Universidad de Sevilla
Departamento de Ciencias de la
Computación e Inteligencia Artificial

Optimization of high-throughput real-time processes in physics reconstruction

A Thesis submitted for the degree of Doctor of Philosophy
Escuela Técnica Superior de Ingeniería Informática
Universidad de Sevilla

Daniel Hugo Cámpora Pérez

Thesis Supervisors

PhD. Agustín Riscos Núñez
PhD. Niko Neufeld

A mi compañera de aventuras.

CONTENTS

ACKNOWLEDGMENTS

En el verano de 2010 tuve la suerte de ser escogido como *Summer Student* en el CERN. Y a ese comienzo le siguieron nueve largos años aprendiendo y creciendo, tanto en lo profesional como en lo personal. Mucho de lo que he recogido durante mi vida aquí está plasmado en esta tesis.

Tengo a muchos a los que agradecer. A Vero, por la felicidad diaria que es estar junto a ella. Me encanta compartir la vida contigo. A mis padres, por su amor y su apoyo constante. Todo lo que he conseguido es gracias a vosotros.

Quiero agradecer a mis amigos todas las bromas que han hecho la vida más divertida estos años. A mi amigo Lolo, que ha pasado mucho frío, sobre todo en ausencia del gato. A mi amigo Dani García, quien es sutil como un submarino. A los *amigasos*, Álvaro, Paco, Adri y Dani Arenas, por las infinitas risas que nos hemos echado juntos. A José, por los poemas en portugués. A Carlos y Fátima, con los que el barco seguirá a flote pase lo que pase. A Andrés, Zapi y Ale, por nuestras discusiones informáticas, y a Miri, por nuestras discusiones artísticas. A los amigos de siempre, Jesús, Rocío, Fran, Seba y Jorge, con los que tengo un vínculo atemporal. A Fer, con el que disfruto entendiendo las tonterías que pasan en el mundo. A Luz, por nuestra cándida amistad. A Vlado, por las discusiones de filosofía y de la vida. A Ome por las tardes de Smash. A Ana, Cris, David y Shenandoah, por hacer Ginebra divertida a ritmo de Swing. A los amigos del club de ajedrez *Tenis Betis*, que alegraron Ginebra pese al frío. A Miguel, por esas tardes de julio. A mi nueva *familia almeriense*, por todos los momentos de felicidad.

He tenido la suerte de trabajar en el departamento de Ciencias de la Computación e Inteligencia Artificial junto a geniales compañeros que me han apoyado en todo lo que he necesitado. En especial, muchas gracias a mi tutor y director Agustín Riscos Núñez, quien me ha guiado y me ha enseñado lecciones muy valiosas para convertirme algún día en Jedi. También a Mario de Jesús Pérez Jiménez, quien me ha alentado siempre y ha sido un modelo a seguir. Muchas gracias a David *"man"* Orellana Martín por las bromas lingüísticas y el buen humor diario, y a Luis Valencia Cabrera, al que le debo todavía algún que otro queo. A Miguel Ángel Martínez del Amor por las discusiones *cuderas*, que espero que sigan por mucho tiempo. A Álvaro Romero Jiménez y Carmen Graciani Díaz, que me recibían con brazos abiertos cada vez que volvía a la escuela por navidad. A Luis Felipe Macías Ramos, por los primeros pasos en la burocracia del doctorado. A Miguel Ángel Gutiérrez Naranjo, con el que la filosofía y la informática se mezclan en nuestras discusiones. Muchas gracias a Fernando Sancho Caparrini, por motivarme y ofrecerme siempre nuevas ideas. A Ignacio Pérez Hurtado de Mendoza, por las discusiones de C++. A Juan Antonio e Isabel Nepomuceno, por su cercanía y sus consejos.

I would also like to thank all of those who have made the experience at CERN and its surroundings unforgettable.

Many thanks to my friend Nazim, with whom it is always a pleasure to discuss the craziest life stories. To my friends Kazuya and Naomi, for many travels, experiences and board games together. Many thanks to my musical friends Mireia Crispín Ortúzar, Gonzalo Martínez, Laura Cantagalli and John Duxbury, who share my passion for music and have allowed me to sing my way through Switzerland. Thanks to Schnucki, Donal, Matevz, Gianni, Atsuko, Krina, Laura and Kate for all the happy times together at CERN. To Jacob, for our friendship and for his ability to stand *dilemmas*. To Ramón and Carmen, for the yearly sevillanas and rebujitos we have shared. To Marco, Rossana and Ema for their love and our stories together. To Albi, Ludi and Thomas for our hikes and good moments. To Xavi and Elisabet, for their friendship and support. To Arantza, Luismi, Brij and José Mazorra, for the Allen support and for helping give the thesis some sweet style.

Many colleagues and friends have made working at CERN a wonderful experience. Thanks to Niko Neufeld, who has been my supervisor, friend and office mate for the last seven years, and has propelled my career while making it fun. Thanks to Rainer Schwemmer, for his ingenuity, support and most importantly, our karaoke sessions. To Plácido for our work together in Allen and the everyday discussions. To Flavio, for his contributions to the forward tracking and our friendship. To Laura, for her ARM and POWER powers. To Tommaso, for our fruitful DAQ discussions. Thanks to the LHCb Online team for the support, the hard work and the many barbecues. To those that were involved in the HTCC, especially to Omar, Jon, Sebastien, Christian and Luca, who contributed to the early work in this thesis. Thanks to Conor for our discussions and his comments on the thesis. Thanks to O. Bouizi at Intel and C. Potterat for their contributions to the Kalman filter work. Thanks to A. Hehn at NVIDIA for his help throughout the development of Allen. Big thanks to all the Allen team, especially to Dorothea, Roel and Vava, for all the hard work to make the GPU HLT1 a reality. Thanks a lot to the LHCb collaboration, and especially to Ben, Marco Cattaneo and Marco Clemencic, for all the support received throughout the years.

Finally, I am very grateful to Brian Martin, who has always been a good influence, for opening the doors of the ATLAS netadmin team to me and letting me into the fun of Data Acquisition, and in particular for his help in proof-reading this thesis. Thanks to my *prieteni* Dan, Irina, Stefan and Silvia, and to Eukeni, for my first DAQ steps which have evolved into friendship. Special thanks to Prof. Dr. G. Raven and Prof. Dr. I. Kisel for providing reports for this thesis.

Muchas gracias a todos,

Daniel

CONTENTS OF THE DOCUMENT

The present document is organized in four parts, adding up to a total of 11 chapters. Their contents are succinctly described in the following.

Part I: Preliminaries

The **first chapter** introduces some basic notions of particle physics. The LHCb detector, placed at the Large Hadron Collider in the European Organization for Nuclear Research (CERN), and all the subdetectors composing it are briefly discussed. The Data Acquisition system of LHCb is presented in its three constituent steps: Event readout, event building and event filtering. The filtering software to which this thesis contributes is presented as the High Level Trigger. The chapter closes quantifying the challenge the upcoming upgrade detector rates signify.

Chapter 2 introduces architectural concepts and performance metrics of modern processors. The historical processor evolution from sequential into parallel chips is presented. The types of parallel processors are shown, alongside various categories of parallelism, required to make efficient use of modern processors. The relation of memory performance with processor performance is discussed. Several memory-related fundamental concepts are presented, and a visual tool known as the *Roofline model* to characterize processors with the relation between processor and memory performance is described. The first part ends with a description of Graphics Processing Units (GPUs), an architecture for which much of the software in this thesis has been written. GPUs are presented as coprocessors to tackle parallel workloads, and several examples from the literature are shown.

Part II: Parallel algorithms

Part II discusses the reconstruction algorithms contributed as part of this thesis. The contributions are organized in five chapters that cover a variety of relevant problems in High Energy Physics (HEP) applications, and that specifically occur in the High Level Trigger reconstruction sequence of LHCb. All the contributed software targets parallel multi and many-core SIMD architectures. This thesis contributes to the following areas of parallel software for HEP: decoding (chapter 3), clustering (chapter 3, section 3.1), pattern recognition (chapters 4, 5 and 6) and the Kalman filter (chapter 7).

Chapter 3 presents the decoding sequences of four LHCb subdetectors. For each subdetector, a parallel decoding design is presented, alongside details of the implementation developed. For the *Velo* subdetector, the steps of the design are justified. An original parallel clustering algorithm is described in depth, and a validation of the method is presented. The design of the other three subdetectors parallel decoding algorithms are succinctly described in three respective sections. A similar pattern to the Velo decoding is followed to achieve a parallel implementation taking into account the specifics of each subdetector.

Chapter 4 introduces the main concepts and techniques related to the problem of pattern recognition of particle trajectories (tracks). The *efficiency indicators* determining the goodness of tracks are discussed, and the specifics of the criteria of LHCb tracking detectors are presented. An overview of tracking methods, categorized into local and global methods, is described and discussed. This chapter lays the foundations for the following discussions on tracking in the next two chapters.

A parallel Velo tracking algorithm *Search by triplet* is presented and discussed in **chapter 5**. The method is described in detail in publication [1], included as appendix A of this thesis. The preexisting sequential method is analyzed and several shortcomings are identified that prevent parallelization. A parallel local method is designed, targeting SIMD architectures, and CPU and GPU versions are developed. Each of the constituent parts of the algorithm are qualified in terms of computational complexity. The algorithm uses efficient memory structures for SIMD architectures, and data reductions based on tight search windows. An iterative two-step tracking procedure guarantees no revisits of detector measurements, resulting in an efficient access pattern to processor memory, using spatial and temporal locality.

Chapter 6 presents a parallel algorithm for Forward tracking, which involves extending tracks to the LHCb *SciFi* subdetector.

The bending of particles due to the LHCb magnet is described, alongside a simplified model describing its effect. The sequential algorithm used in previous runs of the detector is presented and discussed in depth. A parallel algorithm *Looking Forward* is presented, which consists of 12 steps. The design is discussed and each of the steps is qualified in terms of computational complexity. A novel triplet seeding method using specialized hardware is presented, which builds triplets similarly to Search by triplet. The algorithm reduces branched code, and runs efficiently on GPU architectures due to their data-parallelism.

This part ends with the contributions on the Kalman filter in **chapter** 7. The chapter introduces the Kalman filter formulation and links it with the LHCb detector use case. The publication [2] is presented as part of this thesis in appendix B. The existing literature is analyzed, and it is deemed necessary to develop a solution given the specific conditions of the LHCb Kalman filter execution. Three software algorithms are contributed: a proto-application *Cross Kalman Mathtest* allows to compare performance across CPU and GPU architectures; the *Cross-Kalman* application mimics the conditions under which the Kalman filter is executed in the LHCb software and serves as a cross-architecture implementation of the Kalman filter aiming to maximize performance; the findings of the previous two applications are integrated in the LHCb framework in *TrackVectorFitter*, and the performance is validated under LHCb framework run conditions. The applications use a custom scheduler to efficiently use data parallelism in SIMD processors. Roofline models of several processors are shown, demonstrating the arithmetic formulation obtains the peak performance of the processors analyzed.

Part III: Framework

The software contributions of this thesis target a variety of architectures. Concretely, the LHCb High Level Trigger 1 reconstruction application was identified as a target where GPUs could be used. However, the LHCb software was not built to efficiently use hardware accelerators. This thesis contributes a framework to run HEP reconstruction on GPUs (chapter 8). During the development of this thesis, a parallel realization of the full High Level Trigger 1 application was completed in this framework. The physics efficiency of the sequence is presented (chapter 9), and the performance of the entire GPU HLT1 sequence is also described (chapter 10).

Chapter 8 presents a framework for massively parallel physics reconstruction *Allen*. The framework allows algorithms to be run in parallel on thousands of collision events concurrently, exploiting many-core architectures. The design of the framework and all its parts are discussed. The control flow permits execution of a defined set of algorithms. The data flow copes with the memory capacity constraints of GPUs. The performance of the framework is presented under various configurations, and Continuous Integration features are shown.

Chapter 9 discusses the physics efficiency of the tracking sequence. The efficiency of the reconstruction after processing three subdetector algorithms is presented. The efficiency indicators of chapter 4 are used. The HLT1 physics efficiency requirements are shown to be met.

This part finishes with a performance analysis of the HLT1 sequence run on GPUs in Allen, in **chapter 10**. The methodology of the tests is presented, and the HLT1 performance is discussed in depth, on a variety of GPU hardware. Performance is analyzed on its own, and as a function of price and power consumption. The entire sequence is profiled and characterized, and parameters are optimized through scans. Updated results of the Velo sequence described in chapter 5 are presented. Finally, integration considerations into a prospective Data Acquisition System with GPU processing are discussed.

Part IV: Thesis results

The conclusions of the thesis are discussed in **chapter 11**. The presented work makes significant contributions in various HEP related software areas. It also represents the first attempt at a reconstruction of the LHCb HLT1 trigger stage on GPUs, to which the framework and much of the tracking sequence have been presented. Future work will further this development and extend upon the results presented here with the intention of making the GPU HLT1 a production-ready environment.

