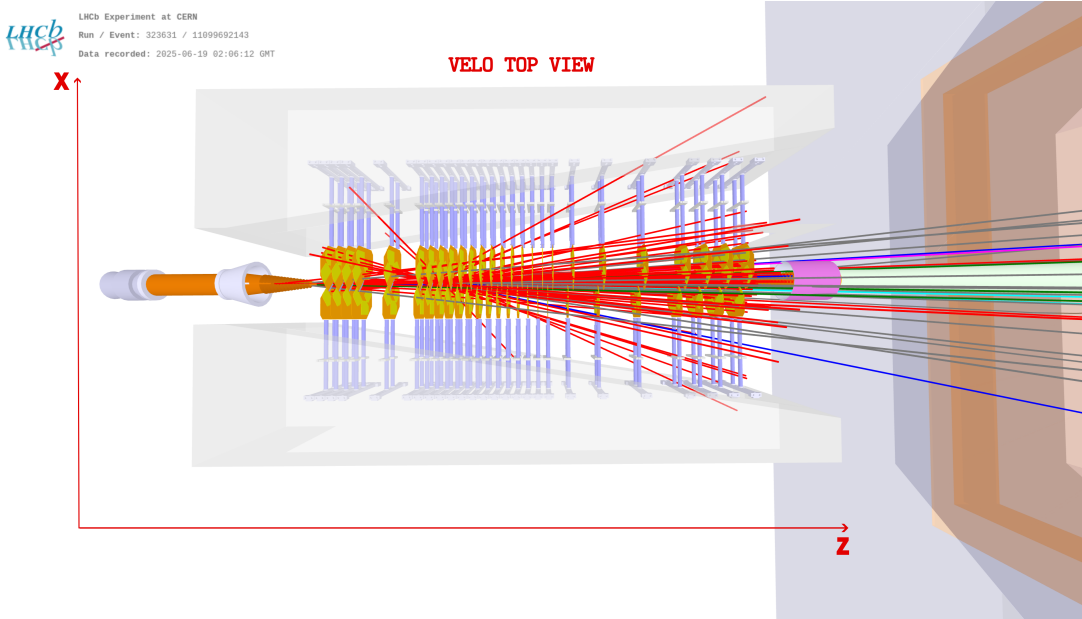


Figure 2.3: VELO Top view including tracks

sible tracks, also called track candidates. These contain both real and false trajectories, which can emerge as combinations of hits from other tracks.

The UT[10] consists of four plane stations of silicon strip sensors. It refines track candidates from the VELO, reducing false or ghost tracks and providing the positional information needed for precise momentum resolution before the bend of charged particles in the magnet. As having the same pixel resolution of VELO across the four panes would be too expensive, the strips have a micrometer resolution across the X-axis, and a centimeter resolution across the Y-axis. Note that the trajectory bend produced by the magnet only affects the Y-axis, and the second and third panes are horizontally sheared to the left and right, respectively, so they are not rectangles, but rather parallelograms. This allows for better Y-axis precision of consecutive hits by overlapping the strips.

The LHCb dipole magnet[11] spans approximately 10 m along the beamline with a 1.1 m vertical gap between its pole faces. Weighing 1600 tons, it generates a horizontal magnetic field of about 1T, with an integrated bending power of roughly 4Tm. By deflecting charged particles as they traverse the field, the dipole magnet enables precise momentum determination when combined with upstream and downstream tracking.

The SciFi[12] is located downstream of the dipole magnet. Built from three stations, each made up of four detection layers, it delivers sub-100 μm spatial resolution across a 5 m length, reconstructing charged-particle trajectories with high efficiency and completing the momentum measurement initiated by the UT and VELO.

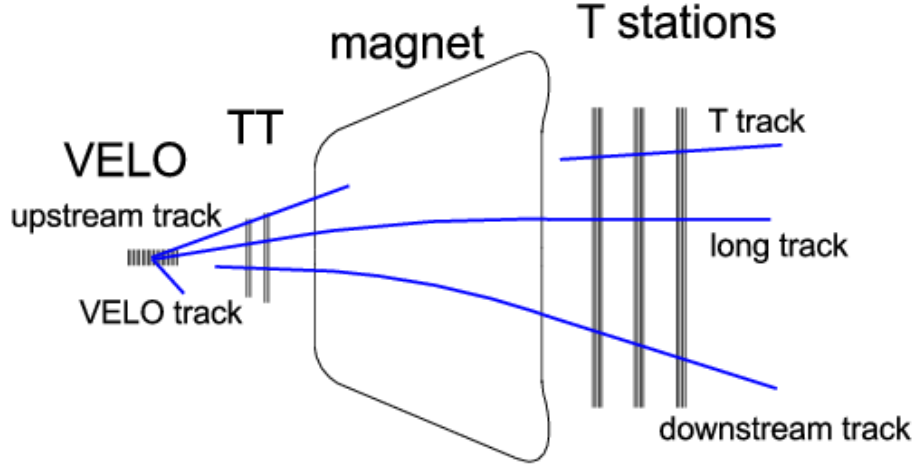


Figure 2.4: Track types visualized. TT=UT; T stations=SciFi. Taken from <https://lhcb.github.io/glossary/glossary/T.html#Track>

Figure 2.4 illustrates how a track is identified differently depending on the subsetectors used to reconstruct it:

- **Long tracks** are formed from hits in Velo, UT and SciFi stations. These have the best momentum and spatial resolution.
- **Downstream tracks** are formed from hits only in the UT and the SciFi stations. These tracks begin downstream of the VELO as they represent decay products from long-lived particles that happen outside the VELO.
- **Other track types** that include Upstream tracks (VELO+UT, very low-momentum particles), Velo tracks (VELO only) and T tracks (SciFi only).

2.2 Kalman filter

When filtering the reconstructed tracks to discard false or ghost candidates, Kalman filtering is used to achieve optimal precision. For this, a very precise 'complete' method that required the use of maps of the magnetic field around the magnet (measured on site[13]) and Runge-Kutta integration to propagate particles through the non-uniform field[14], was employed. This method was very precise, but very slow and memory consuming, so its use on GPUs was not viable given the high throughput requirements. Reserving this complete filtering for HLT2 and offline analysis, Kalman filtering on GPU was restricted to the VELO section, as no magnetic field is present there.

To overcome this problem, parametrized Kalman filter[15] was introduced to deliver accurate track parameter estimates without the need for maps or compute-heavy integration methods. Basically, the look-up of the maps and integration is done offline, and solutions are parametrized. Initially developed for a fast CPU Kalman filter, and later implemented on GPU, it provided much better track fitting efficiency with an affordable or justifiable throughput decrease versus the Velo only Kalman filter. Currently, Kalman filtering is performed only for long tracks in Allen. This project starts from Lennart Uecker's merge request: 'Reviving the parametrized Kalman filter'⁴.

The trajectory of a particle is only measured at discrete points along the beam axis (Z coordinate), where it interacts with various sub-detectors. Each track is represented as an ordered sequence of states, each associated with a particular Z-position. Every state is a 5 component vector:

$$X = (x, y, t_x, t_y, q/p)$$

Where at the given z:

- x and y are transverse coordinates
- $t_x = \frac{dx}{dz}$ and $t_y = \frac{dy}{dz}$ are the slopes
- q/p is the charge over momentum

A 5x5 covariance matrix C is kept in memory throughout the Kalman filtering process, that indicates the level of uncertainty of each element in the state vector against itself and other elements.

The Kalman filter is divided into two steps: predicting and updating the state. Predicting means using specific formulations to estimate the next state from a previous one. A transport matrix F_k is computed from equations of motion, and a noise matrix Q_k is also computed alongside. The transport matrix indicates how much the state changed, and the noise matrix contains the extrapolation error. The covariance matrix is updated using both the transport and noise matrices:

$$X_{k|k-1} = F_k X_{k-1}$$

$X_{k|k-1}$ represents the predicted state, computed from the previous one X_{k-1} and the transport matrix F_k .

$$C_{k|k-1} = F_k C_{k-1} F_k^T + Q_k$$

The predicted covariance matrix $C_{k|k-1}$ is updated using similarity, bringing the C matrix to

⁴Reviving the parametrized Kalman filter (merged) - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1693

F 's geometry, and adding noise.

During the `update` stage, measured values from the sub-detectors are used to filter (or more precisely, correct) the predicted state. At plane k , the Kalman gain value is computed:

$$K_k = C_{k|k-1} H_k^T (H_k C_{k|k-1} H_k^T + R_k)^{-1}$$

Where:

- H_k is a projection matrix, because measurements are usually 2D, and the state vector is 5D. This 'brings' the state into measurement space.
- R_k is measurement error.
- K_k is the kalman gain, balancing how much "trust" the prediction versus the new measurement has.

Both the predicted state and the covariance matrix are updated again using the kalman gain:

$$X_k = X_{k|k-1} + K_k(m_k - H_k X_{k|k-1})$$

$$C_k = (I - K_k H_k) C_{k|k-1}$$

Kalman filtering is implemented by looping both of these steps, predicting, updating and predicting.

2.3 Objectives

The work presented in this master's thesis is centered on the splitting of the current Kalman filter kernel and the structured arrangement of its sub-kernels, with the goal of enabling a fully parametrized Kalman filter for downstream tracks. The subsequent integration and performance evaluation are addressed to assess the implications of this kernel restructuring.

Other objectives include exploring Carlson's square-root filter on GPUs, which is inherently more numerically robust. Numerical stability is a concern given that HLT2 and offline analysis employ double precision floating point numbers (doubles), while in Allen single precision (floats) is used, although for testing purposes, doubles can be used too.

Lastly, the use of half precision (FP16) was presented as an optionally additional objective, but due to other optimisations done in the code and lack of time, it was postponed for future work.

Together, these objectives will enable to improve track fitting for downstream tracks, improving track reconstruction and physics efficiency measurements.

Main developments

This chapter will focus on the development of the main and secondary objectives of the project: splitting the parameterized Kalman filter kernel to implement a downstream track Kalman filter and exploring the use of the Carlson square root filter for numerical stability.

3.1 Kalman filter split

As mentioned in Section 2.2, Lennart's merge request focused on adapting a previous implementation of the parametrized Kalman filter to the current master branch and its event model at the time. The event model is the collection of data structures that algorithms use as input or output. They describe all the necessary objects used by the algorithms, such as long tracks or KalmanStates.

3.1.1 Fit kernel split

The original fit function can be seen in gitlab¹. Listing 3.1 shows the pseudocode for the fit kernel. It gets information about hits in the different sub-detectors, and then performs the Kalman filter loop described in Chapter 2 for each sub-detector. The characteristics of the prediction and update steps vary depending on the sub-detector and whether the step involves the transition from one sub-detector to another. The loop is done first from the closest point to the collision to the furthest point. This is called `forward pass`. Then, a `backward pass` is done, which helps improve accuracy, as a single pass may carry accumulated error, and performing a fresh pass in the opposite direction balances this. The backward pass is limited to Velo, as doing it on the whole detector would increase the cost, and also because VELO has the highest resolution throughout all three axes and is closest to the state we want to

¹Reviving the parametrized Kalman filter (merged), fit function - Last checked: 20-07-2025 <https://gitlab.cern.ch/lhcb/Allen/-/blob/2593f7f821fb180438c1998f87c06cb7a3711fa6/device/kalman/ParKalman/src/ParKalmanFilter.cu#L124>

measure: closest to beam line.

Listing 3.1: fit kernel pseudocode

```

1  __device__ void fit(
2  const Allen::Views::Velo::Consolidated::Track& velo_track,
3  const Allen::Views::UT::Consolidated::Track& ut_track,
4  const Allen::Views::SciFi::Consolidated::Track& scifi_track,
5  const KalmanFloat init_qop,
6  const KalmanParametrizations* kalman_params,
7  FittedTrack& track,
8  const float* various_parametrization_arrays)
9  {
10     get_velo_hits(...);
11     get_ut_hits(...);
12     get_scifi_hits(...);
13     initialize_forward_pass(...);
14     loop_velo(...);
15     loop_ut(...);
16     loop_scifi(...);
17     initialize_backward_pass_velo(...);
18     loop_velo_backward(...);
19     make_track(...);
20 }
```

The kernel was split² into the following sub-kernels, broadly following the structure outlined in the pseudocode. An initial splitting strategy was proposed prior to the project, but the precise way the kernel was partitioned was decided during the project.