CONTRIBUTIONS

It is worth noting the following original contributions of this document.

- The parallel decoding algorithms designs of the subdetectors Velo, UT, SciFi and muons are original contributions of the author. The entirety of the Velo and UT decoding sequences have been implemented by the author, including the Velo clustering design and implementation. The parallel SciFi decoding implementation has been done in collaboration with L. Funke, and the parallel muon decoding implementation in collaboration with D. Pliushchenko.
- The Velo tracking sequence has been designed and implemented by the author. The original work has been presented in the following publication:
  - D. H. Cámpora Pérez, N. Neufeld, and A. Riscos Núñez. A Fast Local Algorithm for Track Reconstruction on Parallel Architectures. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2019), pp. 698–707. Included as appendix A.
- The UT tracking sequence has been done in collaboration with P. Fernandez Declara. The following publication stems from this work:
  - P. Fernandez Declara, D. H. Cámpora Pérez, J. Garcia-Blas, D. vom Bruch, J. D. Garcia, and N. Neufeld. A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures. In: *IEEE Access* (2019), pp. 91612–91626.
- The Forward tracking parallel algorithm design and the triplet search therein is original work by the author. The implementation has been done in collaboration with D. vom Bruch and F. Pisani. This work represents the first implementation of the LHCb Forward tracking algorithm on GPUs.
- The Kalman filter applications *Cross-Kalman Mathtest*, *Cross-Kalman* and *TrackVectorFitter* have been designed and implemented by the author. The collaboration with O. Awile and O. Bouizi has led to an in-depth analysis of modern Intel architectures and the elaboration of the Roofline

plots presented as part of [2]. C. Potterat has helped in the comparison of physics efficiency performance figures. The work has been presented in multiple occasions in the course of this thesis:

  - D. H. Cámpora Pérez. LHCb Kalman Filter cross architecture studies. In: *Journal of Physics: Conference Series* 898.3 (2017), p. 32052.
  - D. H. Cámpora Pérez, O. Awile, O. Bouizi, N. Neufeld. Cross-architecture Kalman filter benchmarks on modern hardware platforms. In: *Journal of Physics: Conference Series* 1085 (Sept. 2018), p. 032046.
  - D. H. Cámpora Pérez, O. Awile, and C. Potterat. A High-Throughput Kalman Filter for Modern SIMD Architectures. In: *Euro-Par 2017: Parallel Processing Workshops*. Springer International Publishing, 2018, pp. 378–389.
  - D. H. Cámpora Pérez and O. Awile. An efficient low-rank Kalman filter for modern SIMD architectures. In: *Concurrency and Computation: Practice and Experience* 30.23 (Dec. 2018), e4483. Included as appendix B.

- Finally, the *Allen* framework has been designed and originally implemented by the author. Further iterations of the framework are the effort of a collaboration led cooperatively by R. Aaij, D. vom Bruch, and the author. The work of Allen contains at this moment contributions of tens of developers and several external collaborators. The physics efficiency and performance of the framework are the fruits of this collaborative work during the course of 18 months.

Part I

PRELIMINARIES

1 LHCB

CERN, the European Organization for Nuclear Research, is the biggest particle physics laboratory in the world. At CERN, a wide variety of physics experiments take place, exploring fundamental questions with regards to the composition of the universe, such as what happened in the Big Bang, the difference between matter and antimatter, or the nature of dark energy and dark matter.

The drive for discoveries in the particle physics domain requires the latest technology in all components involved in the detection of particle collisions. At CERN, a network of particle accelerators accelerate particles to a speed close to the speed of light at the Large Hadron Collider (*LHC*) [3], a 27-kilometer synchrotron accelerator. High precision measurements of particle collisions are carried out in the four major particle detectors ATLAS, ALICE, CMS and LHCb. Trigger and data acquisition systems filter data in real-time at a rate of tens of millions particle collisions per second. A global distribution system *Grid* distributes the data around the globe for posterior analysis. All elements in this chain are state-of-the-art, and new technologies are continuously being explored for pushing the limits of science in the search of *new physics*.

The CERN accelerator complex is depicted in Figure 1.1. Two kinds of particles, protons and heavy ions, are injected into the accelerators at different times. Particles are accelerated through the linear accelerator *LINAC2* and the synchrotrons *Booster*, *PS* and *SPS* prior to being injected into the LHC. The energy of particles is increased within every subsequent accelerator, to a design energy of 6.5 TeV at the LHC. Particles are collided in each of the four major LHC detectors at a combined energy of 13 TeV.

Particles are not accelerated individually, but are rather grouped into *particle bunches*. Hence, instead of individual particle collisions, crossings of particle bunches are discussed. The probability of an actual collision happening when two particle bunches cross is measured by the *average number of collisions per*

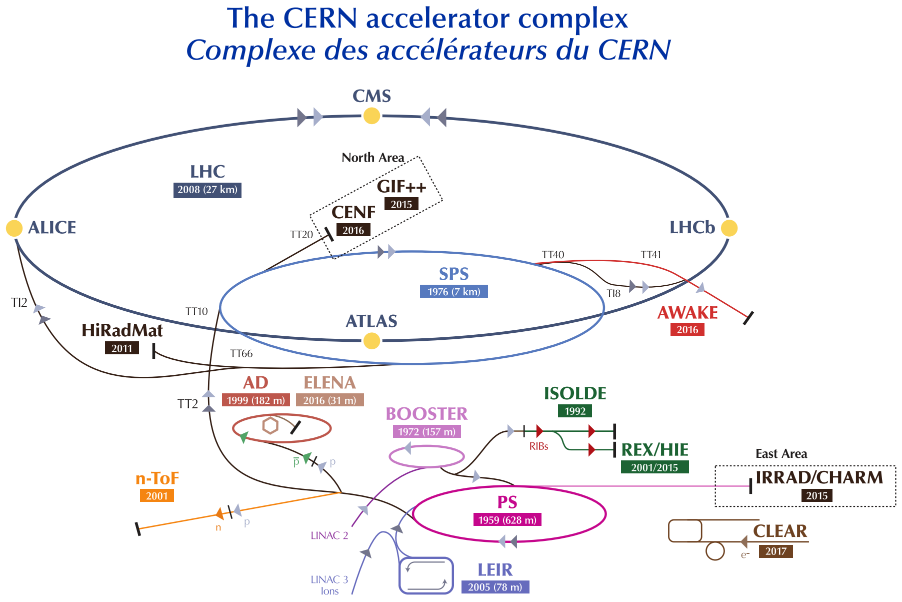


Figure 1.1: CERN accelerator complex. Image from [4].

bunch crossing ν. Each bunch crossing is also referred to as an *event*, and the event rate is measured in Hz. The design event rate of the LHC is 40 MHz, where one event occurs every 25 ns.

Even though the design collision energy is not foreseen to change, other factors impact the collision rate. The *luminosity* is a metric used in accelerator physics to determine the number of particle collisions detected N in a unit of time t to the interaction cross section σ,

$$L = \frac{1}{\sigma}\frac{dN}{dt} \tag{1.1}$$

The luminosity of the LHC beam is set to increase in the next data-taking period to start in 2021, hence increasing the collision rate by roughly $5\times$. The LHC experiments are in an upgrade phase from 2019 until 2021, whereby many components pertaining to the detectors and data acquisition systems will be either updated or changed completely, in the pursuit of new physics results.

1.1 A LARGE HADRON COLLIDER BEAUTY EXPERIMENT

The Large Hadron Collider beauty (*LHCb*) experiment [3, 5] is one of the four major experiments at the LHC. It is a single-arm forward spectrometer, designed for precision measurements of the decay channels that could explain matter-antimatter asymmetry, also known as *CP violation*.

The LHCb detector will be upgraded through 2021 in order to increase precision by 10 times on the main observables of the b and c-quark sectors. The LHCb upgrade detector is depicted in Figure 1.2. The ordinate coordinate system depicts the beam line axis, Z, and is centered at the nominal collision point.

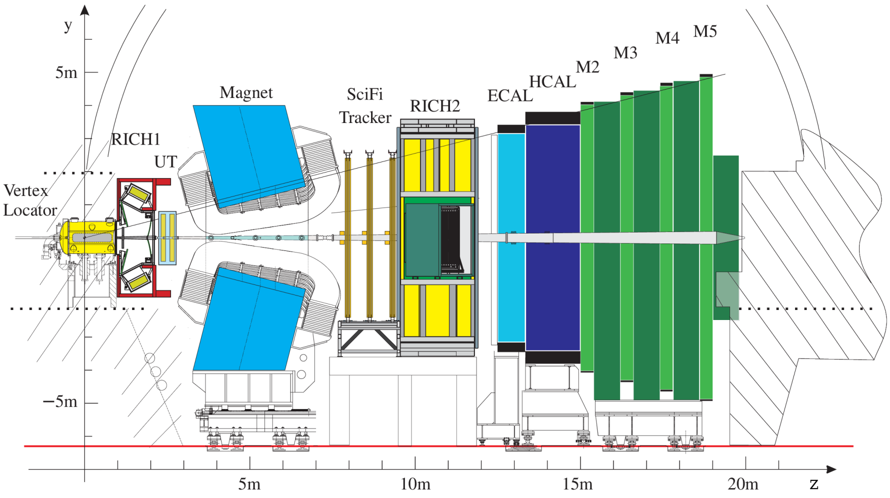


Figure 1.2: LHCb upgrade detector.

When particles collide at the point of collision placed inside the LHCb detector in the high energy conditions of the LHC, the original particles decay into new particles that leave traces in the subdetector instruments conforming the LHCb detector. The process of *reconstruction* consists in reproducing the conditions under which said traces were left in the detector. The quality of the reconstruction achieved can be measured by Monte Carlo simulations in terms of *reconstruction efficiency*. A higher reconstruction efficiency leads to a better identification of the phenomena driving particle decays, and ultimately to a better understanding of the physics foundations therein.

The LHCb detector does not have full coverage. That is, only a subset of produced particles will be detected at LHCb. The coverage angle of the detector in the *forward region* (positive side of horizontal axis in Figure 1.2) is of $300\,\mathrm{mrad}$ in the XZ plane, and of $250\,\mathrm{mrad}$ in the YZ plane. The LHCb magnet bends charged particle trajectories in the XZ plane, which explains the difference in the coverage angles. A small subset of particle trajectories in the *backward region* are measured at the first tracking subdetector in order to identify collision vertices (cref. 1.1.1).

The LHCb detector is composed of three kinds of subdetectors. Tracking subdetectors detect signals in the particles path, from which the momenta and collision vertices are derived. Cherenkov subdetectors measure the velocity of particles. Finally, calorimeters measure the deposited energy of hadrons and electrons. These instruments allow the identification of individual particles by applying relations between momentum, mass and velocity and accounting for relativistic effects.

1.1.1 *Tracking subdetectors*

As particles traverse the tracking subdetectors of LHCb, they interact with detector technology placed in their path. The problem of track reconstruction consists in determining the particle trajectories or tracks left by each individual particle throughout the detector.

The tracking system of LHCb [6] consists of three subdetectors: The Vertex Locator (*VELO*), the Upstream Tracker (*UT*) and the Scintillating Fibre Tracker (*SciFi*). A magnet, placed between the UT and the SciFi, bends particles as a function of their charge in the XZ plane. Figure 1.3 depicts the tracking system of LHCb, alongside all possible track types.

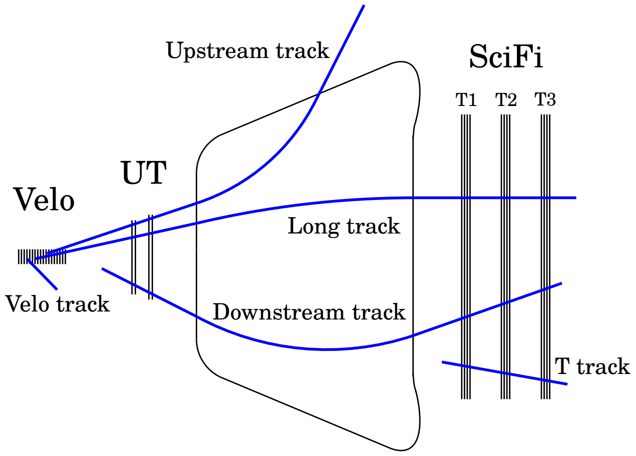
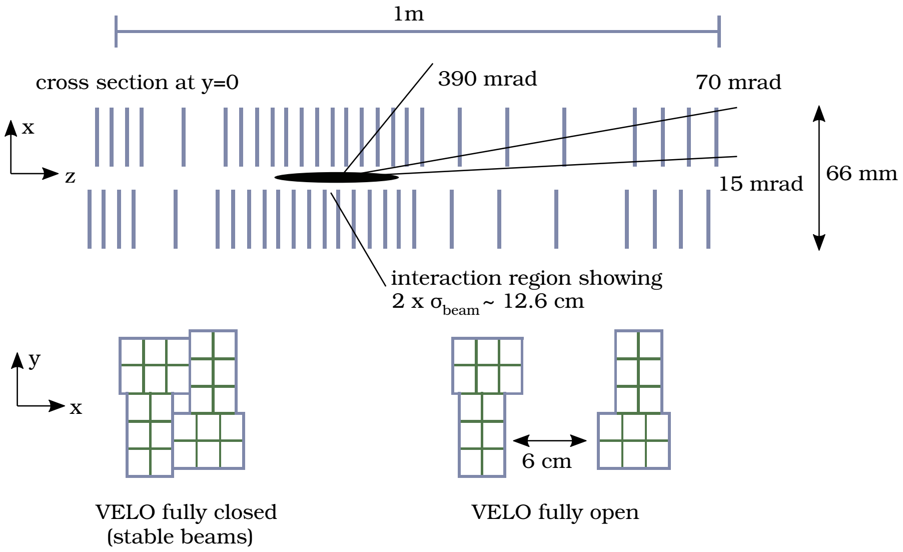


Figure 1.3: LHCb track types.