- **forward_velo_fit** -> Retrieves Velo hits, initializes the forward pass, and enters the Velo fitting loop.
- **get_ut_hitmap** -> This kernel retrieves UT hits and calculates the UT hitmap, which encodes the index of the first hit in each layer. The hitmap is required when processing UT data. An alternative implementation using byte manipulation intrinsics in CUDA was briefly explored with the aim of saving one or two registers per thread. However, this approach introduced additional computational complexity and would have required debugging and benchmarking to assess its effectiveness. Given time constraints and the relatively marginal potential gain, this optimisation was not pursued further. It could be one of the optimisations proposed for future work.
- **forward_VUT_fit** -> Applies the transition step from VELO to UT. This kernel could

²Fit function split into multiple device kernels - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1919/diffs?commit_id=6af1083aa3fc432364ea998c08926f611231319c

be appended to `forward_velo_fit`, but it would result in a bigger kernel, which was one of the reasons behind splitting the original fit kernel.

- **forward_ut_fit** -> Executes the UT fitting loop over all UT hits.
- **forward_scifi_fit** -> Transitions from UT to SciFi, then runs the SciFi fitting loop.
- **initialize_backward_velo_fit** -> Sets up data structures and state for the backward VELO pass.
- **backward_velo_fit** -> Performs the backward Velo fitting loop.

It was paramount to check whether the split caused differences in execution, as even minor inaccuracies in intermediate calculations can propagate and significantly impact the final results. As seen in Table 3.1, there was a small change in line selection.

Line	rate before the split	rate after the split
Hlt1DiElectronLowMass_NoIPNorm:	16000/8000000	0/8000000

Table 3.1: Line rate change as a consequence of splitting the fit function

Note that for testing, usually a high GPU occupancy is needed to measure throughput more precisely. To achieve that with small samples, the test is repeated 1000 times on 16 CUDA streams, so the change seen on 3.1 is actually $1/500$. To find the source of this change, we tried defining `KalmanFloat` as a double, which is not viable during operation because the NVIDIA RTX A5000 GPUs used, natively support doubles but only with 1/64th the TFLOP rate of FP32 operations³, which makes execution really slow. However, for testing whether the source of small value changes is due to numerical precision issues, it is really handy. Using doubles, both versions produced the same results.

It was empirically found that the CUDA compiler has a pretty aggressive approach to optimisation. It uses function inlining, loop unrolling, dead code elimination, constant propagation, code reordering, etc. Since Allen uses floats, and given the high-precision nature of Kalman filter, this small difference when splitting the fit kernel is attributable to numerical rounding errors with different orders of instructions and is not impactful for data taking.

3.1.2 Adding Downstream Kalman filter

The new algorithm is based on the Kalman filter for long tracks. For this, the fit starts from a seed state (in UT, rather than in VELO for long tracks), performs the UT loop, and continues with the loop over SciFi. Then, for the backward pass, it inverts the transport matrix, it

³Text below Figure 2 - Last checked: 20-07-2025 <https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf>

transforms the covariance matrix, and it starts at the end of UT (on a state saved during the forward pass) towards the start of it.

The necessary changes can be seen on a commit in the merge request (MR) ⁴. The CreateUTSeedState kernel is responsible for the initial state, which can be simplified in Listing 3.2. The seed state must contain, as mentioned in Chapter 2, X and Y position and slope (T_x and T_y), and momentum information. In CreateUTSeedState, those are defined as follows:

- X and Y coordinates are taken from the center of the first hit UT strip.
- T_x is the mean of the first and last UT strips hit.
- T_y is the mean of the first hit UT strip in the middle and the point of origin $(0, 0, 0)$.
- Initial momentum information is not given here; it comes from the tracks when they are created.
- The covariance matrix C is initialized to large uncertainties and no correlation.
- The transport matrix is initialized as an identity matrix.

Listing 3.2: Simplified CreateUTSeedState kernel

```

1  __device__ inline void CreateUTSeedState(...) {
2  // Getting the Y position of first and last hit in UT
3  // UT strips have poor resolution on Y-axis (~cm), so the middle point of each strip is considered
4  const auto y0 = ((track.hit(first_hit).yBegin() + track.hit(first_hit).yEnd()) * 0.5);
5  const auto y1 = ((track.hit(last_hit).yBegin() + track.hit(last_hit).yEnd()) * 0.5);
6
7  // Getting the X position of first and last hit in UT
8  const auto x0 = track.hit(first_hit).xAtYEq0() + track.hit(first_hit).dxDy() * y0;
9  const auto x1 = track.hit(last_hit).xAtYEq0() + track.hit(last_hit).dxDy() * y1;
10 x(0) = (KalmanFloat) x0; // X position
11 x(1) = (KalmanFloat) y0; // Y position
12 x(2) = (KalmanFloat)
13 ((x1-x0) / (track.hit(last_hit).zAtYEq0() - track.hit(first_hit).zAtYEq0())); // Tx slope
14 x(3) = (KalmanFloat)
15 (y0 / track.hit(first_hit).zAtYEq0()); // Ty slope
16
17 // Set covariance matrix with large uncertainties and no correlations.
18 C(0, 0) = (KalmanFloat) 100.0;
19 C(0, 1) = (KalmanFloat) 0.0;
20 C(0, 2) = (KalmanFloat) 0.0;
21 ...

```

⁴the kalman filter for downstream tracks - Last checked: 20-07-2025 gitlab.cern.ch/lhcb/Allen/-/merge_requests/1919/diffs?commit_id=c4061d350c8e22ae64bdb63febf48b1132a2f19f

```

22
23 // Transport matrix F
24 F.SetElements(F_diag);
25 }

```

The forward pass is a combination of the right sub-kernels. For the backward pass, the state at the end of UT must be saved during the forward pass. For that, an additional vector was added to the `TrackInfo` object: `m_RefUTStateForwardV`.

3.1.3 Declaring an algorithm

Every function to be executed on the GPU sequence must be declared as an algorithm. For that, `ParKalmanFilter.cuh` declares a `parameters` struct, indicating the type for each parameter of the algorithm (e.g. `unsigned`, `bool`, ...). It also declares whether is for host or device and if it is an input, output or output with dependencies. On Listing 3.3 we can see a couple of examples.

Listing 3.3: `kalman_filter_downstream parameters` struct examples

```

1 struct Parameters {
2     // MACRO(type_alias, underlying_type_or_view) member_name;
3     HOST_INPUT(host_number_of_events_t, unsigned) host_number_of_events;
4     DEVICE_INPUT(dev_multi_event_downstream_tracks_view_t, Allen::Views::Physics::
5         ↪ MultiEventDownstreamTracks) dev_multi_event_downstream_tracks_view;
6     DEVICE_OUTPUT(dev_kalman_fit_results_t, char) dev_kalman_fit_results;
7     DEVICE_OUTPUT_WITH_DEPENDENCIES(
8         dev_kalman_states_view_t,
9         DEPENDENCIES(dev_kalman_fit_results_t),
10        Allen::Views::Physics::KalmanStates)
11    dev_kalman_states_view;
12 };

```

Later on `ParKalmanFilter.cu`, we must declare a function for setting the size of the output arguments. In this case, the Kalman filter is a transformation algorithm that does not modify the dimensions of input data. In other words, for each input track, it delivers fitting information, but does not select or discard any tracks. As such, `set_arguments_size` function is defined in Listing 3.4.

Listing 3.4: `kalman_filter_downstream set_arguments_size` function

```

1 void kalman_filter_downstream::kalman_filter_downstream_t::set_arguments_size(
2     ArgumentReferences<Parameters> arguments,
3     const RuntimeOptions&,
4     const Constants&) const
5 {

```

```

6 auto n_tracks = first<host_number_of_reconstructed_tracks_t>(arguments);
7 set_size<dev_kf_tracks_t>(arguments, n_tracks);
8 set_size<dev_kalman_fit_results_t>(arguments, n_tracks * Allen::Views::Physics::KalmanState::
    ↪ struct_size);
9 set_size<dev_kalman_states_view_t>(arguments, first<host_number_of_events_t>(arguments)
    ↪ );
10 }

```

Lastly, an operator function is declared. As seen in Listing 3.5, this function sets the number of blocks and threads per block to be launched. Currently, this is a hard-coded value of 128 threads per block, calling enough blocks to assign one long track per thread. After `kalman_filter`, `kalman_pv_ip` is called. This matches tracks to primary vertices (points of origin of new resulting particles from a collision). However, it does not make much sense to match primary vertices with downstream tracks, and since this function consolidated the results for other algorithms, an additional kernel is necessary. For this, one block is launched per event (combination of collisions in a given instant), with 128 threads per block. This block size has been tuned using simulated data, keeping in mind that it must be a multiple of the warp size (32), and having a close number of threads to the number of tracks per event.

Listing 3.5: `kalman_filter_downstream` operator function

```

1 void kalman_filter_downstream::kalman_filter_downstream_t::operator()(
2     const ArgumentReferences<Parameters>& arguments,
3     const RuntimeOptions&,
4     const Constants& constants,
5     const Allen::Context& context) const
6 {
7     dim3 block_dim = m_block_dim;
8     int _gridDim = (first<host_number_of_reconstructed_tracks_t>(arguments) + (block_dim.x) -
    ↪ 1) / (block_dim.x);
9     global_function(kalman_filter_downstream)(dim3(_gridDim), m_block_dim, context)(
10         arguments, constants.dev_magnet_polarity.data(), constants.dev_kalman_params);
11
12     // Since no PVIP is called for downstream track fitting, we need to consolidate the kalman tracks (
    ↪ adding offset and event number) separately
13     // Launch one thread per event for this
14     const unsigned n_events = first<host_number_of_events_t>(arguments);
15     _gridDim = (n_events + (block_dim.x) - 1) / (block_dim.x);
16     global_function(kalman_filter_downstream::consolidate_kalman_tracks)(dim3(_gridDim),
    ↪ m_block_dim, context)(arguments, n_events);
17 }

```

There is also an update function needed for copying parameterisation data to GPUs constant memory. Finally, the declaration of the `kalman_filter_downstream_t` algorithm can be constructed as in Listing 3.6.

Listing 3.6: `kalman_filter_downstream_t` algorithm declaration

```
1 INSTANTIATE_ALGORITHM(kalman_filter_downstream::kalman_filter_downstream_t)
2 struct kalman_filter_downstream_t : public DeviceAlgorithm, Parameters {
3     void update(const Constants& constants) const;
4     void set_arguments_size(ArgumentReferences<Parameters> arguments, const RuntimeOptions&,
5         ↪ const Constants&) const;
6
7     void operator()(
8         const ArgumentReferences<Parameters>& arguments,
9         const RuntimeOptions& runtime_options,
10        const Constants& constants,
11        const Allen::Context& context) const;
12 private:
13     Allen::Property<dim3> m_block_dim {this, "block_dim", {128, 1, 1}, "block dimensions"};
14 };
```

Lastly, the algorithm must be added to the sequence of executed algorithms. Allen has multiple Python configuration files, the most relevant for this algorithm being `hlt1_reconstruction.py`. In this file, we define where in the sequence and under which conditions it must be launched. Initially, if the configuration includes UT, downstream and fullKF, the algorithm was executed after `make_downstream`, the group of algorithms that created downstream tracks, as shown in Listing 3.7.