The subdetectors visited by each particle determine the particle track type. VELO, upstream and long tracks are produced at the VELO. Even though the LHC beam collisions take place inside the VELO, produced particles may decay in their path, originating new particles in non-primary vertices. Downstream and T tracks are produced from particle decays, and originate from non-primary vertices.

VELO

The Vertex Locator [7] is a tracking subdetector placed very closely to the interaction point. It consists of 52 modules placed on sides of the beam line. Each module in turn consists of 12 chips with a resolution of 256 × 256 pixels each. Figure 1.4 depicts the entire detector alongside the beam axis (top), and a detail of one module (bottom). The VELO can be retracted while there is no stable beam in the LHC, in order to avoid damage and increase its lifetime.



(a)

Figure 1.4: A schematic of the upgrade Velo detector.

The VELO allows for precision measurement of the collision vertices, as well as creation of seeds for further track reconstruction in subsequent tracking detectors [8]. The LHCb magnet does not influence particle trajectories within the VELO, and hence particles travel in a straight line in this subdetector.

UT

The Upstream Tracker [6] subdetector consists of four planes of silicon strips, named UTaX, UTaU, UTbV and UTbX. The first and last planes have vertical strips, whereas the middle planes are tilted by $-5°$ and $5°$ respectively. Figure 1.5 shows an overview of the subdetector. By combining the measurements from the tilted U and V planes, the Y coordinate can also be determined. Each UT plane can be divided into 3 regions with different geometries, where the inner-most region has a finer granularity (orange in the Figure), and the outer regions have coarser granularity (yellow and green in the Figure).

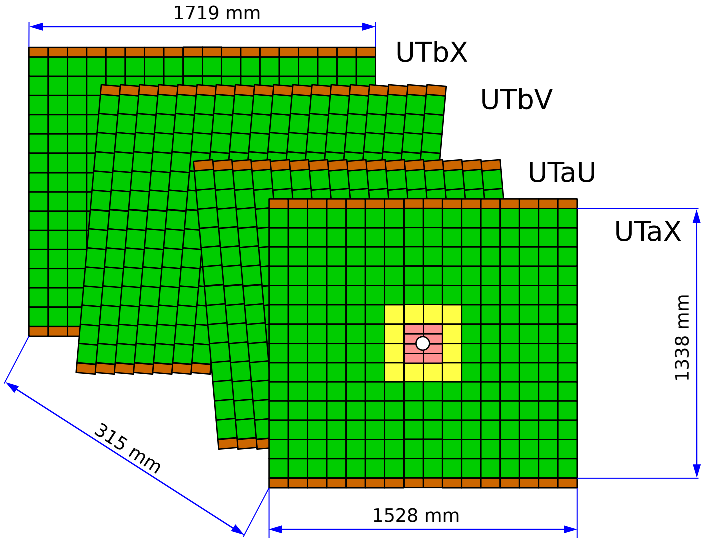


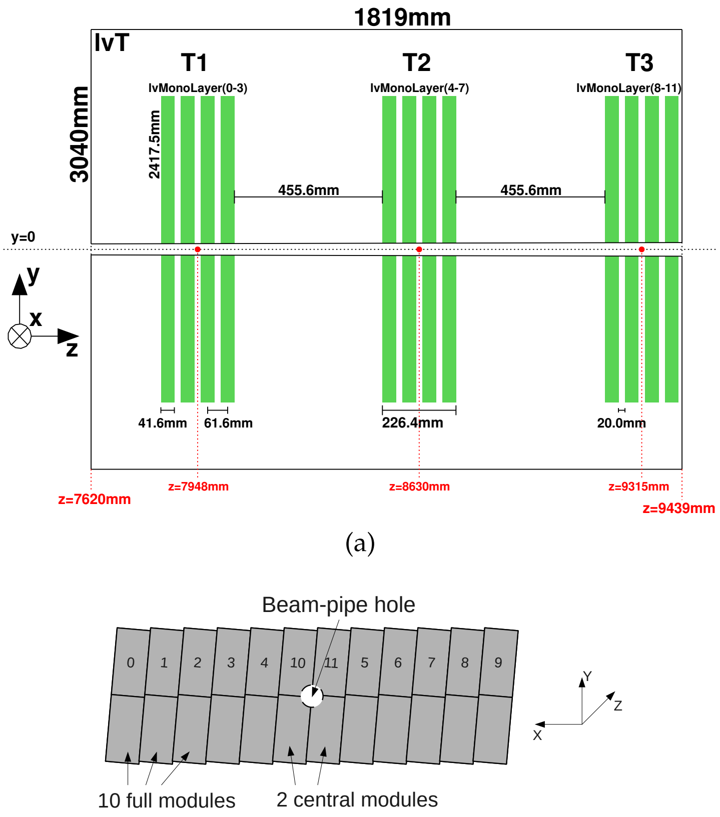
Figure 1.5: Overview of the UT subdetector.

The UT adds information to the tracks reconstructed in the Velo and SciFi. The presence of a residual magnetic field allows measuring the charge and momentum of particles. It allows reconstruction of particles produced outside of the Velo, and of low momentum particles that form upstream tracks.

SciFi

The Scintillating Fibre Tracker [6] consists of three stations placed after the LHCb magnet, in the forward region. They were designed to provide standalone pattern recognition with a high efficiency together with high resolution in the bending plane of the magnetic field. Each station is composed of four layers $\{x, u, v, x\}$, with scintillating fibers orientated at {0°, 5°, −5°, 0°} respectively. A side view of the SciFi tracking stations is shown in Figure 1.6. Each layer is composed of 12 modules, where the two central modules feature a cut-out to allow the beam-pipe to pass through the detector.

Modules are composed of six layers of thin scintillating fibres that react to the passing of charged particles by emitting light



(a)

(b)

Figure 1.6: (a) Arrangement of the SciFi tracker detection layers. (b) Modules within one U plane. Figure from [6].

into its fibre ends. Fibres are arranged in parallel, and a fibre is 250 µm in diameter. Silicon Photo-Detectors *SiPMs* are solid state photon detection devices that detect photons into pixel channels. Each SiPM is composed of 2 blocks of 64 channels each. Figure 1.7 depicts a SiPM (left), a detail of several contiguous channels with the placement of six fibres (center), and a pixel (right).

The position of the particle can be determined with a weighted average of neighboring fired pixel channels. Figure 1.8 depicts this process. Photons produced in each fibre fire SiPM pixels. A weighted average sum is then performed over the fired pixels of each channel in order to determine the position of the particle. The signal is discriminated according to two conditions: There must be at least one channel with a charge over the *seed threshold*, and the sum of all neighboring channel charges over the *neighboring threshold* must be over the *sum threshold* [10].

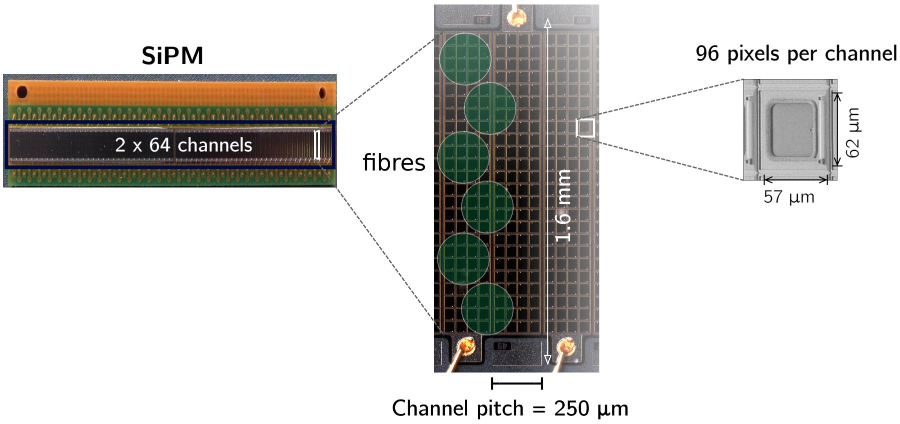


Figure 1.7: Detail of SiPM. Image from [9].

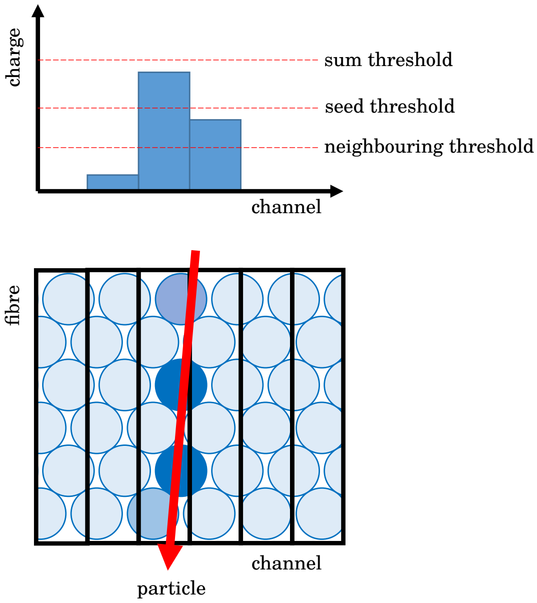


Figure 1.8: Detection of a particle position as it crosses a module.

The SciFi subdetector allows detection of long, downstream and T tracks. Long tracks have a good momentum resolution. Tracks reconstructed in the SciFi subdetector are further analyzed with the particle identification system.

1.1.2 *Particle identification system*

Muon stations

Detection of muons is a fundamental part of the physics program of LHCb. In particular, they are required for detecting very rare decays[1] such as $B_s \rightarrow \mu^+\mu^-$, $B_s^0 \rightarrow \mu^+\mu^-$ or $B^0 \rightarrow K^*\mu^+\mu^-$, which can provide evidence of New Physics.

The upgraded LHCb detector will feature four muon stations [11], as shown in Figure 1.9. The stations are placed behind the hadronic calorimeter, and are interleaved with iron walls that act as muon filters. The muon detectors are equipped with Multi–Wire Proportional Chambers *MWPCs*, and the detector is divided in several regions with a varying granularity of MWPCs according to the expected variation in particle rate from the central regions to the periphery.

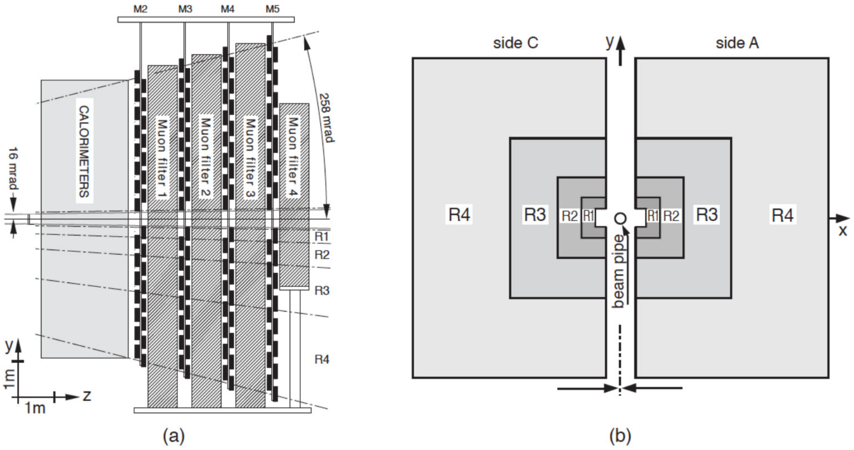


Figure 1.9: (a) Side view of the Muon Detectors. (b) Station layout with the four regions R1-R4 indicated.

The requirements of this subdetector are to guarantee a high reconstruction and identification performance of muons, while maintaining a low misidentification rate to other particle types. These requirements were already met prior to the LHCb upgrade, and have in fact relaxed for the upgrade as the LHCb trigger system will be fully done in software (cref. 1.3).