Listing 3.7: `hlt1_reconstruction` section involving downstream KF

```
1 if with_ut and enableDownstream:
2     # Downstream tracking
3     downstream_tracks = make_downstream(
4         decoded_ut=long_tracks["ut_hits"],
5         scifi_seeds=long_tracks["seeding_tracks"],
6         velo_scifi_matches=long_tracks['matched_tracks'],
7         pvs=pvs,
8         dev_used_ut_hits_offsets=long_tracks[
9             "dev_used_ut_hits_offsets"])
10    output.update({"downstream_tracks": downstream_tracks})
11
12    if with_fullKF:
13        kalman_downstream_tracks = make_kalman_downstream(downstream_tracks)
14        output.update({"kalman_downstream_tracks": kalman_downstream_tracks})
```

Listing 3.8: Python function `make_kalman_downstream`

```
1 def make_kalman_downstream(downstream_tracks):
2     number_of_events = initialize_number_of_events()
3
```

```

4 | kalman = make_algorithm(
5 |     kalman_filter_downstream_t,
6 |     name='kalman_downstream_{hash}',
7 |     host_number_of_events_t=number_of_events["host_number_of_events"],
8 |     dev_number_of_events_t=number_of_events["dev_number_of_events"],
9 |     host_number_of_reconstructed_tracks_t=downstream_tracks[
10 |         "host_number_of_downstream_tracks"],
11 |     dev_multi_event_downstream_tracks_view_t=downstream_tracks["
12 |         ↪ dev_multi_event_downstream_tracks_view"],
13 |     dev_offsets_downstream_tracks_t=downstream_tracks["dev_offsets_downstream_tracks"],
14 | )
15 | return {
16 |     "downstream_tracks": downstream_tracks,
17 |     "dev_kf_tracks": kalman.dev_kf_tracks_t,
18 |     "dev_kalman_fit_results": kalman.dev_kalman_fit_results_t,
19 |     "dev_kalman_states_view": kalman.dev_kalman_states_view_t
20 | }

```

`make_kalman_downstream` is shown in Listing 3.8, the Python function responsible for invoking the `kalman_filter_downstream_t` algorithm and binding its inputs and outputs to other algorithms.

3.1.4 Validating the algorithm

At this point, no other algorithms use `kalman_filter_downstream_t`'s outputs, so the algorithm scheduler removes it from the sequence. To account for this, we added a validator⁵ that compares the reconstructed tracks and their estimated parameters to the Monte-Carlo simulation's ground truth. By doing so, not only do we give use to the output, but we also give a convenient and fast method to check how future modifications (as changing the `createUTSeedState`, for example) could affect measurements.

The file containing the tree is in ROOT format, created at CERN. This is a binary, hierarchical data format designed for efficient storage and retrieval of large-scale scientific data[16]. Developed as part of the ROOT framework, it enables fast I/O and compact storage through techniques like compression and columnar storage. ROOT files (.root) contain objects such as histograms, trees (TTrees), and graphs, which are organized in a directory-like structure. TTrees, a core feature, allow structured storage of large datasets with millions of entries, enabling efficient querying and analysis without loading the entire dataset into memory. The output is later analyzed in Chapter 5.

⁵Added `downstream_kalman_filter` algorithm and validator to the sequence - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1919/diffs?commit_id=378c0fa48e4fb1bf6e24ae5611db06cb3b198f5b

3.2 Carlson's sqrt filter

The Kalman filter is a fast and efficient method for improving track estimates. However, it is crucial that the covariance matrix remains non-negative along its diagonal; otherwise, NaN values may appear. While we did not observe this issue during normal execution, certain bugs, such as the one discussed later in Section 4.2, along with numerical instability caused by large-scale differences in the X and Y-axis error margins in UT and SciFi, or the use of single-precision arithmetic, can lead to small negative values on the diagonal. These errors can subsequently propagate and result in NaN values.

Carlson's square-root filter[17] is a reformulation of the Kalman filter that propagates the Cholesky factor[18] S_k of the covariance rather than the covariance C_k itself. By maintaining $C_k = S_k * S_k^T$, all covariance matrices remain implicitly positive, eliminating the risk of negative variances along the diagonal and the subsequent NaNs.

3.2.1 New filter formulation

As in the classical Kalman filter, we compute the transport matrix F_k and process-noise matrix covariance Q_k . With this we have $W_k = F_k S_{k-1}$, so that the predicted state $X_{k|k-1}$ and sqrt-covariance S can be computed like this:

$$X_{k|k-1} = F_k X_{k-1}$$

$$S_{k|k-1} = \text{cholesky}(W_k W_k^T + Q_k) = \text{predictRSS}(W_k, Q_k)$$

RSS stands for Root-Sum-Square operation. For the filtering step, given a measurement m_k , projection matrix H_k and measurement error R_k , we can define $f_k = S_{k|k-1} H_k^T$ and $S_{zz} = R_k + f_k^T f_k$. Then we can compute the gain factor and triangular update by first defining the A_k upper triangular matrix, along B :

$$A_k = \text{cholesky}(I - f_k S_{zz}^{-1} f_k^T)$$

$$B = A_k A_k^T$$

However, elements of A and B can be generated sequentially by a sequence of partial sums to avoid that much computation. Being $S_{k|k-1}^{(i)}$ the i th column of $S_{k|k-1}$, and the resulting columns of S_k by b_i , then for $i = 1, \dots, 5$:

$$b_i = \frac{1}{A_{ii}} \left(S_{k|k-1}^{(i)} - \sum_{j=1}^{i-1} A_{ji} \right)$$

In particular, the last column b_5

$$b_5 = \frac{S_{k|k-1} f_k}{S_{zz}} = K_k$$

is the Kalman gain, necessary to filter the state vector X_k :

$$X_k = X_{k|k-1} + K_k(m_k - H_k X_{k|k-1})$$

3.2.2 Implementation

Starting from a CPU implementation by Arthur⁶ (inspired by [17] and existing implementations in Java⁷ and Matlab⁸), the functions that needed to be implemented in GPU were Cholesky decomposition, carlson filter and a prediction function which (either PredictHouseholder or predictRSS). I choose predictRSS for its simplicity and slightly better computation efficiency[17]. It requires 4 lines of code with matrix multiplication, summation, and Cholesky decomposition, the code being understandable at first sight. The merge request related to this implementation⁹ explores how this alternative filter can be included in Allen.

Cholesky decomposition is a method that factors any symmetric positive matrix A as $A = LL^T$ (with L being a triangular matrix). It can be understood as a 'square root' of A , and cuts the factorization cost to $n^3/3$ flops for a $n * n$ square matrix. It not only uses roughly half of the flops as in the LU decomposition, but also half of the storage. The GPU Cholesky decomposition is shown in Listing 3.9. Although several possible implementations of this decomposition are available online, the one used was part of Arthur's Carlson square root CPU filter. For now, as to being able to use the defined operations upon matrices, the triangular matrix is defined as a square matrix. To save memory, a new datatype should be implemented, only storing the $\frac{n*(n+1)}{2}$ elements, but for testing purposes this is enough.

Listing 3.9: Cholesky GPU kernel

```

1 typedef SquareMatrix<false, 5> TriMat5x5;
2
3 __device__ inline TriMat5x5 cholesky (const SymMatrix5x5& L_in)
4 {
5     TriMat5x5 L_out;
```

⁶CPU sqrt filter implementation draft - Last checked: 20-07-2025 <https://gitlab.cern.ch/lhcb/Rec/-/commit/b0bb42ae32a0f5680f7ce1c372ae64961e5a9747>

⁷Java implementation of Carlson's sqrt filter - Last checked: 20-07-2025 <https://github.com/alexEvangelidis/VerFilter/blob/master/models.kalman.filter/SchmidtCarlsonFilterPropC.java>

⁸Matlab implementation of Carlson's sqrt filter - Last checked: 20-07-2025 https://staff.ulsu.ru/semushin/_index/_pilocus/_gist/docs/mycourseware/13-stochmod/6-tools/matlab-codes-grewalandrews/CHAPTER6/CARLSON.M

⁹Adding Carlson's sqrt filter - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/2108

```

6
7   for (unsigned j = 0; j < 5; j++)
8   {
9       KalmanFloat acc = 0.;
10      for (unsigned k = 0; k < j; k++) acc += L_out(j,k) * L_out(j,k);
11
12      KalmanFloat s = L_in(j,j)-acc;
13      const KalmanFloat sqrt_s = sqrt(s);
14      L_out(j,j) = sqrt_s;
15
16      for (unsigned i = j+1; i<5; i++)
17      {
18          acc = 0.;
19          for (unsigned k = 0; k < j; k++) acc += L_out(i,k) * L_out(j,k);
20          KalmanFloat s = L_in(i,j)-acc;
21          L_out(i,j) = s/sqrt_s;
22      }
23  }
24  return L_out;
25  }

```

With this, the prediction and update (filter) steps of the filter can be implemented as in the Listings 3.10 and 3.11. The filter step is different from Carlson’s paper[17], the loop iterates from $i = (n-1)..0$ instead of $i = 0..(n-1)$. Arthur has found this bug in Carlson’s paper and in the Java and Matlab implementations mentioned before. After running some tests on CPU and GPU, we verified that using this loop order produced equal results to Kalman filtering on CPU and near equal on GPU (attributable to the use of single precision).

Listing 3.10: predictRSS GPU kernel

```

1  __device__ inline void predictRSS(const Matrix5x5& F, const SymMatrix5x5& Q, TriMat5x5& S)
2  {
3      auto W = F * S.T();
4      SymMatrix5x5 WWt;
5      WWt = W * W.T();
6      S = cholesky( WWt + Q);
7  }

```

Listing 3.11: carlsonFilter GPU kernel

```

1  __device__ inline Vector5 carlsonFilter(const Vector5& H, KalmanFloat& R, TriMat5x5& S)
2  {
3      Vector5 K;
4
5      KalmanFloat a_ = R;
6      for (int i = 4; i>= 0; i--) {

```

```

7      KalmanFloat f = 0.;
8      for (int j = i; j < 5; j++) {
9          f += H(j) * S(j,i);
10     }
11
12     KalmanFloat a = a_ + f*f;
13     KalmanFloat b = sqrt(a_/a);
14     KalmanFloat v = f / (b*a);
15
16     K(i) = 0.;
17     for (int j = i; j < 5; j++) {
18         KalmanFloat S_ = S(j,i);
19         S(j,i) = b * S_ - v * K(j);
20         K(j) = K(j) + f * S_;
21     }
22     a_ = a;
23 }
24
25 K = K * (KalmanFloat(1.) / a_);
26 return K;
27 }

```

These functions would replace the old Kalman steps in `ParKalmanMethods.cuh`¹⁰. As seen on Listing 3.12, the similarity and `AeApB` lines for filtering can be replaced by `predictRSS`; and the Kalman gain vector can be calculated using `predictCarlson`.

Listing 3.12: `carlsonFilter` GPU kernel

```

1  __device__ inline void PredictStateVUT(...)
2  {
3      ...
4      { // Kalman predict
5          // Transport the covariance matrix
6          C = similarity_5_5_VUT(F, C);
7
8          // Add noise.
9          // C = C + Q;
10         AeApB(C, Q);
11     }
12
13     { // Carlson predict
14         // Transport the covariance matrix and add noise
15         predictRSS(F, Q, S);
16     }

```

¹⁰ `device/kalman/ParKalman/include/ParKalmanMethods.cuh` - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/2108/diffs#fde1e90afc823585038e2d3a243579e39959143a

```
17  ...
18  }
19
20  __device__ inline void UpdateStateUT(...)
21  {
22      ...
23      { // Kalman update
24          //  $K = P * H$ 
25          Vector5 K;
26          multiply_S5x5_2x1(C, H, K);
27
28          //  $K * S^{-1}$ 
29          K = K / CRes;
30          x = x + res * K;
31
32          //  $K * S * K^T$ 
33          SymMatrix5x5 KCResKt;
34          tensorProduct(sqrtf(CRes) * K, sqrtf(CRes) * K, KCResKt);
35
36          //  $P -= KSK^T$ 
37          //  $C = C - KCResKt$ ;
38          AeAmB(C, KCResKt);
39      }
40
41      { // Carlson update
42          Vector5 K = carlsonFilter(H, err2, S);
43          x = x + res * K;
44      }
45
46      // Update chi2.
47      tI.m_chi2UT += res * res / CRes;
48      ...
49  }
```


Other optimizations

This chapter will cover other changes in Allen that are not related to the main and secondary objectives of the project discussed in Chapter 3. The need for these changes rose during development, so a short explanation will be given explaining the cause.

4.1 Original code: kalman_filer kernel

The `kalman_filter` originally performed a reverse search, where, starting from the thread id, it would find the event to which that track belonged. This was important because later, when the result was set in the output buffer, the offset for that event was added to the offset of the track to maintain the ordering from the input buffers. If no event or track was found, it would return. Note that extra preprocessor directives are added because Allen can run CUDA code on the CPU, as Allen is run on CERN's LHC Computing Grid[19] as part of the simulation production process[20], and computers on the grid only have CPUs.

However, `kalman_filter` is a transformation algorithm that does not modify the dimensions of input data. In other words, it delivers fitting information for each input track, but does not select or discard any tracks. Thus, this reverse search was not needed. An alternative was proposed¹. Here, the base pointer for the first track is obtained, and all threads set their result using absolute offsets relative to the first track. The preprocessor directives were also removed, making the code simpler and more readable. Listing 4.1 shows the previous code section, and Listing 4.2 shows the proposed alternative.

Listing 4.1: `kalman_filter` event_number reverse search

```

1 | #ifndef TARGET_DEVICE_CPU // IF GPU: Find the event number matching the position in the
   | ↪ grid
2 | const unsigned track_id = blockIdx.x * blockDim.x + threadIdx.x; // global now, reduce later

```

¹Event_list alternative - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1910/diffs?commit_id=c52aa1320905f184254f73e0ade0c0f4915f630c

```

3 unsigned event_number = UINT_MAX;
4 unsigned n_tracks = 0;
5 unsigned final_track_id = UINT_MAX;
6 // TODO: This implementation does not assume anything about the ordering of events, check
    ↪ which assumptions could be
7 // made.
8 for (unsigned event_id = 0; event_id < parameters.dev_number_of_events[0]; ++event_id) {
9     unsigned current_event_number = parameters.dev_event_list[event_id];
10    unsigned n_long_tracks = parameters.dev_long_tracks_view->container(current_event_number
    ↪ ).size();
11    int condition = (track_id >= n_tracks && track_id < n_tracks + n_long_tracks);
12    final_track_id = condition * (track_id - n_tracks) + (1 - condition) * final_track_id;
13    event_number = condition * current_event_number + (1 - condition) * event_number;
14    n_tracks += n_long_tracks;
15 }
16
17 if (event_number == UINT_MAX) {
18     return;
19 }
20 if (final_track_id == UINT_MAX) {
21     return;
22 }
23 #else
24 // CPU implementation as many blocks as events
25 unsigned event_number = parameters.dev_event_list[blockIdx.x];
26 #endif

```

Listing 4.2: kalman_filter event_number reverse search alternative

```

1 // Base pointer for the list of all tracks (contiguous in memory)
2 // This way we avoid the need to get the track_id for any event, just process them in a loop
3 const Allen::Views::Physics::LongTrack* track_base = &(parameters.dev_long_tracks_view->
    ↪ container(0).track(0));
4 const unsigned total_number_of_tracks = parameters.dev_long_tracks_view->
    ↪ number_of_contained_objects();
5
6 Velo::Consolidated::States kalman_states {parameters.dev_kalman_fit_results,
    ↪ total_number_of_tracks};

```

The code is greatly reduced and is much more readable and maintainable. There were some issues regarding event_lists, as Arthur expressed in a Gitlab issue², and as a solution for one of the points, the above proposal was mentioned. However, as my MR was taking long

²Ideas for EventList rework - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/issues/581#note_9187145

and a bug related to this was found by Da Yu Tou³, it was merged with a comment by Arthur relating the fix to the proposed alternative above⁴.

The rest of the kernel iterates the tracks, preparing the data for the `fit` kernel, which performed the filter per se.

4.2 Bugs in ParKF

During the development of the downstream Kalman filter, initial tests showed that during the transition states between the UT and SciFi, the covariance matrix C had values so large that during the first steps of the SciFi loop it would cause floating point absorption, resulting in negative values on the diagonal. By definition, the Kalman filter should not have negative values in the diagonal, and this caused the filter to break and produce NaN values.

The magnet bends charged particles between the UT and SciFi, and the parameterizations used to extrapolate particles through the magnet are very sensitive to starting states. Another aspect to keep in mind is that VELO, with 52 modules and pixel precision, allows for relatively low uncertainties at the last state. Combining these two ideas, we tried changing the UT seed state to have a state that resembles more what VELO would produce, but the issue persisted, both using floats and doubles.

Another theory was that since precision across the X-axis and the Y-axis is significantly different in magnitude, operations between these values gave way to numerical absorption, causing negative values in the diagonal. We took this fact as hint that Kalman filter was not numerically stable enough for downstream tracks, further proving the need for a more numerically robust alternative: Carlson square-root filter.

We started with comparisons of double-precision CPU implementations for the three different filtering options: Carlson, CPU version of Kalman and GPU version of Kalman. Specifically, we tried the prediction and update steps with the states that caused NaN values to see if implementing the new algorithm would indeed avoid this. During these tests, we saw that while the CPU Carlson and Kalman produced near identical results and positive covariances, the GPU Kalman version produced highly different values and NaN values.

We decided to recreate and check step by step all the computation steps that happened on GPU, executing it on CPU⁵, and found a bug in the current parametrized Kalman filter implementation. During the prediction step of the filter, the covariance matrix is transported using the similarity function: $similarity(F, C) = F * C * F^T$, a method to bring C to

³Fix Segfault in Kalman filter - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1958

⁴Context on the seg fault and the fix - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1958#note_9338700

⁵kalman/Carlson CPU/GPU comparison in CPU - Last checked: 20-07-2025 <https://godbolt.org/z/8aarrPbPq>

F 's space. The similarity function is defined in two ways, as seen on Listing 4.3: one by using math operations upon matrices (similarity_5_5_alt), and one explicitly writing all the operations (similarity_5_5). By leveraging the first option to the compiler, it should unroll all operations and optimise it the same as having all operations explicitly, but some tests showed that this produced different results from the explicit version when using floats, but the same results when using doubles. In fact, values from the explicit version with floats were closer to the 'correct' values when using doubles. As throughput did not change when one or another was used, the implicit version was removed altogether.

Listing 4.3: similarity_5_5 implicit and explicit versions

```

1  __device__ __host__ SquareMatrix<true, 5> similarity_5_5_alt(
2      const SquareMatrix<false, 5>& F,
3      const SquareMatrix<true, 5>& C)
4  {
5      SquareMatrix<false, 5> tmp = F * C;
6      return AssignSymmetric(tmp * F.T());
7  }
8
9  __device__ __host__ SquareMatrix<true, 5> similarity_5_5(
10     const SquareMatrix<false, 5>& F,
11     const SquareMatrix<true, 5>& C)
12 {
13     SquareMatrix<true, 5> A;
14     KalmanFloat* a = A.vals;
15     const KalmanFloat* f = F.vals;
16     const KalmanFloat* c = C.vals;
17     KalmanFloat _0 = c[0] * f[0] + c[1] * f[1] + c[3] * f[2] + c[6] * f[3] + c[10] * f[4];
18     KalmanFloat _1 = c[1] * f[0] + c[2] * f[1] + c[4] * f[2] + c[7] * f[3] + c[11] * f[4];
19     KalmanFloat _2 = c[3] * f[0] + c[4] * f[1] + c[5] * f[2] + c[8] * f[3] + c[12] * f[4];
20     KalmanFloat _3 = c[6] * f[0] + c[7] * f[1] + c[8] * f[2] + c[9] * f[3] + c[13] * f[4];
21     KalmanFloat _4 = c[10] * f[0] + c[11] * f[1] + c[12] * f[2] + c[13] * f[3] + c[14] * f[4];
22     ...
23     return A;
24 }
25
26 // Spezialized versions (simplified)
27 __device__ __host__ inline SquareMatrix<true, 5> similarity_5_5_VP(F,C);
28 __device__ __host__ inline SquareMatrix<true, 5> similarity_5_5_VUT(F,C);
29 __device__ __host__ inline SquareMatrix<true, 5> similarity_5_5_UT(F,C);
30 __device__ __host__ inline SquareMatrix<true, 5> similarity_5_5_UTT(F,C);
31 __device__ __host__ inline SquareMatrix<true, 5> similarity_5_5_T(F,C);
32 __device__ __host__ inline SquareMatrix<true, 5> similarity_5_5_TFT(F,C);

```

For the parametrized Kalman filter, at specific points of the detector, certain properties of

the covariance matrix can be assumed, such as some values being 1 and others 0. This gave way to implementing 'specialized' similarity_5_5 functions, as in Listing 4.3⁶, which explicitly described the similarity operation, but removing multiple sums and multiplications with the mentioned assumptions, for the sake of higher throughput (around 1-2% higher throughput on the Kalman filter). However, similarity_5_5_T was missing a value when calculating `a[1], '_1'` needed to be added, but it was not. This changed the results.

During the development of this thesis, LHC started to work again after its annual winter stop. All changes to the code had to be done before the restart, as any changes later would need to wait until specific dates for patches. However, some discussion took place in the MR I opened for fixing the bug⁷, regarding the use of momentum for the prediction step in VELO, regarding the best solution to the bug (fixing it or avoiding specialized versions of the function) and regarding whether to inline the extrapolate steps inside the prediction ones. As with the event_list solution proposed before 4.1, a quick fix of the function was merged to master by Lennart to start data-taking bug-free⁸

Fixing the bug also solved the problem of the NaN values in the covariance matrix, and since data-taking started with the bug fixed, the MR did not have such a high priority. However, of all the specialized_5_5 functions, the only one that was different was similarity_5_5_V. This function used a 2×2 F matrix instead of a 5×5 one, and the reason was that only three values were distinct to 1 or 0, so the matrix was being used as a buffer. However, when mapping the buffer positions to their respective positions in the matrix, another bug was found⁹. The MR lastly replaced all the specialized similarity functions with the general one, to avoid other possible bugs that would be really difficult to find, as were the first two. As a consequence, the codebase was reduced greatly, which is a plus by itself, and also improved readability and maintenance.

4.3 Deprecated code

The data structure used for the Kalman states is `Allen::Views::Physics::KalmanStates`, but the original Kalman filter algorithm used another one: `Velo::Consolidated::States`. This is because `KalmanStates` are constant and during the Kalman filter these states need to be modified with the computed states. As a workaround, `Velo::Consolidated::States` were added with the

⁶similarity_5_5_VUT - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/blob/master/device/event_model/kalman/include/ParKalmanMath.cuh#L714

⁷Using general similarity_5_5 instead of specialized ones - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1941

⁸`a[1] = _1` - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/commit/ea4868a2e011e0c87007da4374940cd6848a#e1987c4af6c1b952ffa6e5893850d0ee7de3f855_845_845

⁹similarity_5_5_VP issue - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1941#note_9521105

same structure but without the const identifier. That structure is deprecated¹⁰ and should be replaced by the mentioned Kalman states¹¹. Moreover, the deprecated struct is used in other parts of the code, so I made changes to the KalmanStates to be able to use them in a non-const way.