Cherenkov detectors

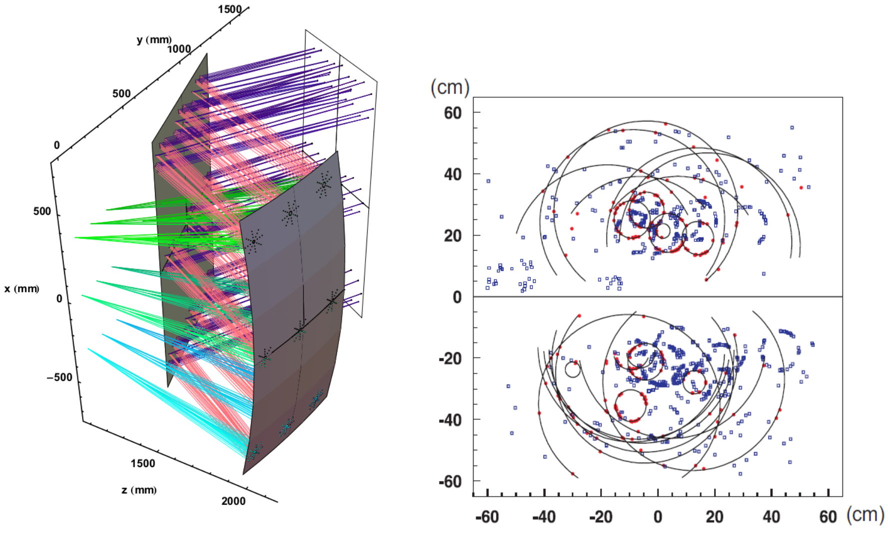
The speed of light depends on the refractive index of the medium of tranmission. It is possible that particles move through certain media at a speed faster than the speed of light in that medium. When that occurs, a cone of photons is emitted from the particle

[1] The particles referenced are part of the Standard Model of particle physics. μ are fundamental lepton particles, whereas B-mesons and K are two-quark combinations known as mesons.

at an angle θ, known as *Cherenkov radiation*. There is a relation between the angle of emission of the cone of photons, the refractive index of the material or radiator n and the velocity of the particle β [12]:

$$\cos(\theta) = \frac{1}{n\beta} \tag{1.2}$$

LHCb is equipped with two Ring Imaging Cherenkov (*RICH*) detectors [11], *RICH1* and *RICH2*. Each of them is composed of a radiator gas, mirrors and Multi-anode Photon Multiplier detectors *MaPMTs*. As particles move through either of the RICH radiators of LHCb, a cone of Cherenkov light is produced. The photons produced are reflected on a section of a spherical mirror and a planar mirror, prior to being detected in MaPMTs. Figure 1.10a depicts this process. The resulting image in the MaPMTs, shown in Figure 1.10b, contains slightly deformed circumferences that can be reconstructed and assigned to tracks, for a precise determination of the velocity of the particle.



(a)

(b)

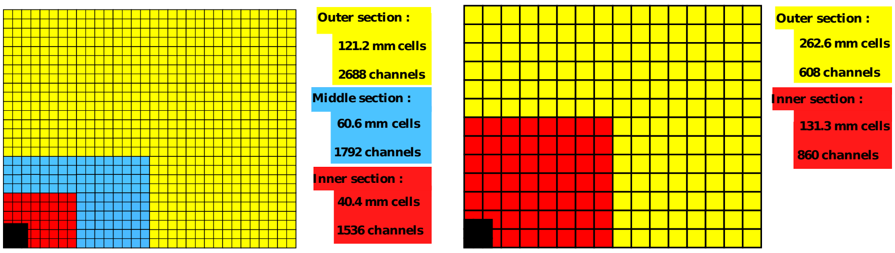
Figure 1.10: (a) Simulation of Cherenkov photons and their reflection off the mirrors of RICH1. (b) Simulation of detected Cherenkov photons in both sides of RICH1.

The RICH reconstruction yields a precision measurement of the velocity of particles through the radiator gas. When combined with the momentum measured in the tracking stations, this allows identification of individual particles. The RICH system of LHCb provides particle identification of charged hadrons over the momentum range 1.5–100 GeV.

Calorimeters

The main purpose of a calorimeter is to measure the energy deposit and position of particles. The LHCb calorimeter system [11, 13] is composed of a hadronic calorimeter (*HCAL*) and an electromagnetic calorimeter (*ECAL*). The system can identify hadrons, electrons and photons. In particular, in LHCb they enable the detection of B-decay channels containing a prompt photon or π^0.

Figure 1.11 show a front view of the LHCb calorimeters. The hit density varies by two orders of magnitude between the sections closer to the beam pipe and the outer sections. Therefore, the size of the cells vary accordingly. The ECAL and HCAL absorb energy of photons, electrons and hadrons.



(a) (b)

Figure 1.11: (a) Electromagnetic calorimeter (*ECAL*). One quarter of the detector is shown. The cell dimensions of the ECAL are shown. (b) Front face of the hadronic calorimeter (*HCAL*). One quarter of the detector is shown.

1.2 THE DATA ACQUISITION SYSTEM

The upgraded LHCb detector will produce data at an event rate of 30 MHz with an average event size of 100 kB[2], for an estimated total throughput of 40 Tb/s. In order to cope with the immense amount of data produced by the detector, data are fed through a data acquisition system and filtered prior to being stored into long-term storage.

[2] The *event size* refers to the size of the raw data of each bunch crossing collision event. The measurements in all subdetectors per LHCb event amount to an expected average 100 kB, and an expected maximum of 150kB.

The data acquisition (DAQ) system [14, 15] is a real-time system[3] that distributes the event fragments received from the front-end electronics of the subdetectors composing LHCb, *builds* events by combining the event fragments into coherent self-contained contiguous blocks of data, and distributes them to an *event filter farm* of commodity servers that filters data by reconstructing events and selecting specific events that are of interest to the current physics understanding.

The data acquisition system of LHCb is composed of the readout system, the event builder and the event filter. Figure 1.12 presents an overview of the system, with an estimate of the servers, links and storage required for the upgrade. Data are read out from the detector front-end electronics into around 500 event builder PCs. Each of these PCs distribute the individual event fragments to a single destination at a time following a synchronized round robin scheme, through the event builder network. Data are finally transmitted for further processing to the event filter farm, which is expected to consist of up to 4000 commodity servers.

1.2.1 *Event readout*

Data from the front-end electronics of the LHCb subdetectors are fed into the DAQ system in a distributed manner to around 500 data acquisition cards, known as *TELL40s*. Data are fed using 10000 simplex optical links, with a custom protocol *GBT* [16], certified to operate under radiation-hardened conditions. The front-end electronics, the TELL40s and the event builder network are synchronized with the LHC collision frequency through the timing and fast control system (*TFC*).

TELL40s are implemented as a PCI form-factor card known as the *PCIe40*, connected to PCI Gen-3 slots of event builder PCs. Each of the TELL40s receive data of a portion of the LHCb detector through a maximum of 48 GBT links adding up to a data rate of around 80 Gb/s. The TELL40s pack the data in multi-event packets (*MEPs*), and transmit the data to a pinned

[3] The term *real-time* is employed here to refer to a DAQ system which requires processing a design throughput. In contrast, *hard* real-time DAQ systems (such as the previous LHCb DAQ in Run 2) must also meet a tight latency requirement. The upgraded DAQ uses a distributed RAM buffer as a readout buffer, relaxing the latency requirement.

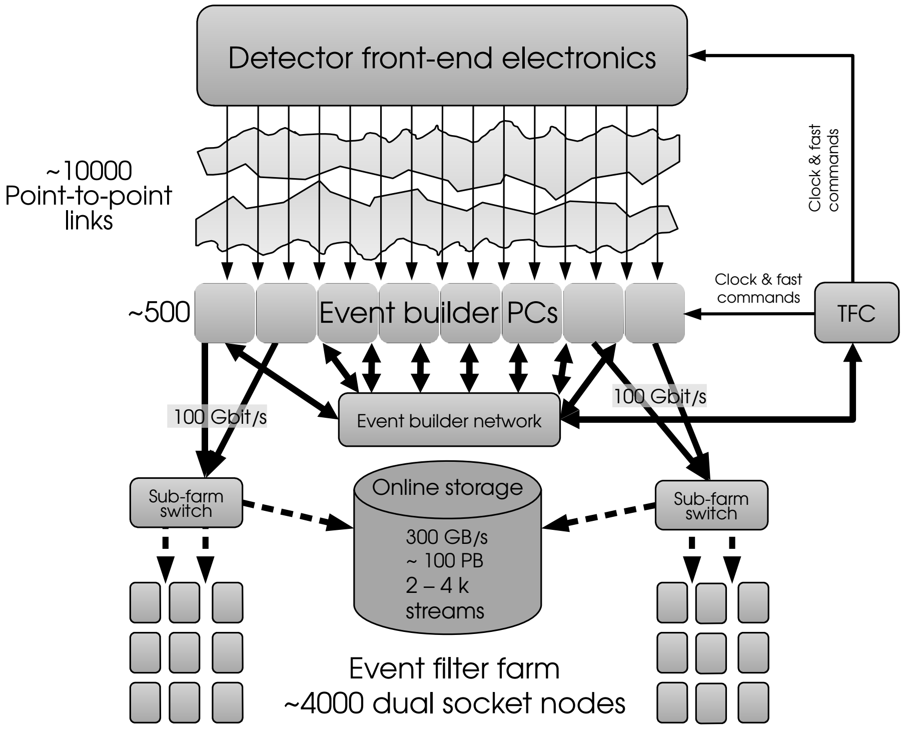


Figure 1.12: Overview of the LHCb upgrade data acquisition system.

data buffer of the event builder, ready for the event building stage.

1.2.2 *Event building*

The process of event building consists in aggregating fragments originating from subdetector signals into one coherent event, which is needed to be able to analyze and filter the event in a posterior step. Multi-event packets originated in the readout stage flow from all *readout units* into a single *builder unit*. The receiving builder unit is chosen following a round-robin selection, thus load balancing the data transmissions and evening the throughput requirements across the builder units. This imposes an all-to-all data dependency between the readout units and the builder units. Once full events are built, they are further sent to the event filter farm for reconstruction and event selection.

Since the data flows in a single direction, it is possible to *fold* the system, by which each node behaves both as a readout and a builder unit, and the full-duplex capacity of each link is used. Figure 1.12 depicts this with a two-sided arrow between the event builder network and the event builder PCs, that act both as readout and builder units. Figure 1.13 represents the data dependencies of an event builder node. Data are fed from the front-end electronics into the PCIe40 readout card inside event builder PCs. The TFC synchronizes and controls the behavior of said cards. MEPs produced by the TELL40 inside the PCIe40 are fed into a pinned memory location, ready to be sent to a receiving builder unit through the event builder network. Each PC acts both as a sender and a receiver in this scheme. Events are built inside each event builder node, and are finally sent out to the event filtering stage.

Each PC will sustain the population of the MEP buffer, sending and receiving data to the event builder network, operating on the data and sending the data out to the event filter farm. The expected data throughput required in memory for each builder node is around 500 Gb/s. Since there are no data reductions in intermediate steps, the bisection bandwidth of the system is a constant 40 Tb/s until the event filtering stage.

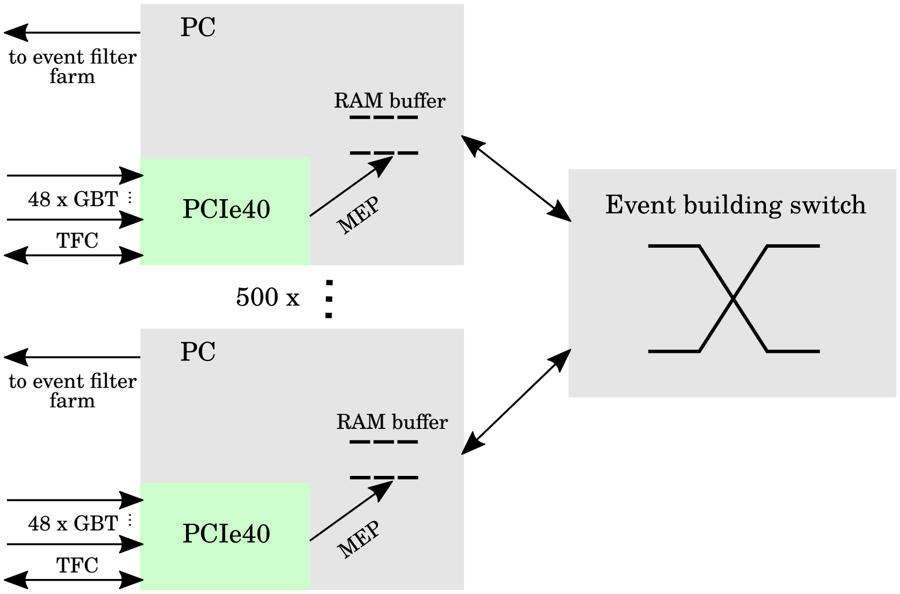


Figure 1.13: Data dependencies of a bidirectional event builder.

The isolated nature of the event building network permits using any high-performance network technology with no regard of interoperability with other networks. An event building simulation software across network technologies *DAQPIPE* [17, 18] has been developed and is currently being used to evaluate the network technology and specifics of the traffic scheduling.