Listing 4.4: Simplified version of original KalmanStates

```

1 namespace Allen {
2   namespace Views {
3     namespace Physics {
4       struct KalmanState {
5         private:
6           const float* m_base_pointer = nullptr;
7           unsigned m_index = 0;
8           unsigned m_total_number_of_tracks = 0;
9
10        public:
11          __host__ __device__
12          KalmanState(const char* base_pointer, const unsigned index, const unsigned total_number_of_tracks) :
13            m_base_pointer(reinterpret_cast<const float*>(base_pointer)),
14            m_index(index), m_total_number_of_tracks(total_number_of_tracks) {}
15
16          __host__ __device__ float x() const { return m_base_pointer[nb_elements_state * m_index]; }
17          __host__ __device__ float y() const { return m_base_pointer[nb_elements_state * m_index + 1]; }
18          ...
19        };
20
21        struct KalmanStates {
22          private:
23            const char* m_base_pointer = nullptr;
24            unsigned m_offset = 0;
25            unsigned m_size = 0;
26            unsigned m_total_number_of_tracks = 0;
27
28          public:
29            __host__ __device__ KalmanStates(
30              const char* base_pointer,
31              const unsigned* offset_tracks,
32              const unsigned event_number,
33              const unsigned number_of_events) :
34              m_base_pointer(base_pointer),
35              m_offset(offset_tracks[event_number]), m_size(offset_tracks[event_number + 1] - offset_tracks[

```

¹⁰Deprecated - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/blob/master/device/event_model/velo/include/VeloConsolidated.cuh#L311

¹¹Use the same states for Kalman filter - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/blob/master/device/event_model/velo/include/VeloConsolidated.cuh#L385

```

36     ↪ event_number]),
37     m_total_number_of_tracks(offset_tracks[number_of_events]) {}
38     ...
39     __host__ __device__ KalmanState state(const unsigned track_index) const
40     {
41         assert(track_index < m_size);
42         return KalmanState {m_base_pointer, m_offset + track_index, m_total_number_of_tracks};
43     };

```

Listing 4.4 shows a simplified version of how `Allen::Views::Physics::KalmanStates` was defined before¹², and Listing 4.5 shows how by using C++’s templates we can use the same structure instead of needing a deprecated workaround. I also added another constructor for `KalmanStates`, since in other parts of the code the event number was unknown, and the deprecated states used a second constructor for that too. With these changes, non-const access had to be explicit with `Allen::Views::Physics::KalmanStates_t<char>&`, and const access was kept the same, avoiding major refactors throughout the codebase.

Listing 4.5: Simplified version of modified `KalmanStates` for const and non-const access

```

1 namespace Allen {
2     namespace Views {
3         namespace Physics {
4             // char or const char
5             template<typename Char_type>
6             struct KalmanState_t {
7             private:
8                 Char_type* m_base_pointer = nullptr;
9                 unsigned m_index = 0;
10                unsigned m_total_number_of_tracks = 0;
11
12                // If Char_type is const char, use const float*, else, use float*. This allows to have const or non-const
13                ↪ version, avoiding the use of deprecated states
14                using FloatPtr = std::conditional_t<std::is_const_v<Char_type>, const float*, float*>;
15
16                __host__ __device__ FloatPtr base() const {return reinterpret_cast<FloatPtr>(m_base_pointer);}
17
18                template<bool B, class T = void>
19                using enable_if_t = typename std::enable_if<B, T>::type;
20
21            public:
22                __host__ __device__
23                KalmanState_t(Char_type* base_pointer, const unsigned index, const unsigned total_number_of_tracks) :

```

¹²struct `KalmanState` - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/blob/master/device/event_model/common/include/States.cuh?ref_type=heads#L130

```

23     m_base_pointer(base_pointer), m_index(index), m_total_number_of_tracks(total_number_of_tracks)
24 {}
25
26     __host__ __device__ float x() const { return base()[nb_elements_state * m_index + 0]; }
27
28     template<typename C = Char_type, enable_if_t<!std::is_const_v<C>, int> = 0>
29     __host__ __device__ float& x() { return base()[nb_elements_state * m_index + 0]; }
30
31     ...
32 };
33
34     template<typename Char_type>
35     struct KalmanStates_t {
36     private:
37         Char_type* m_base_pointer = nullptr;
38         unsigned m_offset = 0;
39         unsigned m_size = 0;
40         unsigned m_total_number_of_tracks = 0;
41
42     public:
43         __host__ __device__ KalmanStates_t(
44             Char_type* base_pointer,
45             const unsigned* offset_tracks,
46             const unsigned event_number,
47             const unsigned number_of_events) :
48             m_base_pointer(base_pointer),
49             m_offset(offset_tracks[event_number]), m_size(offset_tracks[event_number + 1] - offset_tracks[
50 ↪ event_number]),
51             m_total_number_of_tracks(offset_tracks[number_of_events])
52 {}
53
54         __host__ __device__ KalmanStates_t(
55             Char_type* base_pointer,
56             const unsigned total_number_of_tracks,
57             const unsigned offset = 0) :
58             m_base_pointer(base_pointer),
59             m_offset(offset), m_size(total_number_of_tracks),
60             m_total_number_of_tracks(total_number_of_tracks)
61 {}
62
63         __host__ __device__ KalmanState_t<Char_type> state(const unsigned track_index) const
64         {
65             assert(track_index < m_size);
66             return KalmanState_t<Char_type> {m_base_pointer, m_offset + track_index,
67 ↪ m_total_number_of_tracks};
68         }

```

```
67     };  
68  
69     // By default the access to them is constant like before  
70     // must be explicit when wanting the non-const version  
71     using KalmanState = KalmanState_t<const char>;  
72     using KalmanStates = KalmanStates_t<const char>;
```


5.1 Running conditions

The CERN DIRAC (Distributed Infrastructure with Remote Agent Control) portal provides a unified web-based gateway for distributing raw and derived run data across the Worldwide LHC Computing Grid. It manages reliable file replication and transfer by coordinating services and agents that monitor workloads, handle errors, and distribution policies. Through the portal, all RAW files are first archived at CERN and then automatically replicated to Tier-1 centers[21]. The user interface on the DIRAC web allows users to access the data¹.

In DIRAC, 'DIGI' files contain the digitized detector output-raw banks of sub-detector hits (e.g., VELO pixels, UT and SciFi strips, etc). These files, staged through DIRAC's data management², serve as the input to generate the 'MDF' (Moore Data Format) files required for standalone Allen processing. The provided `mdf_for_standalone_Allen.py` script³ in the Moore framework reads in DIGI files (along with certain tags) and produces two products: a binary .mdf file containing the concatenated raw banks event-by-event, and a companion geometry directory holding the serialized geometry needed by Allen's HLT1 algorithms. Since standalone Allen cannot query the LHCb conditions database, this binary geometry ensures that the correct detector description is embedded directly alongside the MDF data.

¹DIRAC web portal, CERN account is necessary - Last checked: 20-07-2025 <https://lhcb-portal-dirac.cern.ch/DIRAC/>

²Downloading a file from the grid - Last checked: 20-07-2025 <https://lhcb.github.io/starterkit-lessons/first-analysis-steps/files-from-grid.html>

³Produce MDF files for standalone Allen - Last checked: 20-07-2025 https://allen-doc.docs.cern.ch/setup/input_files.html#produce-mdf-files-for-standalone-allen

5.2 Downstream Kalman filter

As mentioned in subsection 3.1.4, the validator records different metrics of the downstream Kalman filter's output. Using the online ROOT file reader⁴, we can check those outputs. In Allen, qop refers to the ratio of the charge of a particle over its momentum (q/p). This parameter is critical, as the curvature of a charged particle's trajectory in the magnet is directly proportional to this ratio, and is used to accurately predict how much a particle will bend as it traverses the magnetic field in the dipole magnet, which affects the extrapolation of its path to downstream detectors. Figure 5.1 shows the initial qop for downstream tracks, that is, the qop they have before passing the filter, and Figure 5.2 shows the best qop estimate after the filter. The difference is very subtle, increasing both the mean and standard deviation slightly. It is very difficult to estimate the effects of the filter from these figures alone, so the next step would be to integrate the filter's output with other algorithms and explore how line selection changes.

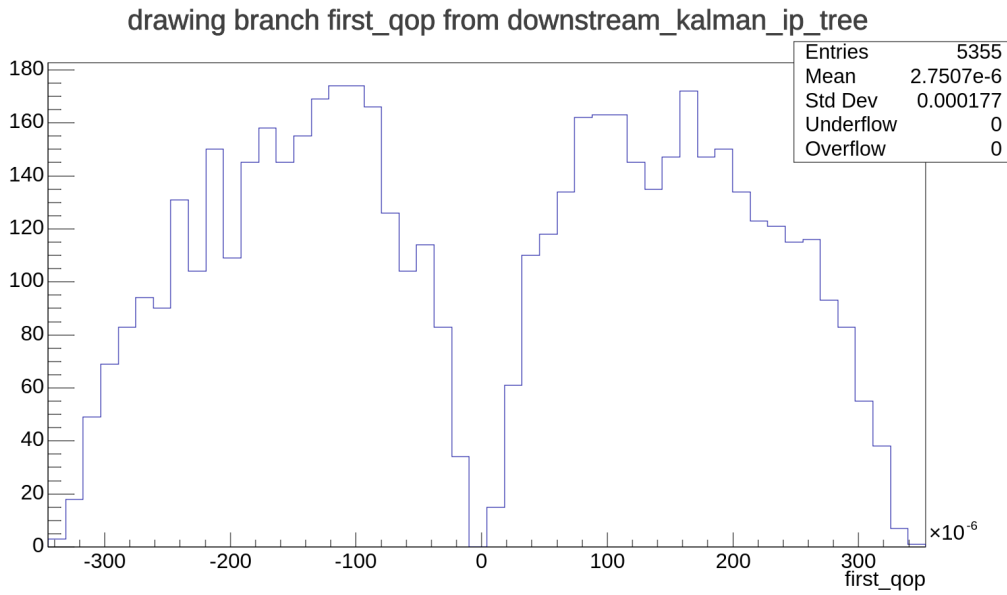


Figure 5.1: first_qop for downstream Kalman filter showing the initial qop of downstream tracks

In the case of long tracks, reconstruction algorithms generally process the full track information directly; while downstream algorithms usually work with particles. These 'particles' are simplified physics objects that contain basic properties without requiring a full reconstruction of the particle's origin or complete trajectory history. For downstream algorithms, working directly with particles rather than full tracks allows for faster and more efficient

⁴Read a ROOT file - Last checked: 20-07-2025 <https://root.cern.ch/js/latest/>

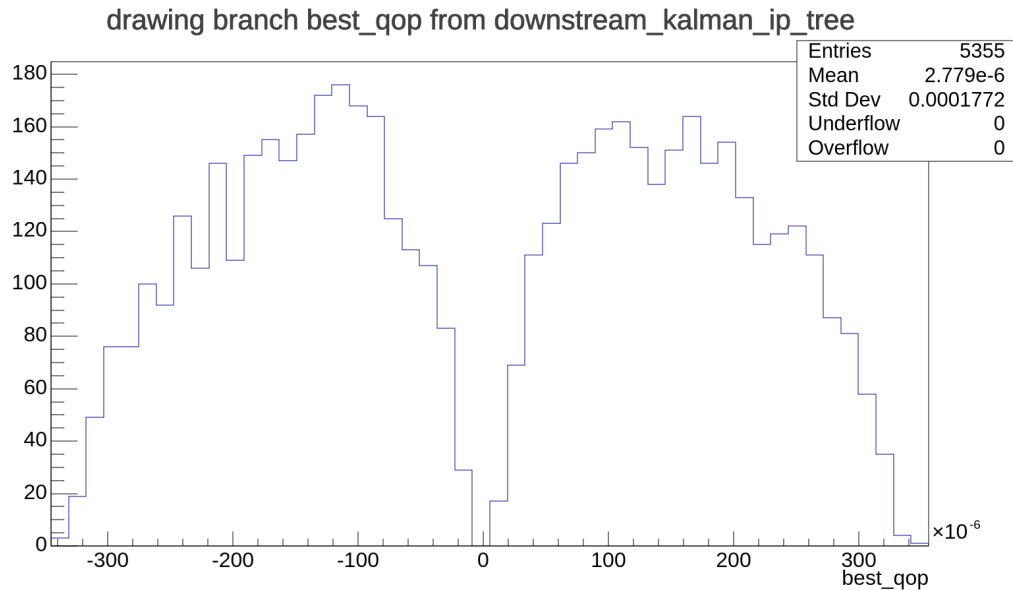


Figure 5.2: best_qop for downstream Kalman filter showing the final qop estimate of downstream tracks

processing, providing a balance between sufficient detail for selection decisions and computational performance suitable for real-time processing in the HLT1. A proposed point to introduce the downstream Kalman filter is between the formation of downstream tracks and the creation of these particles. Listing 5.1 shows this proposal in the downstream reconstruction configuration file⁵ where the downstream Kalman filter could be introduced.