In spite of the challenging nature of the presented setup, a full demonstrator was developed with early incarnations of

100 G technology as of 2014 [14]. More compact solutions may yield a better price performance ratio by condensing the functionality of two event builders in a single node. The choice of the event builder server directly impacts the usability of hardware accelerators (cref. section 10.3) in the event builder as a function of the available PCIe slots, slot width and bandwidth.

1.2.3 *Event filtering*

Built events from the event builder stage will be sent to an event filter farm. The event filter farm is in charge of reconstructing the events and selecting the interesting ones. Event selection or *triggering* will be performed in two stages in software: The *High Level Trigger 1* (*HLT1*, cref. section 1.3) performs a selection based on PV displacement, momentum and optional muon identification. The available storage in each event filter node will be used as temporary storage for the output of HLT1. A second software trigger stage *High Level Trigger 2* (*HLT2*) will be processed parasitically during the execution of HLT1, and in the downtime periods of the detector (while it is not taking data), in a process often described as *deferred triggering*.

The reduction in data rate from the combined event selection of HLT1 and HLT2 is estimated to be of a factor 1 000. Interesting events are transformed into self-contained and compressed *event signatures*, and are sent to the Online storage and kept temporarily. Event signatures are further sent to the IT storage infrastructure and to the Grid in order to perform physics analyses. The infrastructure of the IT data center or the Grid will not be discussed as they are out of the scope of this thesis.

The event filter farm will be composed of up to 4000 servers with dual socket *x86-64* architecture-based processors. Only a subset of those servers will be newly acquired. Legacy servers based on the *Sandy Bridge* architecture onwards will also be supported by the Online system and the High Level Trigger software.

1.3 THE HIGH LEVEL TRIGGER

LHCb software is written using the Gaudi framework [19], a framework for building High Energy Physics-oriented applications. The two applications that are related to this thesis are the online trigger application *Moore* and the offline trigger application *Brunel*. Gaudi abstracts the creation of software into algorithms, tools and component libraries. Gaudi applications are steered through Python processes that configure the application, including the sequence of algorithms to be run and any exposed options.

The sole trigger of the upgrade LHCb data acquisition system will be a software trigger known as the High Level Trigger subdivided in two stages, HLT1 and HLT2. HLT1 reconstructs the subdetectors involved in the tracking system and muon stations, whereas HLT2 is a more precise full-detector reconstruction and selection. A schematic of the main processes involved in HLT1 and their data dependencies are shown arrowed in Figure 1.14. The HLT1 must process the entire 30 MHz of events and perform a 30:1 data selection.

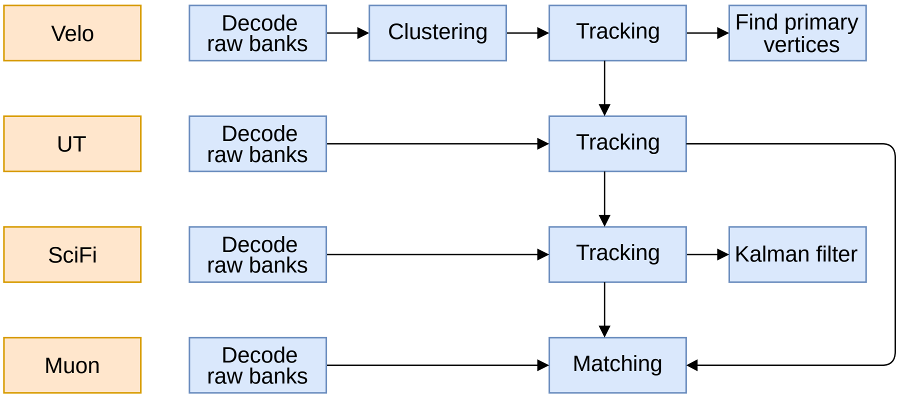


Figure 1.14: Main algorithms of High Level Trigger 1 sequence (HLT1).

An overview of the HLT1 processes is presented:

- **Velo reconstruction** – Raw Velo data are first decoded into 8-pixel containers known as *superpixels* (SPs). Only fired SPs are received as raw data. The problem of Velo clustering consists in transforming the input data into hits in the Velo subdetector, where each hit is represented with its $\mathbb{R}^3$ coordinates and a unique identifier. The problem involves extracting information from sets of 8-connected pixels, a version of *connected component analysis*. Particle trajectories or *tracks* are then reconstructed from hits. The LHCb magnet does not influence the Velo subdetector, and therefore Velo tracks are straight lines. Velo tracks serve as seeds for the subsequent subdetectors reconstruction.

- **Primary vertex finder** – The vertices where particles originate after the LHC beam collisions are known as *primary vertices*, as opposed to vertices from particles originating in decays. The primary vertex finder combines forward and backward tracks found in the Velo reconstruction, to reconstruct the primary vertices of individual collisions.

- **UT reconstruction** – Raw UT data are decoded into UT hits. The UT reconstruction sorts the UT hits to facilitate finding compatible hits. Velo tracks are extrapolated into the UT planes, and a minimum of three hits in three different planes are required to form a UT track from a Velo track. UT tracks are fitted using a parabolic trajectory, incorporating a slight deviation introduced by the weak effect of the LHCb magnetic field in the UT.

- **SciFi reconstruction** – Raw SciFi data are decoded into pre-sorted and pre-clustered SciFi hits. Only long tracks are reconstructed at the HLT1 stage, by extrapolating Velo and UT tracks into the SciFi stations. For UT tracks, the momentum measured in the UT subdetector is used to obtain a better estimate of the deviation of the particle through the magnetic field. The reconstruction of SciFi long tracks faces computationally expensive problems such as the estimation of particle trajectories through a magnetic field, detector inefficiencies and fake track reductions.

- **Kalman filter** – The Kalman filter is a software estimator widely used in literature to estimate objects trajectories (see chapter 7). In HEP it is commonly used to estimate particle trajectories with a precise error covariance matrix that integrates both the mathematical model of trajectories and the uncertainty due to scattering. A simplified Kalman filter is applied to reconstructed tracks for an accurate estimation of their *impact parameter*, and to reduce fake tracks.

- **Muon reconstruction** – After the decoding of muon stations raw data, the identification of muons can be performed following one of two strategies: Either taking the UT or long tracks as input.

- **Data reduction algorithms and selections** – Configurable data reduction algorithms permit to filter out events according to factors like their detector occupancy or the impact parameter of tracks. Finally, selection algorithms filter events that have been fully reconstructed, and decide whether to keep or discard them according to predefined criteria in accordance with the LHCb physics program.

1.3.1 *The LHCb software upgrade*

The present chapter conforms the design ambitions of the LHCb upgrade program, as described in the *Technical Design Report* [14].

The LHC detectors such as LHCb evolve during their lifetime. Hardware replacements may be performed during technical stops, improving the resolution and performance of the detector. The experience acquired in the first runs of the detectors revealed possible improvements, which may impact the physics goals of the experiment. The TDR describes the indended upgrades of the detector making educated guesses on the hardware available and the evolution observed in the last years.

In particular, the LHCb upgrade removes the *hardware level trigger* which was used in Runs 1 and 2 of the detector. The hardware level trigger did a 30:1 selection of the events according to partial subdetector information. In contrast, the upgrade software level trigger (HLT) performs a selection based on full detector tracking, which allows to more efficiently select interesting events. In combination with the increased luminosity, the projected data increase to be processed in software is estimated to be $40\times$.

The required hardware and software improvements to cope with the increase in data rate were predicted in the TDR, with associated unknowns in terms of computing infrastructure and the performance of the software codebase on upcoming hardware. However, the code performance and compute resource cost were underpredicted. Progress was closely monitored as algorithms for the new detectors were being written, and in 2016 it was determined that there was a shortfall of between 6 to 10 $\times$ less performance than expected.

This figure was far beyond any of the built in contingencies in the budget. As a consequence, the software codebase was benchmarked and profiled, and new algorithms were developed where necessary. In addition, different computing models and alternative hardware architectures were explored to see if they could be exploited within the constraints of the planned system. The LHCb physics reconstruction became therefore a real-time software challenge, whereby the design performance of the system would have to be met within the hardware constraints.

2 PARALLEL COMPUTING

Traditionally, software has been written for serial computing [20]. Serial programs are broken down into a discrete list of instructions, and are run sequentially on a single processor. Determinism is preserved by guaranteeing the order of execution of instructions, strictly following the logic specified in the algorithms composing the program.

The performance of a serial program is determined by the time it takes to run, or runtime. The runtime of a program depends on factors related to how the program is written and on the processor it runs on. Comparisons between program runtimes are measured by comparing the slower program to the faster program as equation 2.1 shows. The metric name is *Speedup*, and is measured in a unitless manner in × (*times*).

$$\text{Speedup} = \frac{t_{\text{slower}}}{t_{\text{faster}}} \tag{2.1}$$

During the last decades, the number of transistors in integrated circuits has increased exponentially. This observation, known as *Moore's Law* [21], has held true since the early days of integrated circuits in the 60s. The scale of integration is classified into *small-scale*, *medium-scale*, *large-scale* and *very large-scale*, referring to the number of *logical gates* in an integrated circuit. Since the 80s, the term *very large-scale integration* (VLSI) has been coined and maintained to refer to any chip with more than 10 000 gates [22]. *Dennard's Scaling Law* [23] can be seen as a consequence of Moore's Law. It states that as transistors shrink in size, they become faster, consume less power and are cheaper to produce. Modern computer processors are chips that consist of billions of transistors.

Three areas that have lowered serial program execution runtimes due to higher transistor integration are clock frequency, execution optimization and cache. The clock frequency determines the time it takes the processor to complete an execution cycle. It holds a linear dependency with the runtime of the program. Execution optimization refers to a set of techniques to minimize the amount of cycles it takes to execute instructions, also known as *cycles per instruction* (CPI). Cache is an on-chip memory with fast access latency and throughput. Modern processors have a hierarchically structured cache into several levels of increasing speed and decreasing size.

Higher clock frequencies, better execution optimizations and bigger and faster caches led to a steady increase in serial programs performance for several decades. However, the trend stopped around 2004 due to heat and power consumption issues. Figure 2.1 shows the evolution of transistor integration, clock speed and power consumption. The red tendency line shows the performance as a function of clock speed flattening around 2004. H. Sutter qualified this phenomenon with the phrase *the free lunch is over* [24], noting that efficient programs would have to evolve and not rely on serial computing alone.

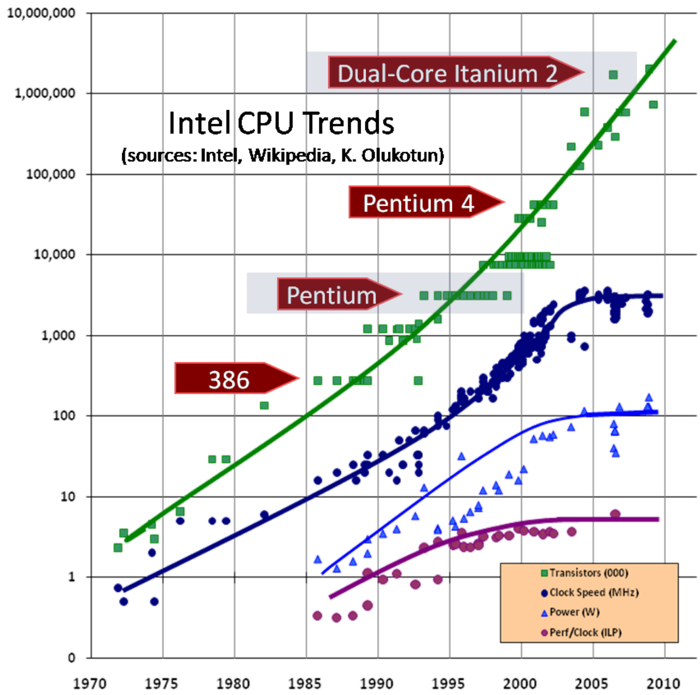


Figure 2.1: Evolution of transistor count, clock speed, power consumption and performance per clock speed. Image from [24].

Parallel computing consists in the simultaneous execution of sections of a problem. Parallel processors are composed of multiple cores. *Multi-core* processors are equipped with up to tens of cores, whereas processors with hundreds of cores or more are called *many-core* processors. Problems that have sections that can be parallelized, also called *parallelizable*, can be executed more efficiently on parallel processors.

The runtime of a program executed in a parallel processor is measured as the elapsed real time or *wall time* of the program execution, irrespective of the number of computing resources employed to solve it. Amdahl's law [25] establishes a relation between the speedup attainable on a program of a fixed size, with a parallelizable fraction p and n processing units, shown in eq. 2.2. This relation is referred to as *strong scaling*. The number of resources employed affects the runtime of the program, and the parallelizable fraction plays a determining role in the scale of the speedup, as shown in Figure 2.2.