Listing 5.1: Simplified `make_downstream` python configuration function

```

1 @configurable
2 def make_downstream(...):
3     """Performs downstream track reconstruction.
4     """
5     number_of_events = initialize_number_of_events()
6
7     ...
8
9     downstream_tracks = track_maker(decoded_ut, scifi_seeds,
10                                     velo_scifi_matches, ghost_killer_threshold,
11                                     dev_used_ut_hits_offsets)
12
13     ...
14
```

⁵configuration/python/AllenConf/downstream_reconstruction.py - Last checked: 20-07-2025
https://gitlab.cern.ch/lhcb/Allen/-/blob/master/configuration/python/AllenConf/downstream_reconstruction.py?ref_type=heads#L512

```

15 if with_fullKF: # <- The downstream KF takes constructed tracks as input, and prepared the
    ↪ view for downstream_make_particles #
16     kalman_downstream_tracks = make_kalman_downstream(downstream_tracks)
17     dev_downstream_track_states_view = kalman_downstream_tracks['
    ↪ dev_kalman_states_view']
18 else: # <- If KF is disabled, use the default view #
19     dev_downstream_track_states_view = downstream_tracks['
    ↪ dev_downstream_track_states_view']
20
21 downstream_make_particles = make_algorithm(
22     downstream_make_particles_t,
23     name="downstream_make_particles",
24     ...
25     # States
26     dev_downstream_track_states_view_t=dev_downstream_track_states_view,
27     ...
28 )
29
30 output_map = {
31     "downstream_make_particles":
32     downstream_make_particles,
33     ...}
34
35 return output_map

```

After enabling the downstream Kalman filter algorithm, other algorithms can be developed/modified to make use of its output states. The impact assessment of positioning the algorithm at the proposed point in the sequence is pending due to time constraints and technical challenges in the plotting process, which are going to be addressed after the submission of the master thesis.

5.3 Carlson's square-Root Filter

In order to understand the possible differences when switching between the traditional Kalman update/filter steps and Carlson's alternative formulations, we designed a series of systematic experiments in which we replaced the standard "PredictState" and "UpdateState" kernels one at a time (and in various combinations) with their Carlson-based counterparts. From these configurations, three clear takeaways emerged.

First, using the Carlson version for the three UpdateState functions does not greatly reduce quality. As seen in Figure 5.3, the difference is on par with the difference when fixing the similarity bug in Section 5.4.

Using Carlson on the PredictState functions one by one shows a small decrease in quality

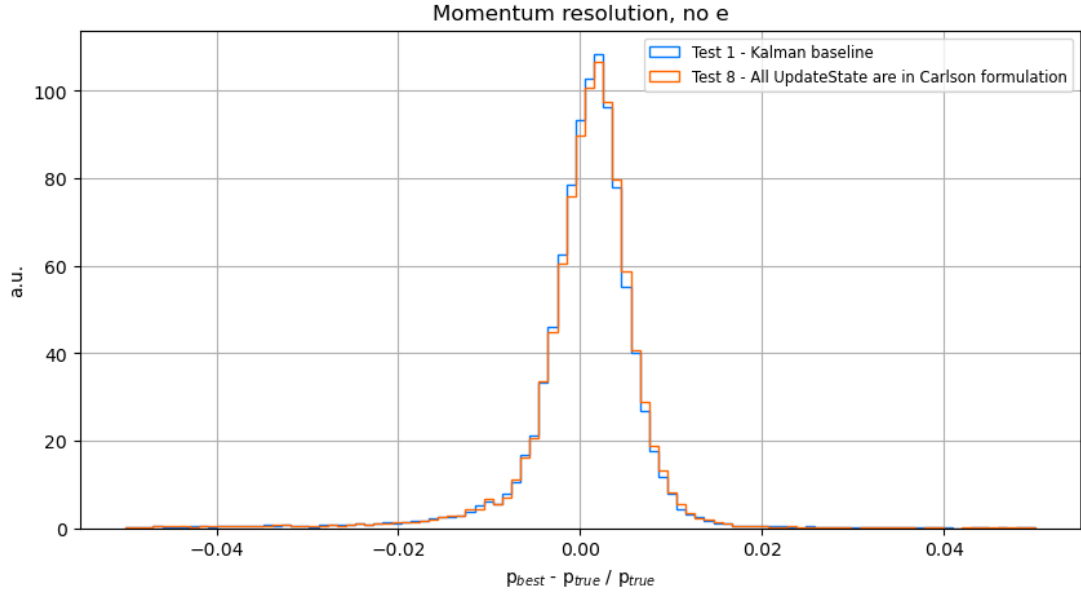


Figure 5.3: All UpdateState kernels in Carlson's sqrt filter formulation

(specially in VELO, where the decrease is nearly zero). Figure 5.4 shows the impact of replacing each of these PredictState with Carlson, and 5.5 shows the same plus all UpdateState being Carlson. In the second case, results are worse than in the first. This is discussed in the second takeaway.

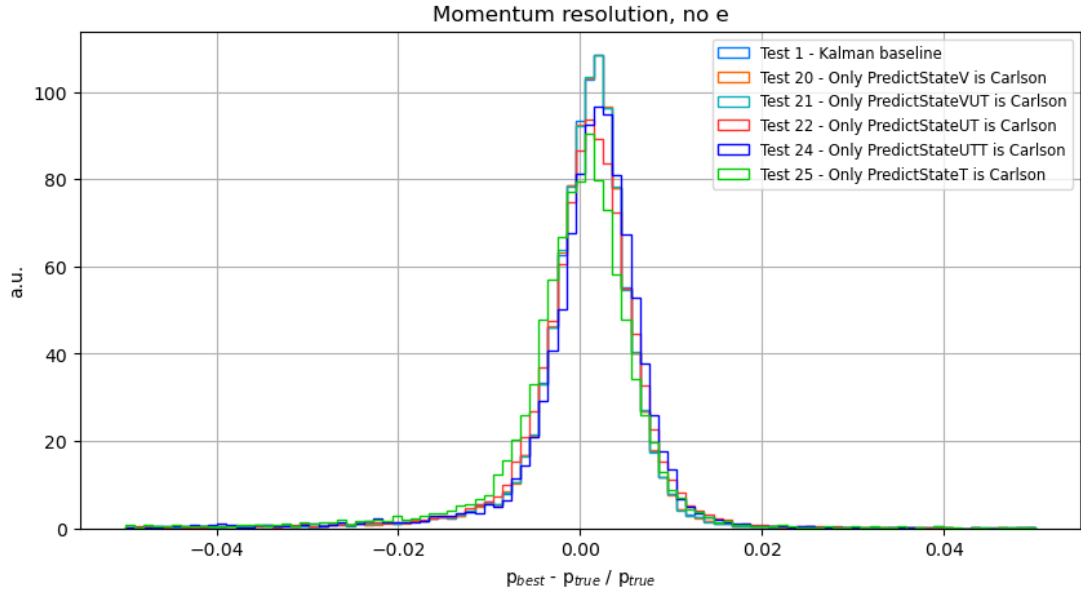


Figure 5.4: PredictState functions replaced by Carlson formulation, one by one

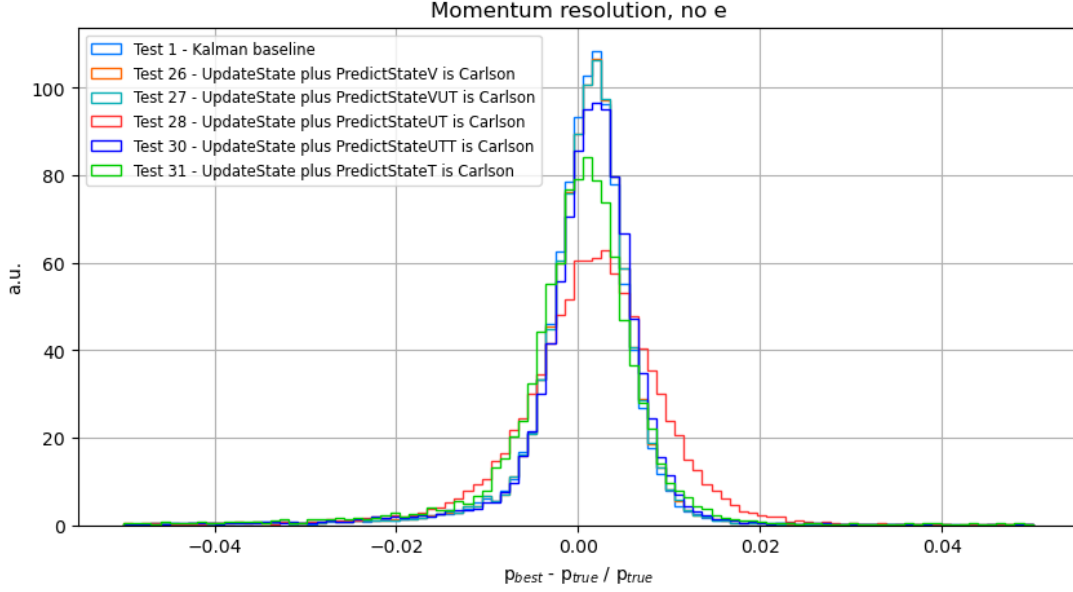


Figure 5.5: UpdateState functions plus PredictState functions replaced by Carlson formulation, one by one

Second, having more and more PredictState functions as Carlson, the quality goes down regularly. Figure 5.6 illustrates this. For these tests, the first approach was the conversion from C to S and back in every occasion⁶, allowing for convenient management of which functions used the Kalman or Carlson formulation. This not only negatively affects performance by introducing a lot of extra computation for every hit of every trajectory, but it also introduces accumulative errors that happen during these operations. In production, S must be maintained throughout the filter. Figure 5.7 shows the difference between having every kernel in the Carlson formulation but maintaining S in memory between kernels or converting it in every step. The effects of the cumulative rounding error are visibly apparent. Nevertheless, currently certain parts of the code require an obligatory conversion to C . Judging by the results in the plot, modifying these parts to work with the triangular matrix could drastically improve the quality of measurements.

Third, there is an exception to the previous points. If the PredictStateTFT function uses the Carlson formulation, independently of being the only function with the Carlson formulation or not, it makes the quality extremely poor, as seen in Figure 5.8. All the previous figures in this section exclude PredictStateTFT for clarity. This kernel is different from the rest because it does not compute the noise matrix Q , and therefore does not add it to C . However, the PredictStateUTT function does not compute Q in its first prediction either, and replacing this

⁶Conversion C and S in every step - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/2108/diffs?commit_id=1bdb4941a084276cb27131a5521978d1f3fdb39c

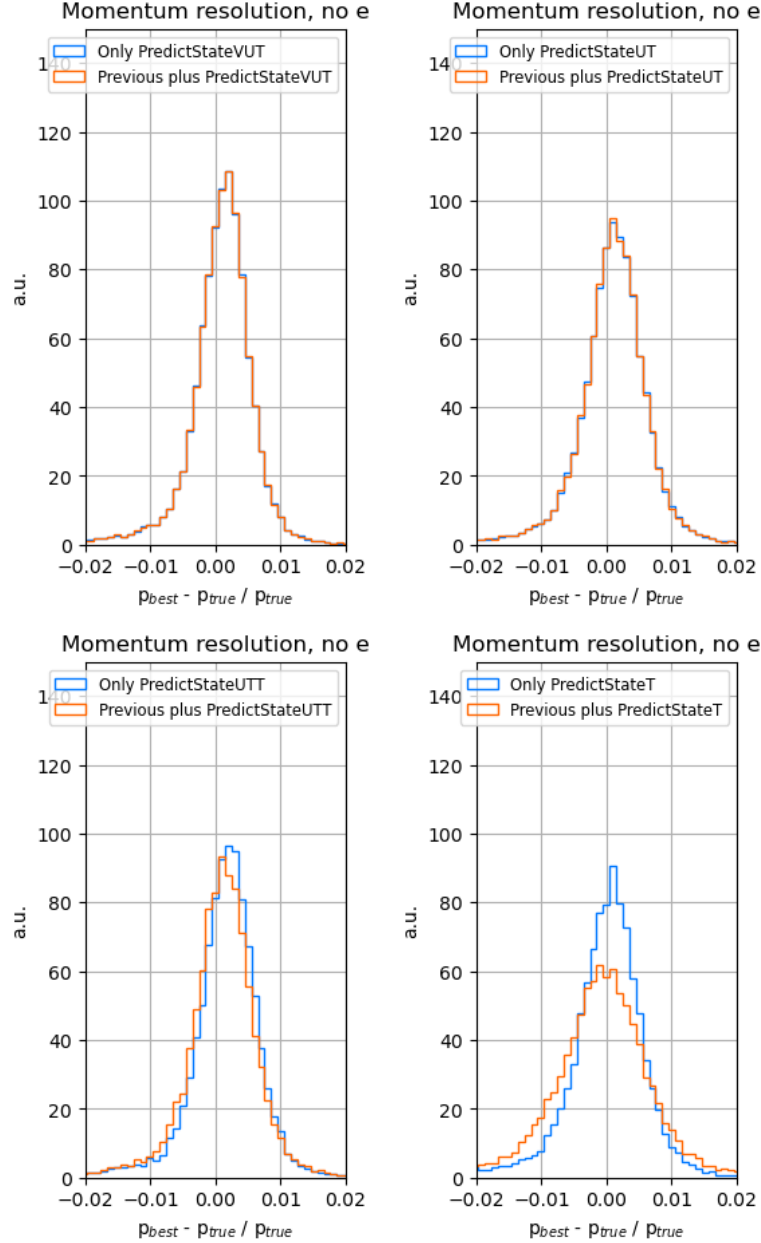


Figure 5.6: Having only one PredictState in carlson versus having one plus all the previous ones. From top to bottom, from left to right: PredictStateVUT, PredictStateUT, PredictState teUTT, PredictStateT

part with Carlson's formulation in the same way does not alter the results in the slightest.