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}} \tag{2.2}$$

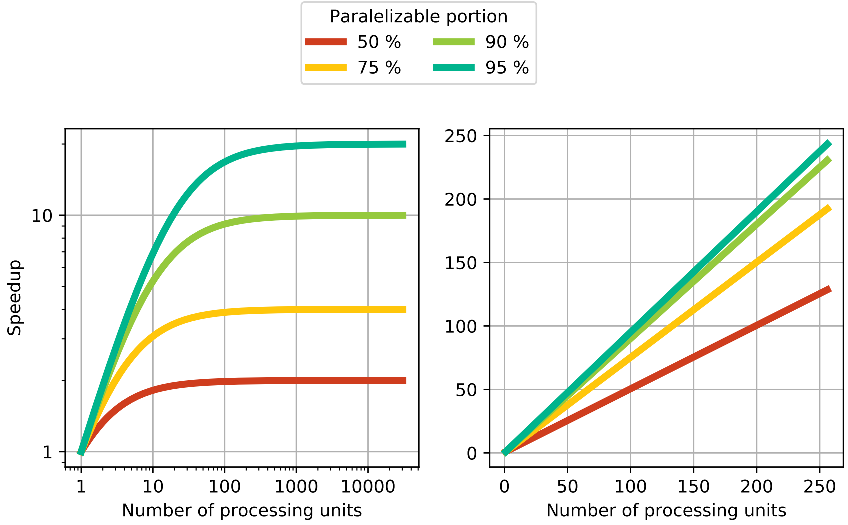


Figure 2.2: Amdahl's law (left) and Gustafson's law (right).

A different perspective can be obtained by fixing the execution time of a process. Gustafson's law [26] defines the theoretical speedup attainable on a program of a fixed execution time, with a parallelizable fraction p and using n processing units, shown in eq. 2.3. Gustafson argues that this relation, known as *weak scaling*, takes into account that as the computing resources increase, more aspects of a problem may be analyzed which would otherwise not be considered. The scaling of both laws is depicted in Figure 2.2. Amdahl's law is shown with a logarithmic scale (left), and Gustafson's law with a linear scale (right).

$$Speedup = n + (1 - n) \cdot (1 - p) \tag{2.3}$$

During the last twenty years, processor manufacturers have transitioned to providing an increasing number of cores on processors. In order to make an efficient use of multi- and many-core processors, codebases likewise must transition to using programming models and algorithm designs that take into account the underlying parallel hardware.

2.1 TYPES OF PARALLEL PROCESSORS

Flynn's taxonomy [27] divides processors into four categories as a function of the number of concurrent instructions streams and the data instructions the processor operates on, as shown in table 2.1. SISD processors issue one instruction operating on one data at a time. Old desktop machines and mainframes fit in this category. In the MISD category, a processor issues multiple instructions operating on the same data at a time. This model is used in fault tolerant environments. SIMD processors issue a single instruction operating on several data at a time. The SIMD model is implemented in modern processor cores through vectorization or multiple threads (SIMT). Finally, MIMD processors operate on several data independently, issuing multiple instructions operating on multiple data concurrently.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: Flynn's taxonomy.

Modern desktop and server processors are realizations of MIMD processors. Various degrees of parallelism [20] co-exist and must be considered in order to make an efficient use of parallel processors. Parallelism can be obtained at the level of *data*, *instructions*, *threads* and *processes*.

Data parallelism is realized in modern hardware through functional units dedicated to that end. The `x86-64` architecture is a family of 64-bit processors[1]. Multiple data execution is achieved in these processors through the use of wider registers and an extended *Instruction Set Architecture* (ISA). The sets of extensions in `x86-64` processors are shown in table 2.2. These extensions are called *vector extensions* and refer to the processor capability to execute a number of instructions (*vector width*) onto its specialized functional units (*vector units*). The availability of an extension can be checked against the processor flags. Similar extensions exist on other hardware architectures, such as *Neon* in ARM processors or *Altivec* in Power processors.

SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT	128-bit
AVX, AVX2	256-bit
KNC, AVX512-family	512-bit

Table 2.2: Vector extensions and their corresponding register widths.

Instruction-level parallelism (ILP) refers to the number of instructions executed in parallel in a processor. Two metrics are associated with this type of parallelism. Sequential processors measure the number of cycles per instruction (CPI, lower is better), whereas for parallel processors instructions per cycle (IPC, higher is better) is used instead. Both metrics are equivalent. It is uncommon for a sequential processor to achieve a CPI smaller than one, and for a parallel processor to achieve an IPC smaller than one in a program is usually an indication the code is not properly optimized.

The following concepts are associated with ILP:

- *Pipelining* – Instruction execution on a processor is a task that can be divided in stages, which are performed by functional units or *ports*. This division permits to pipeline several instructions, such that a different stage is being performed on each instruction. Primitive processor designs divide instructions in *fetch* or *issue* (the next instruction to be executed is retrieved), *decode* (the instruction is identified and the operands are decoded), *execute* (arithmetic operations are carried out), *memory access* (read or store instructions are executed) and *write back* or *commit* (the operation is finally performed). An efficient processor design maximizes the potential parallel use of its ports, accounting for the latency of each stage.

[1] `x86-64` has been implemented by various manufacturers, most commonly Intel and AMD.

Pipelining requires knowing or determining the order of execution of instructions in advance. *Jump* instructions break the linear execution order. *Branch prediction* units predict the result of jump conditions and keep a track of last results, improving processor utilization. Certain ISAs allow developers to impact the branch prediction unit by specifying the *likeliness* of a jump to be taken.

- *Multiple issue* – Superscalar processors allow to issue several instructions at a time. This design choice requires a duplication of ports across all stages, and verifying the instructions can be completed in parallel, checking no data dependencies exist. The verification may be done at runtime, in *dynamically scheduled* processors, or at scheduling time on processors that require *static scheduling*, such as VLIW processors.

The requirements to exploit ILP are hard to predict and processor dependent, and often the task to optimize ILP is left to hardware developers.

Thread-level parallelism (TLP) consists in specifying different units of work from a control perspective, to be executed independetly in processor cores. In contrast with ILP, the control of the work assigned to each thread is left completely to the developer. It is worth distinguishing several types of TLP:

- *Fine-grained multithreading* – The developer creates and joins *threads* explicitly, and assigns the work each thread will do. The underlying hardware executing the code can also be specified. This method grants a fine-grained control to the developer of the computing resources utilized and the memory assigned to each resource. In particular, it permits the developer to select processors in multi-socket systems, and cores in processors with simultaneous multithreads[2], which impact the performance of applications.
- *Task-based multithreading* – The work is divided into tasks, which are then automatically mapped to processor cores. The developer has no control of the assignment of the task to a specific thread. *Dependency graphs* permit to specify a list of data and control dependencies between tasks,

[2] Simultaneous Multi-Threading (SMT) or Hyperthreading is a hardware technology of some processors that exposes multiple logical cores for each physical hardware core. This technology allows processors to prevent stalls by executing multiple threads with the ports of a single core.

such that tasks will be executed as their dependencies are fulfilled.

Finally, **process-level parallelism** (PLP) refers to expressing the work into several independent processes, which are scheduled and managed by the operating system. A process can be assigned a precedence, increasing (decreasing) the probability it will be scheduled. Processes are a coarser-grained level of parallelism, which may be assigned to a specific core or processor. PLP can also be implemented across multiple computing nodes, and message passing libraries such as MPI facilitate the instantiation and communication between processes.

The operating system can interact with processes through signals. When a signal is caught in a process, the execution context is saved and the context is shifted to a function handling the process signal. Once the signal has been handled, and if the process did not terminate, the control is returned to the previously executing context, and computation is resumed.

The four levels of parallelism discussed offer orthogonal considerations when developing a parallel program. A more in-depth discussion can be found in [20].

2.2 MEMORY

During the last decades, memory performance has not progressed at the same pace of that of processors. Figure 2.3 depicts this effect, normalized to the performance of processors and memory to 1980, where the performance axis is in logarithmic scale. The performance increase of processors until 2005 was of 1.25× per year, whereas from there after it is 1.20×. In contrast, memory has steadily increased its performance at a rate of 1.07× per year.

The gap in performance between memory and processors has widened over the years, and therefore memory access-related issues are increasingly important to produce efficient programs.

Computer memory is organized in a hierarchical manner, where memory capacity and latency are directly correlated. Processors use low-latency registers to store operands of instructions, with a varying amount of registers available by processor model. On-chip cache memory is organized in layers, identified by its level, commonly L1, L2 and L3. Dynamic random access

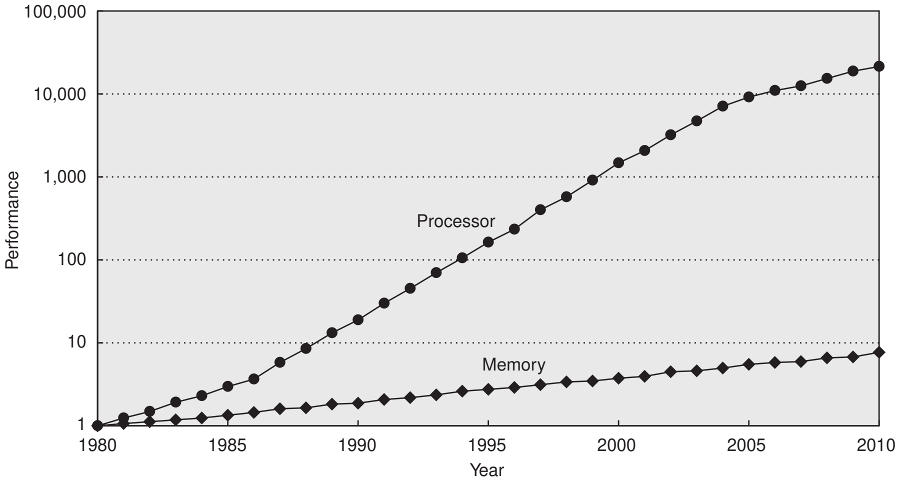


Figure 2.3: Memory performance versus processor performance in the last decades. Image from [28].

memory (DRAM) can hold entire applications, and finally I/O devices are used as a persistent storage system. Common sizes and speeds of all these memories are shown in Figure 2.4.

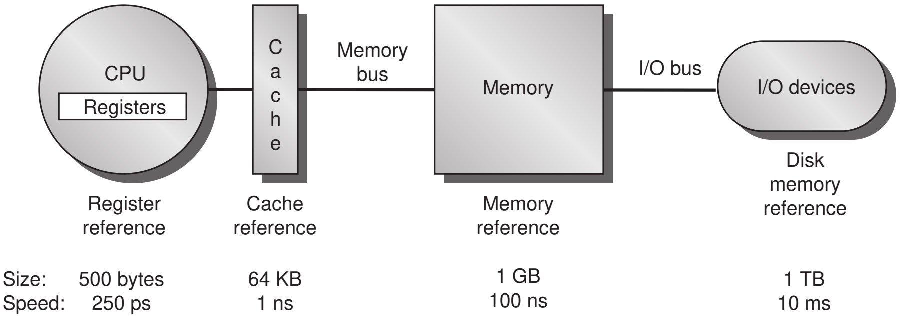


Figure 2.4: Memory hierarchy. Image from [28].

The difference in latency for accessing the next levels in the hierarchy are several orders of magnitude, and it is desirable to trigger memory accesses only when it is necessary. The principle of *locality* is an empirical result that states that when a memory location is accessed (1) it is probable that it will be accessed again shortly, an effect also known as *temporal locality*, and (2) it is probable that increasing memory locations in the immediate vicinity are also accessed, also known as *spatial locality*. The effects of locality are exploited in cache memories. When memory is read, a contiguous chunk of memory of a predetermined size is retrieved, referred to as *cache line*. Memory is kept in cache, and the upper levels of cache and memory are updated only when necessary, according to the cache write policy.

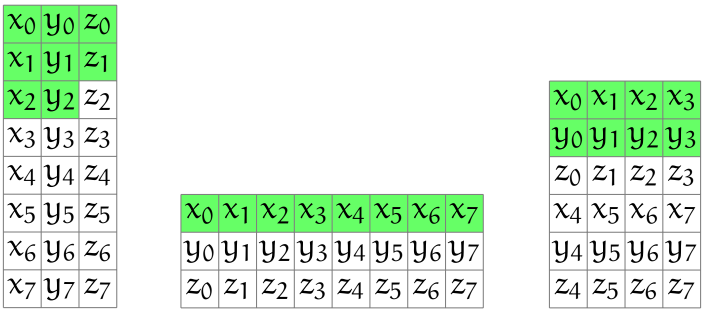
Since cache memories cannot store a complete map of the application's memory, the position of a memory location is transformed to a position in cache. The destination location is determined by shifting the memory location as many bits as required by the cache line size and retrieving the least significant bits. The resulting location may collide with pre-existing data in cache, which requires removing the pre-existing data from cache and updating the upper memory level according to the write-policy. A number of collisions can be sustained by configuring the cache memory with an *n-way associativity*, which results in a trade-off between memory space and resiliency for memory location collisions.