Overall, this indicated that further study is necessary to determine the strange behaviour of PredictStateTFT. Aside from that, Figure 5.7 seems to show that Carlson's formulation might have a better mean (closer to 0), while Kalman seems biased to the right. Although the

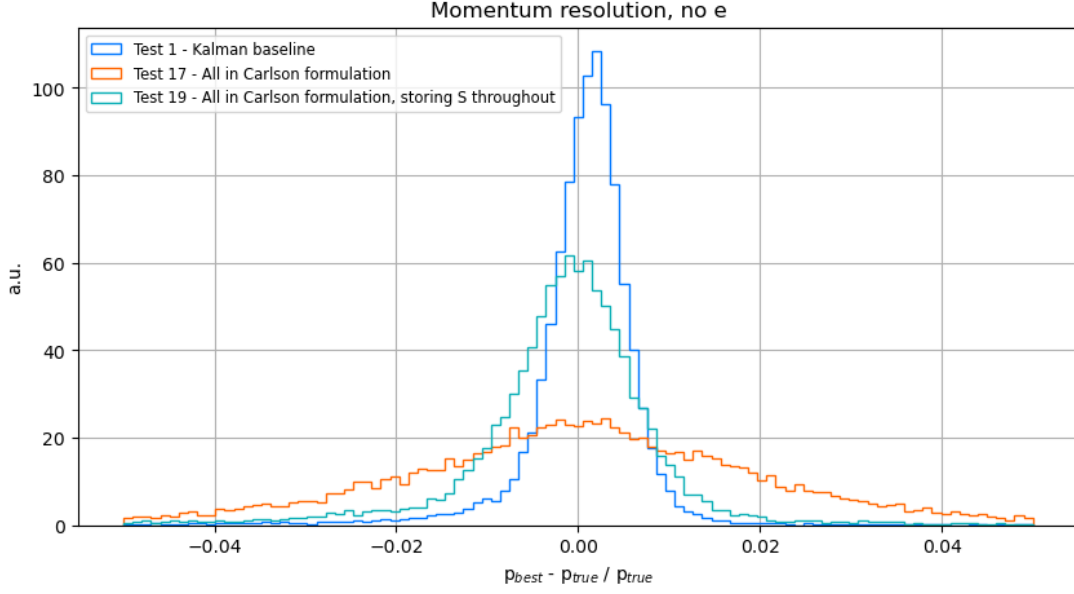


Figure 5.7: Comparing the effects of storing S throughout the code versus conversion in every step

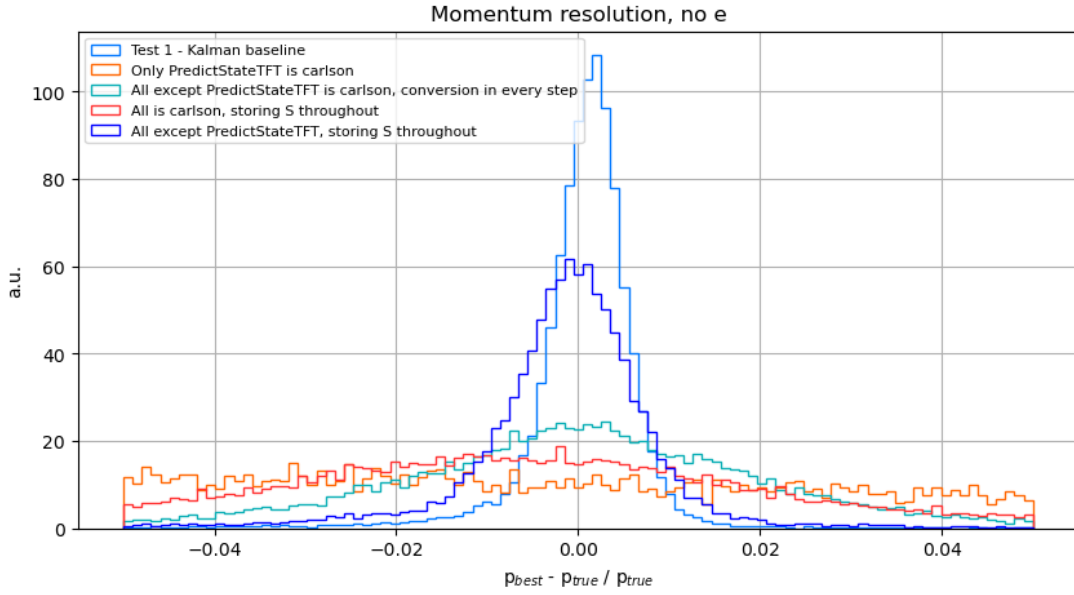


Figure 5.8: Various cases showcasing how PredictStateTFT in Carlson implementation severely degrades quality

standard deviation and peak height could be improved, we think this is an encouraging result.

5.4 Similarity Bug Fix

To analyze the similarity bug fix introduced in Section 4.2, Figure 5.9 illustrates the effects of fixing the first bug found on `similarity_5_5_T`, and also the effects of fixing the second bug found on `similarity_5_5_VP`. The changes are very subtle, but by Lennart’s analysis, they have a positive impact in J_{ψ} mass⁷.

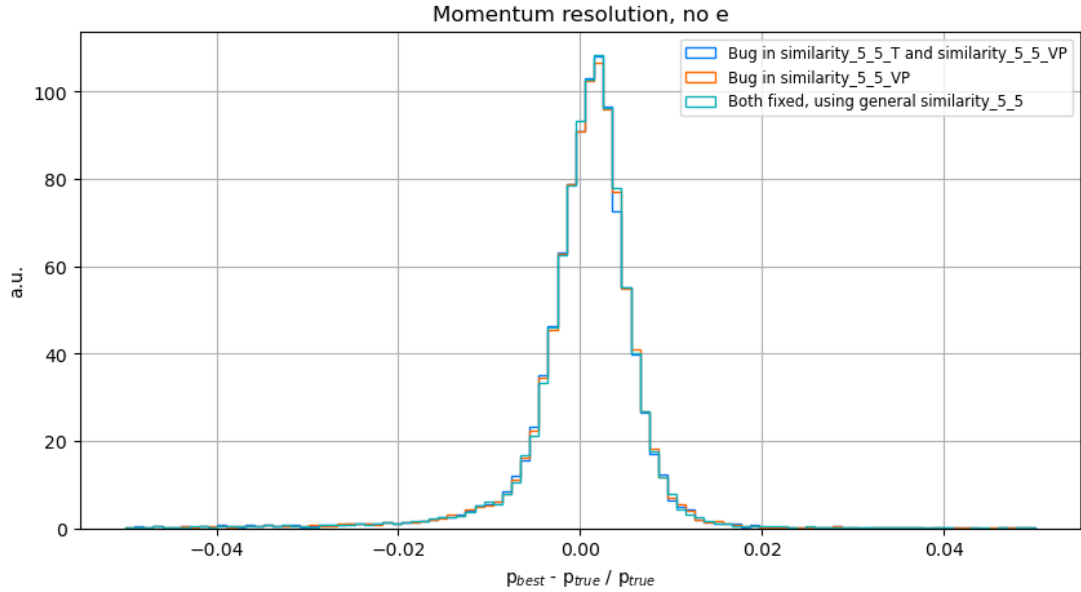


Figure 5.9: Effects in momentum resolution quality of the two bug fixes

The throughput is also affected. To test this, Allen runs pipelines on CI/CD machines that test different run sequences on different systems. CI/CD stands for Continuous Integration/Continuous Delivery; and it encapsulates a set of software development practices that automate the integration, testing, and deployment of code. The machines running these pipelines are the same ones running HLT2 in the event building system. Each node has two AMD EPYC 7502 with 32 cores each, 512 GB of DDR4 RAM @2933 MT/s, and eight PCIe Gen slots[22]. Throughput tests are conducted by running the pipeline in a single-CPU configuration while varying the GPU models to assess scalability in performance across different GPU tiers. Specifically, NVIDIA RTX A5000 GPUs are used for HLT1, so the RTX 2080 Ti and RTX 3090 serve as a comparison of how Allen scales with cheaper and more expensive cards, respectively. Table 5.4 shows the change in throughput for the most relevant sequence for each system. The speed-up for NVIDIA GeForce RTX 2080 Ti is 20% higher than reference (can be due to execution fluctuations), while the other two GPUs have the same throughput

⁷Impact on the J_{ψ} mass - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/merge_requests/1941#note_9521698

as reference. CPU execution slows down, dropping the speed-up against the reference in 3%. These results indicate that the extra computation of using the general `similarity_5_5` kernel is not affecting throughput in any relevant way, so the choice of removing the specialized similarity functions and allocating time to finding possible bugs was the correct one.

Table 5.1: `hlt1_pp_forward_then_matching_with_parkf` Performance

Device	Device Throughput (kHz)	Reference Throughput (kHz)	Speedup (x)	% Change	Status
NVIDIA RTX A5000	81.88	81.58	1.00	0.38	OK
AMD EPYC 7502 32-Core	11.38	11.68	0.97	-2.51	OK
NVIDIA GeForce RTX 2080 Ti	61.50	51.45	1.20	19.52	OK
NVIDIA GeForce RTX 3090	101.11	101.55	1.00	-0.43	OK

Conclusions

This master thesis has overall successfully addressed its primary and secondary objectives, contributing to the parametrized Kalman filter implementation within the LHCb's Allen framework. The work has focused on enabling the parameterized Kalman filter to improve track reconstruction for downstream tracks, exploring alternative numerical stability in Kalman filtering, and other optimisations in the codebase.

In Section 3, the Kalman filter kernel was split into smaller sub-kernels, enabling the implementation of a downstream Kalman filter. Although not yet integrated in the algorithm sequence, this will allow improved trajectory estimates of downstream tracks, which are critical for detecting decay products of long-lived particles. The momentum resolution changes introduced by the filter were studied and the integration of the algorithm into the reconstruction sequence was proposed.

One of the secondary objectives was initiated, by developing Carlson's square-root filter as an alternative to the traditional Kalman filter. The initial results are promising, showing potential for improved numerical stability and impact on track estimate quality.

Two critical bugs in the similarity functions of the parametrized Kalman filter were identified and fixed, leading to small, yet measurable, improvements in momentum resolution. Additionally, deprecated code was removed, significantly reducing the codebase while improving readability and maintainability.

6.1 Personal conclusions

As for my personal conclusions, I have collected some takeaways during my time in this project. On one hand, the large workspace was a challenge in itself: grasping the flow of execution, module distribution and functionality, navigating the codebase for each change and query. I found the amount of documentation insufficient at times, which can be an obstacle for newcomers, as leaning too much on senior's expertise can feel like a bother.