An indicator of how efficient cache memories are being utilized is the cache *hit* and *miss* rates of a program. The expected hit and miss rates depends theoretically on the program under analysis. Both instruction and data caches exist, although the former is usually ignored as a program's code undergoes many difficult to predict transformations prior to resulting in the list of instructions visible by the processor.

Memory architectures influence memory access times. *Uniform memory access* (UMA) systems guarantees an equal access time to a shared memory across all processing units. In *non uniform memory access* (NUMA) systems, the time to access memory depends on where the memory resides. NUMA systems may be non-uniform at different levels, such as cache or main memory. In some cases control is given to the developer to manage and pin memory for processes in order to avoid inter-domain memory accesses.

Memory structures play a role in the efficiency of memory operations. In an *array of structures* (AOS), each structure stores its members contiguously. Further elements in the array are stored in this manner, and each structure in the array is stored next to each other. In contrast, an *structure of arrays* (SOA) stores same structure elements in the array contiguously. *Array of structure of arrays* (AOSOA) is a mix between the two previous structures, where a stride size s is chosen. Array elements are then stored in SOA in groups of s elements, resulting in an array of contiguous SOA structures.

Figure 2.5 shows an AOS, SOA and AOSOA of stride 4, for the structure $\{x, y, z\}$. A sequential code would benefit from the AOS datatype, as accesses to one parameter are likely to be followed by accesses to the other parameters by the spatial locality principle. Data-parallel architectures would benefit from SOA – if the vector unit performs the same operation across datatypes, it will fetch all values with one cache line data access. AOSOA datatypes are used in conjunction with a particular vector width, to further optimize the data access pattern by aligning the data boundaries with the vector width. The fetched data with a single access to x_0 is shown in green for all configurations in the figure, assuming the structure elements are double precision floating point numbers, and a cache line size of 64 bytes. If executing on a vector width of 4, (c) is likely the most advantageous, since the x and y values will be populated in cache.



x_0	y_0	z_0
x_1	y_1	z_1
x_2	y_2	z_2
x_3	y_3	z_3
x_4	y_4	z_4
x_5	y_5	z_5
x_6	y_6	z_6
x_7	y_7	z_7

(a)

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
z_0	z_1	z_2	z_3	z_4	z_5	z_6	z_7

(b)

x_0	x_1	x_2	x_3
y_0	y_1	y_2	y_3
z_0	z_1	z_2	z_3
x_4	x_5	x_6	x_7
y_4	y_5	y_6	y_7
z_4	z_5	z_6	z_7

(c)

Figure 2.5: From left to right: AOS, SOA and AOSOA of stride 4 for structure $\{x, y, z\}$. In green, cache line access triggered by access to x_0, assuming x, y and z are double precision floating point numbers, and a cache line size of 64 bytes.

When data is not contiguous and requires accesses to multiple memory locations, the terms *gather* and *scatter* are used. Gather and scatter operations are often necessary in data parallel workloads, but an abuse of these operations may lead to an inefficient program. Memory alignment also affects data parallel architectures, which distinguish *aligned data accesses* from *misaligned data accesses*, with different operation microcodes for each. Aligned data accesses are preferred and lead to faster code, where possible.

2.2.1 *The Roofline model*

The *arithmetic intensity* of a program is its number of floating point operations divided by its number of bytes loaded and stored into memory, and is measured in $\frac{\text{FLOP}}{\text{B}}$. It is possible to characterize the performance attainable on a processor, in terms of FLOP/s, as a function of the arithmetic intensity. Conceptually, if the arithmetic intensity of a program is below a threshold defined by the peak processor performance and the fastest memory where the data is expected to reside, then the performance of the program is bound by memory bandwidth rather than by the peak processor performance. Given the gap between memory and processor performance increase in the last decades, it is likely that more and more programs become memory bound.

The *Roofline model* [29] is a visual performance model that characterizes in a condensed manner the peak performance attainable by a processor under various conditions, and the empirical performance obtained by one or several programs. Figure 2.6 shows the Roofline model of processor Intel Xeon Haswell E5-2683. The model is composed of various *roofs*, that depict the peak performance with respect to processor technologies (compute bound), and the peak performance when a program is subject to a particular memory bandwidth (memory bound). Three roofs related to the Haswell processor are shown. A program consisting solely of double precision (DP) floating point *scalar* arithmetic would have a theoretical peak performance of 48.38 GFLOP/s. If the program is vectorized, the theoretical peak performance would be 204.04 GFLOP/s. If the program uses *fused multiply-add* (FMA) instructions, the theoretical peak performance would be 391.07 GFLOP/s.

On the other hand, four roofs are defined that are dependent on memory and cache bandwidth. If the program solely relies on accesses to main memory (DRAM), which has a bandwidth of 44.76 GB/s, then the peak performance is capped by the relation between arithmetic intensity and bandwidth. Similarly, the relation between peak performance and arithmetic intensity of the program if all memory accesses occur in any of the cache levels are also depicted.

For instance, let us consider the pseudo-code for two different programs, shown in listing 2.1. The first is *saxpy*, a well known benchmarking program. Let us assume x and a to be input variables, y to be an input and output variable, and both x and y to have six elements. The second example is the Cholesky decomposition for 3×3 matrices. Let us assume that C is an input array of six elements, and L is an output array of six elements. All variables and operations shown are using double precision,

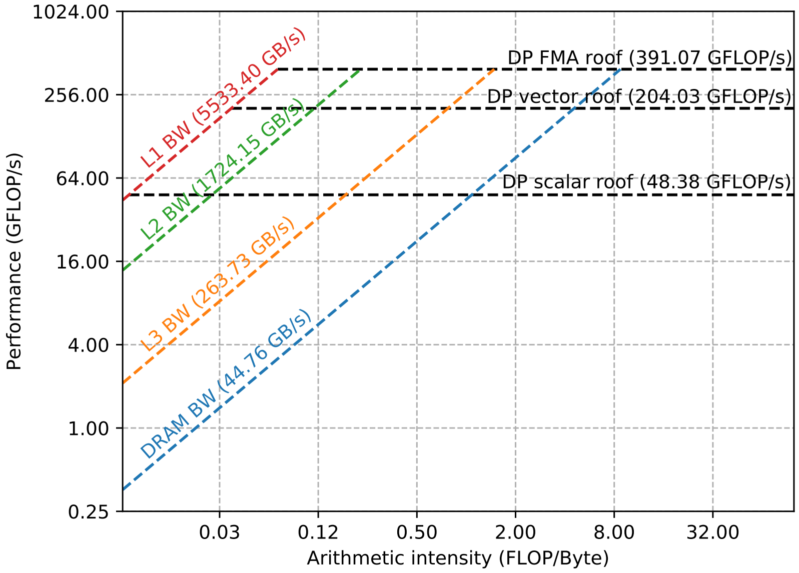


Figure 2.6: Roofline model characterizing processor Intel Xeon Haswell E5-2683.

and FMA units are used, whereby multiplications followed by additions or substractions count as a single operation. Table 2.3 characterizes both sample programs.

Program	Loads (B)	Stores (B)	FLOP	Arithmetic intensity (FLOP/B)
saxpy	$13 \cdot 8$	$6 \cdot 8$	6	0.04
cholesky3x3	$6 \cdot 8$	$6 \cdot 8$	16	0.17

Table 2.3: Arithmetic intensity of sample programs.

The arithmetic intensity obtained for `saxpy` is 0.04 FLOP/B, whereas for `cholesky3x3` it is 0.17 FLOP/B. If the programs would be executed on the server equipped with the Haswell E5-2683 processor, the peaks would be as shown on Figure 2.7. Assuming data is read and written to main memory, the programs would be memory bound to 1.8 GFLOP/s and 8 GFLOP/s respectively, which represent 0.5% and 2% of the maximum performance the processor could deliver.

The creation of a Roofline model requires processor-specific and program-specific knowledge. The roofs of a processor require knowing its peak performance under several conditions, and the memory throughput of its available memories. It is possible to obtain these quantities theoretically or by employing simple benchmark programs and determining the roofs empirically. Similarly, the arithmetic intensity of a program can be

```
void saxpy(double* x, double* y, double a) {
  y[0] = x[0] * a + y[0];
  y[1] = x[1] * a + y[1];
  y[2] = x[2] * a + y[2];
  y[3] = x[3] * a + y[3];
  y[4] = x[4] * a + y[4];
  y[5] = x[5] * a + y[5];
}

void cholesky3x3(double* C, double* L) {
  L[0] = sqrt(C[0]);
  double L_inv = 1.0 / L[0];
  L[1] = C[1] * L_inv;
  L[3] = C[3] * L_inv;
  L[2] = sqrt(C[2] - L[1] * L[1]);
  L_inv = 1.0 / L[2];
  L[4] = (C[4] - L[3] * L[1]) * L_inv;
  L[5] = sqrt(C[5] - L[3] * L[3] - L[4] * L[4]);
}
```

Listing 2.1: Sample programs.

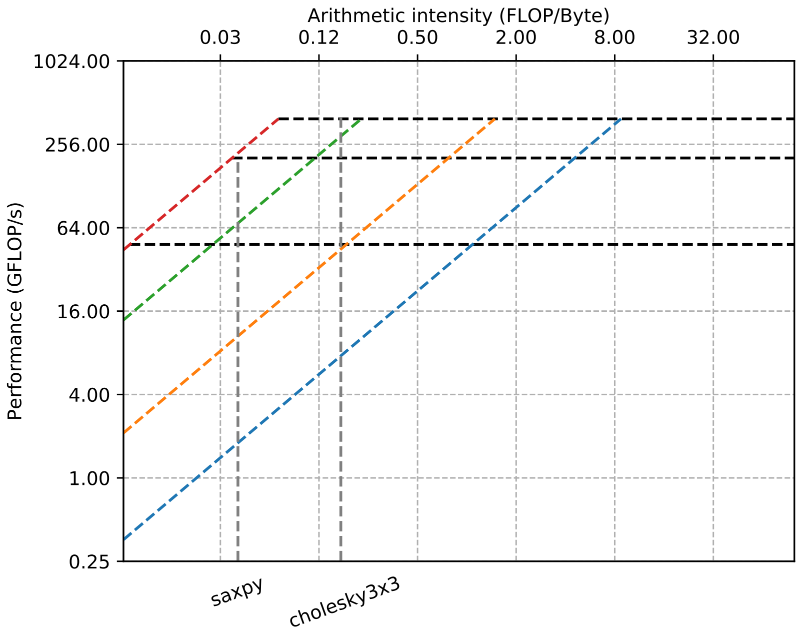


Figure 2.7: Roofline model of Intel Xeon Haswell E5-2683 populated with `saxpy` and `cholesky3x3` arithmetic intensities.

determined either theoretically or by reading the assembly code of a compiled program. Complex algorithms involving branch instructions are harder to quantify both in terms of arithmetic intensity and peak throughput.

The Roofline model is a visual tool that permits to abstract various technical concepts in a single image. It can be used for profiling what the status of the program is, and what is preventing the program to obtain a better performance. It also allows to make predictions to either different hardware, or different arithmetic intensities, such as the result of moving to different floating point precisions.

2.3 GRAPHICS PROCESSING UNITS

Graphics Processing Units, or GPUs, are parallel processors specialized to deal with image and video processing. In the early 2000s, the graphics processing pipeline executed on a GPU was homogeneized in several stages dealing with geometrical and rasterization calculations [30], and it was allowed to program parts of the pipeline using *shaders*, small programs which would run on GPU hardware.

Shaders were initially written in specific GPU domain languages. Through the use of shaders, GPUs became programmable not only for graphics processing, but for any kind of application. The use of GPUs for general purpose computing, or *GPGPU*, has expanded in recent years. GPUs can be programmed for general purpose computing currently through language extensions like CUDA, OpenCL [31] or HIP, and GPU-specific development environments consisting of profilers, debuggers and development tools are available. A GPU requires to be executed with a host CPU, whereby the GPU is used to accelerate a portion of the computation. GPUs are being adopted as computing accelerators for various workloads, and hundreds of the supercomputers in the Top 500 [32] are equipped with GPU accelerators.

For the purposes of this thesis, the author will focus on describing the general architecture of modern NVIDIA GPUs, which for the most part are applicable to other commercially available GPUs. A modern NVIDIA GPU consists of dozens of *Streaming Multiprocessors*. Figure 2.8 shows a schematic of one such processor from the *Fermi* architecture. The memory of the processor (in blue) is composed of an instruction cache, registers, a configurable L1 cache and a uniform cache. Each control unit (in orange) manages *warp of threads* in groups of 32 threads. The *warp scheduler* and the *dispatcher* assign computation to the computing resources of the GPU (in green). Each Streaming Multiprocessor is equipped with two sets of 16 CUDA cores, consisting of a floating point and an integer functional unit, which perform basic 32-bit arithmetic operations. The functionality is complemented by special functional units (SFUs) which execute trascendental instructions like sin, cosine, reciprocal and square root. The load / store (LD/ST) units allow to transfer data back and forth to cache and DRAM.