On the other hand, the context for this project is the clear definition of a High-Performance Computing environment, and I am grateful of having developed my master thesis in such an environment. Furthermore, I found the physics programme exciting, as well as the back and fourth with computing and physics experts when I needed help to increase my knowledge. The people working at CERN are very welcoming and inspiring, and its workplace corridors and expositions are brimming with history. Collaborating with CERN has been fantastic, and seeing your work be part of a project of this scale is stimulating.

Future work

Several aspects of this thesis can be explored in more detail in the future. To summarize some of them:

- The incorporation of downstream Kalman filter with other algorithms is the next step of this project, as well as measuring its impact on line selection to determine whether the extra computation is worth it in HLT1. Other aspects such as modifying the starting UT state, defined in `CreateUTSeedState`, can be studied to try to get the best estimate quality. Lastly, albeit primary vertex matching does not make sense for downstream tracks, it maybe makes sense to do secondary vertex matching.
- Regarding Carlson's `sqrt` filter, `PredictstateTFT`'s bizarre behavior needs to be analyzed. In addition, adapting the codebase to only store the upper triangular matrix would avoid unnecessary conversions between S and C , as well as save memory for each thread. Finally, a throughput analysis should be done to determine the efficacy of Carlson's filter against the current Kalman filter.
- There is potential for newcomers to identify bugs, such as the previously mentioned similarity bug, since senior developers and maintainers are often occupied with higher-priority or more impactful tasks.
- In addition to the deprecated states mentioned above, there are other deprecated structures^{1 2 3} that, while not critical for the moment, removal would reduce technical debt.

¹Deprecated `Velo::Consolidated::ConstTracks` - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/blob/master/device/event_model/velo/include/VeloConsolidated.cuh?ref_type=heads#L312

²Potentially deprecated `UT::Consolidated::ConstTracks` - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/blob/master/device/event_model/UT/include/UTConsolidated.cuh?ref_type=heads#L255

³Potentially deprecated `SciFi::Consolidated::ConstTracks` - Last checked: 20-07-2025 https://gitlab.cern.ch/lhcb/Allen/-/blob/master/device/event_model/SciFi/include/SciFiConsolidated.cuh?ref_type=heads#L199

-
- Using half-precision floating-point format for certain variables can be studied in detail, as to analyze the effects on physics efficiency and throughput. With that, we could assess potential cases where storage, computation and communication between CPU and GPU can be greatly reduced without sacrificing much efficiency.

Glossary of Acronyms

HPC *High Performance Computing*: The use of supercomputers and parallel processing techniques to solve complex computational problems that require significant processing power.

MR *Merge Request*: A request to merge changes from one branch into another in a version control system like GitLab, equivalent to a pull request in Github.

CERN from the French *Conseil Européen pour la Recherche Nucléaire*: The European Organization for Nuclear Research, one of the world's largest centres for scientific research in particle physics.

LHC *Large Hadron Collider*: The world's largest and most powerful particle accelerator, located at CERN, designed to collide protons and heavy ions at very high energies to study fundamental particles.

LHCb *Large Hadron Collider beauty*: One of four major experiments, specialized in the study of hadrons containing b and c quarks.

ATLAS *A Toroidal LHC ApparatuS*: A general-purpose detector at the LHC, designed to explore a wide range of physics phenomena, including the search for the Higgs boson, extra dimensions, and particles that could make up dark matter.

CMS *Compact Muon Solenoid*: A general-purpose LHC detector, similar in scope to ATLAS but with a different design approach, is used to study the Standard Model of particle physics and search for new physics phenomena.

Alice *A Large Ion Collider Experiment*: An LHC experiment dedicated to studying the quark-gluon plasma, a state of matter believed to have existed just after the Big Bang, by colliding heavy ions like lead.

VELO *Vertex LOcator*: A sub-detector of LHCb designed to precisely track the positions where particles decay, located very close to the collision point for high-resolution vertex reconstruction.

UT *Upstream Tracker*: A silicon strip detector in the LHCb experiment, placed upstream of the magnet to improve tracking precision and help identify charged particles before they enter the magnetic field.

SciFi *Scintillating Fibre Tracker*: A high-precision tracking detector in LHCb, using scintillating fibres and silicon photomultipliers to detect charged particles with excellent spatial resolution.

RICH *Ring Imaging Cherenkov Detector (RICH I / II)*: Detectors in LHCb that identify charged particles by measuring Cherenkov radiation, enabling the differentiation between pions, kaons, and protons over a wide momentum range.

Glossary of Terms

Particle physics : The branch of physics that studies the fundamental constituents of matter and radiation, and the interactions between them.

Standard Model : The theoretical framework in particle physics that describes all known elementary particles and their interactions (except gravity), including quarks, leptons, bosons, etc,

Decay : The process by which an unstable particle transforms into lighter particles, releasing energy. Particle decays follow probabilistic rules, with certain decay modes occurring more frequently than others, characterized by a particle's lifetime.

CP violation : A charge-parity symmetry or charge-conjugation-parity symmetry violation. CP violation is essential to explain the dominance of matter over antimatter in the universe, as it allows processes that treat matter and antimatter differently.

Boson : One of the two classes of subatomic particles that have integer spin. Certain bosons are force carriers, such as the gluon being the carrier of strong nuclear force.

Higgs boson : An elementary particle associated with the Higgs field, responsible for giving mass to other fundamental particles through the Higgs mechanism, discovered at the LHC in 2012.

Z|W boson : A pair of bosons known as carriers of weak nuclear force.

Quarks : Elementary particles and fundamental building blocks of matter. Quarks combine to form hadrons, such as protons and neutrons, and exist in six flavors: up, down, charm, strange, top, and bottom (beauty).

b quarks : Bottom or beauty quarks. A type of heavy quark, important for studying CP violation and rare decays, playing a key role in experiments like LHCb.

c quarks : Charm quarks. A type of heavy quark, studied for its role in hadronization processes and decay patterns

PP collisions : Proton-Proton collisions. High-energy collisions between protons in particle accelerators such as the LHC are used to study fundamental particles and forces at the smallest scales.

Real-time analysis : Also called trigger systems, they process the enormous data rates from pp collisions in real-time to select only the most interesting events for storage and further analysis.

Fake/Ghost tracks : Reconstructed particle trajectories in a detector that do not correspond to real particles, typically resulting from random combinations of detector noise or hits from various real trajectories.

Alignment : The process of precisely determining the positions and orientations of detector components to ensure accurate particle trajectory reconstruction.

Calibration : the process of adjusting the detector responses to obtain better measurements.

Cherenkov radiation : Electromagnetic radiation emitted when a charged particle moves through a medium at a speed greater than the velocity of light in that medium. This effect produces a characteristic cone of blue light, which can be studied in RICH detectors to identify particle types based on their velocities.

Bibliography

- [1] home.cern. The birth of the web. [Online]. Available: <https://www.home.cern/science/computing/birth-web>
(Cited on Section 1)
- [2] A. Collaboration, “Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc,” *Physics Letters B*, vol. 716, no. 1, p. 1–29, Sep. 2012. [Online]. Available: <http://dx.doi.org/10.1016/j.physletb.2012.08.020>
(Cited on Section 1)
- [3] home.cern. Lhcb weighs up the z. [Online]. Available: home.cern/news/news/physics/lhcb-weighs-z
(Cited on Section 1)
- [4] homeweb.cern.ch. Knowledge transfer group. [Online]. Available: <https://knowledgetransfer.web.cern.ch/about-us>
(Cited on Section 1)
- [5] LHCb collaboration, “The LHCb upgrade I,” *JINST*, vol. 19, no. 05, p. P05065, 2024, all figures and tables, along with any supplementary material and additional information, are available at <http://lhcbproject.web.cern.ch/lhcbproject/Publications/LHCbProjectPublic/LHCb-DP-2022-002.html> (LHCb public pages). [Online]. Available: <https://cds.cern.ch/record/2859353>
(Cited on Section 2)
- [6] gitlab.cern.ch. Allen on gitlab. [Online]. Available: <https://gitlab.cern.ch/lhcb/Allen>
(Cited on Section 2)
- [7] “LHCb Upgrade GPU High Level Trigger Technical Design Report,” CERN, Geneva, Tech. Rep., 2020. [Online]. Available: <https://cds.cern.ch/record/2717938>
(Cited on Section 2)

-
- [8] L. collaboration, “Allen: A high-level trigger on gpus for lhcb,” *Computing and Software for Big Science*, vol. 4, no. 1, p. 7, Apr 2020. [Online]. Available: <https://doi.org/10.1007/s41781-020-00039-7>
(Cited on Section 2)
- [9] S. de Capua, “The LHCb VELO Upgrade,” *PoS*, vol. ICHEP2016, p. 250, 2016. [Online]. Available: <https://cds.cern.ch/record/2287320>
(Cited on Section 2.1)
- [10] W. C. Parker, “Overview of the LHCb Upstream Tracker (UT),” 2016. [Online]. Available: <https://cds.cern.ch/record/2224515>
(Cited on Section 2.1)
- [11] M. Losasso, F. Bersgma, W. Flegel, P.-A. Giudici, J. A. Hernando, O. Jamet, R. Lindner, J. Renaud, and F. Teubert, “Tests and Field Map of LHCb Dipole Magnet,” CERN, Geneva, Tech. Rep., 2005. [Online]. Available: <https://cds.cern.ch/record/1444496>
(Cited on Section 2.1)
- [12] P. Hopchev, “Scifi: A large scintillating fibre tracker for lhcb,” 2017. [Online]. Available: <https://arxiv.org/abs/1710.08325>
(Cited on Section 2.1)
- [13] M. Losasso, F. Bersgma, W. Flegel, P.-A. Giudici, J. A. Hernando, O. Jamet, R. Lindner, J. Renaud, and F. Teubert, “Tests and Field Map of LHCb Dipole Magnet,” CERN, Geneva, Tech. Rep., 2005. [Online]. Available: <https://cds.cern.ch/record/1444496>
(Cited on Section 2.2)
- [14] J. van Tilburg, “Track simulation and reconstruction in lhcb,” PhD-Thesis - Research and graduation internal, Vrije Universiteit Amsterdam, 2005.
(Cited on Section 2.2)
- [15] P. Billoir, M. De Cian, P. Günther, and S. Stemmle, “A parametrized kalman filter for fast track fitting at lhcb,” *Computer Physics Communications*, vol. 265, p. 108026, Aug. 2021. [Online]. Available: <http://dx.doi.org/10.1016/j.cpc.2021.108026>
(Cited on Section 2.2)
- [16] R. Brun, F. Rademakers, and S. Panacek, “ROOT, an object oriented data analysis framework,” 2000. [Online]. Available: <https://cds.cern.ch/record/491486>
(Cited on Section 3.1.4)
- [17] N. A. CARLSON, “Fast triangular formulation of the square root filter.” *AIAA Journal*, vol. 11, no. 9, pp. 1259–1265, 1973. [Online]. Available: <https://doi.org/10.2514/3.6907>

(Cited on Sections 3.2, 3.2.2, and 3.2.2)

- [18] wikipedia.org. Cholesky decomposition. [Online]. Available: https://en.wikipedia.org/wiki/Cholesky_decomposition

(Cited on Section 3.2)

- [19] web.cern.ch. Cern's worldwide lhc computing grid. [Online]. Available: <https://wlcg-public.web.cern.ch/>

(Cited on Section 4.1)

- [20] ——. The lhcb simulation project. [Online]. Available: <https://lhcb-simulation.web.cern.ch/>

(Cited on Section 4.1)

- [21] C. Bozzi, "Utilization of LHCb Offline Computing Resources in 2024," CERN, Geneva, Tech. Rep., 2025. [Online]. Available: <https://cds.cern.ch/record/2924639>

(Cited on Section 5.1)

- [22] indico.cern.ch. Lhcb event building system in run3. [Online]. Available: https://indico.cern.ch/event/1483219/contributions/6317246/attachments/3008890/5304431/EB_at_LHCb.pdf

(Cited on Section 5.4)