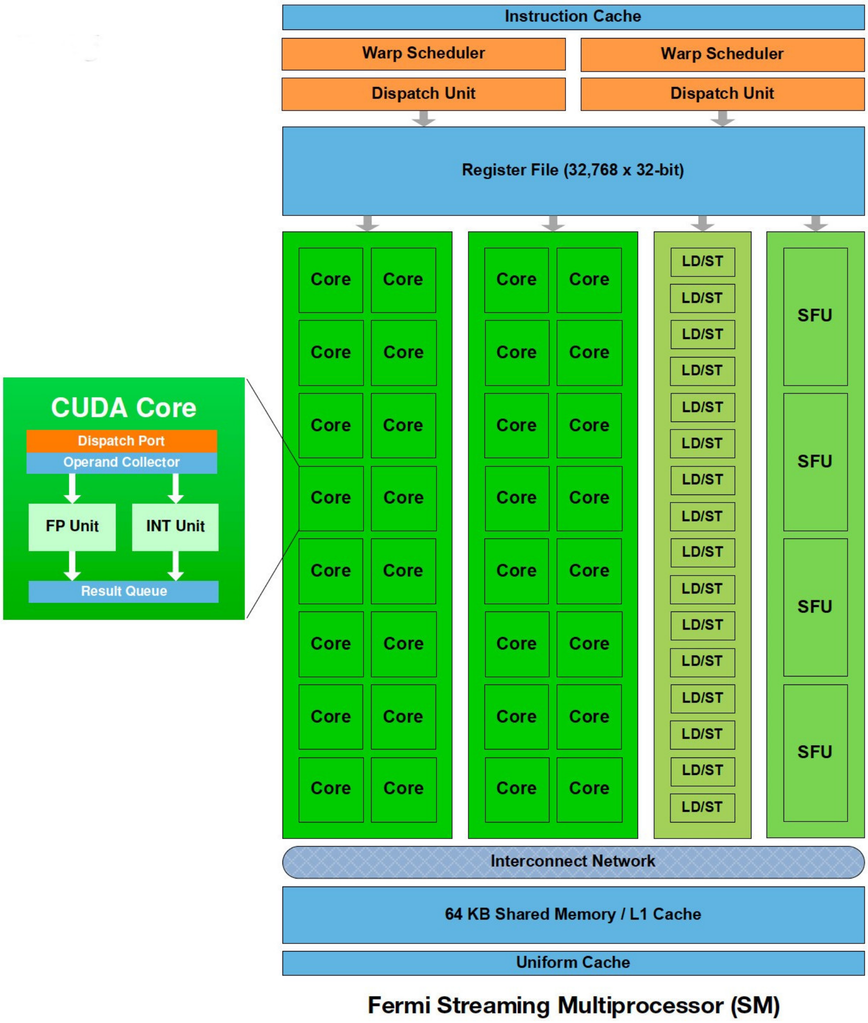


Figure 2.8: Schematic of a *Streaming Multiprocessor* inside a GPU.

GPU processors are of the Multiple Instruction Multiple Data type in Flynn's taxonomy. Each individual warp scheduler dispatches an instruction on up to 32 threads at a time, in a model known as *Single Instruction Multiple Thread* (SIMT).

The CUDA programming model divides the work in two groups. *Blocks of threads* are composed of at most 1024 threads, and execute on a single SM. Threads within a block can communicate with each other, and can be synchronized with a thread barrier. A *grid of blocks* is meant to execute independent workloads across the GPU. Any CUDA function is executed by a grid of blocks of threads, and is referred to as a *kernel*. Several kernels may be executed concurrently on the GPU in separate *CUDA streams*, and the resource usage is determined by a dynamic GPU scheduler. Both the grid of blocks and the block of threads may be specified using either a 1-, 2- or 3-dimensional array. The indices of the executing thread and block can be accessed at all times through specialized keywords.

GPUs have a complex memory hierarchy composed of several layers. Figure 2.9 shows the thread and memory organization of a GPU. For each execution context (left), the available memory buffers specific to that context are shown (right).

The fastest memory available on GPUs are thread registers. The number of registers in a thread vary depending on the GPU model. When the number of registers available is exceeded, *register spilling* occurs, whereby registers are stored in local memory. *Local memory* is only available to the thread writing it, and it is stored physically on global memory (DRAM). Local memory utilization is determined by the compiler, either when large arrays or structures are used, or when the aforementioned register spilling occurs. Local memory is cached, so the performance impact is diminished if sufficient L1 cache memory is available.

Blocks of threads have access to shared memory. Shared memory is an addressable memory that resides in L1 cache, and the amount of memory available for L1 cache and for shared memory is configurable. A total of 64 kB are available for each Streaming Multiprocessor, of which 16 kB are L1 cache, 16 kB are dedicated to shared memory, and the additional 32 kB are configurable by the developer. It is possible to choose the configuration of each specific kernel individually. NVIDIA GPUs have two cache levels. While L1 cache is available to each SM, L2 is shared between all SMs.

Global memory is the largest memory available on GPUs, dozens of GBs in capacity. Global memory is cached and available to

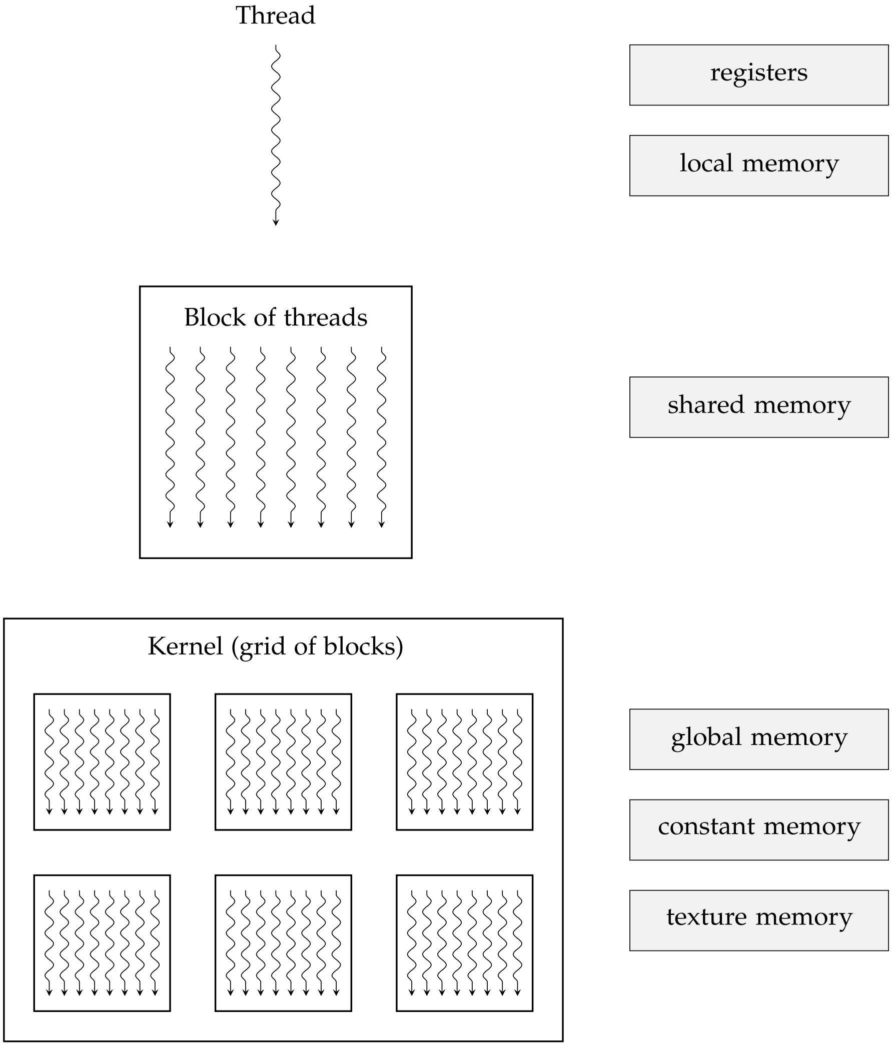


Figure 2.9: CUDA thread and memory organization.

all threads in the kernel and the host. *Constant memory* is a read-write memory for the host, but it is read-only for the GPU. A small dedicated read-only cache is available globally to cache constant memory, with a capacity of tens of kB. *Texture memory*, similarly to constant memory, is read-write for the host and read-only to the GPU. Cache locality of texture memory is defined as a 2D-lattice, whereby elements close in the lattice will benefit from cache locality, as opposed to the 1-dimensional locality behavior of other memories.

As GPU memory resides in the graphics card, any communication to and from the host must be done over the PCIe network. GPUs allow to transmit using the full-duplex capability of the channel while executing code. Therefore, it is possible to hide the latency introduced by these transmissions by creating a pipeline of two-way transmission and execution on separate CUDA streams.

Newer iterations of GPU hardware may introduce features that are exposed through updates to the CUDA language. The availability of said features in a GPU processor are determined by its *major* and *minor* number, which can be programmatically accessed, allowing developers to produce hardware-dependent code through the use of macros. An up to date list of features are available in the CUDA programming guide [33].

2.3.1 *GPUs as parallel coprocessors*

The amenability of GPUs to speed up parallel workloads in the scientific computing field has been shown on a variety of data-intensive and throughput-driven applications.

DNA sequencing is a compute-intensive field where GPUs have improved performance. Parallel sort and reduction techniques are implemented with GPUs in the framework Arioc to compute alignments of DNA sequences, where it achieves up to 10× higher throughput compared to other CPU aligners while maintaining or improving the accuracy of the results [34], [35]. A benchmark comparing various DNA sequencing algorithm including different GPU-based tools was performed by Pawar et al. including comparison against CPU-based tools. The benchmark showed how GPU delivered better throughput, where the authors mention how GPU-based tools should replace CPU ones due to the better performance [36].

Radio telescopes have DAQ systems which filter in a similar way as HEP experiments. Frameworks like Bifrost implement their algorithms with C++, Python and CUDA to process data in real-time, where the CUDA version delivers significant higher throughput in the framework [37]. A real-time pipeline for the CHIME Pathfinder radio telescope requires processing data at rates close to 1 Tbit/s, where they use a GPU-based framework for their data flow, benefiting from aggressive optimizations to cope with the data rate [38].

In High Energy Physics, several workloads have been adapted to many-core architectures. GPUs have been used for event selection and reconstruction in various environments [39, 40]. I. Kisel designed an *automata* based track reconstruction that has been widely adopted over recent years [41]. The CMS collaboration has successfully ported their vertex detector reconstruction to GPUs [42], and the ALICE collaboration has transitioned to GPU-based processing for a large part of their Online reconstruction [43].

Using hardware accelerators to speed up computations of the LHCb High Level Trigger has been attempted before. A parallel implementation of the VELO subdetector tracking was attempted for the first time on GPUs in the author's master's thesis [44]. A more fine-tuned implementation of the GPU Velo tracking was presented in the LHCb Computing Workshop in November, 2015. The implementation, written in OpenCL, offered early promising results of the amenability of GPUs to process sections of the LHCb trigger.

Through analysis and profiling of the LHCb HLT codebase it became increasingly evident that the underlying hardware architectures executing the code could be more efficiently utilized by applying various of the principles described in this chapter. The HLT codebase is *memory bound* [45], and thus a more efficient exploitation of memory access patterns, memory structures and cache accesses (see section 2.2), alongside new parallel methods, were explored. Other hardware architectures such as GPUs were also considered as alternative solutions to overcome the performance gap (see 1.3.1). A variety of problems and hardware architectures are considered in this thesis, with the intent of optimizing the workloads in the High Level Trigger application of LHCb.

Part II

PARALLEL ALGORITHMS

3 DECODING ALGORITHMS

Incoming data from the LHCb detectors are encoded in formats specific to each subdetector. Each subdetector *raw data* must be decoded prior to being usable within the reconstruction sequence. The sequential decoding routines defined in the Gaudi HLT applications leave little space for innovation, given the format must be followed within strict typecasting rules. Nevertheless, deterministic and efficient parallel realizations of these routines are far from trivial.

Figure 3.1 depicts a detail of the HLT1 decoding sequences. The contributions of this thesis are the four parallel decoding designs and implementations. For the Velo reconstruction, a parallel Velo clustering implementation has also been developed. In the figure, each box represents a different GPU algorithm, part of the containing dashed box subsequence. Blue boxes represent algorithms for which both CPU and GPU implementations have been developed.

The decoding sequences share a common pattern. The number of hits in each subdetector is necessary to allocate buffers that can hold the decoded subdetector data. Therefore, the number of hits is calculated as the first step of either decoding sequence. In the Velo case, an upper limit is obtained by requiring a distinct data pattern (cref. 3.1). The accumulated sum or *prefix sum* over data of all events under process provide the buffer size. The specifics of each subdetector decoding will be discussed in the following sections.

3.1 VELO DECODING AND CLUSTERING

Velo raw data ships all individual pixels that have fired in an event. The top of Figure 3.2 depicts the Velo modules within the beam line, and a front view of a pair of Velo modules. Each module is composed of four sensors with three chips each. The bottom of the Figure shows a detail of the three chips in any one

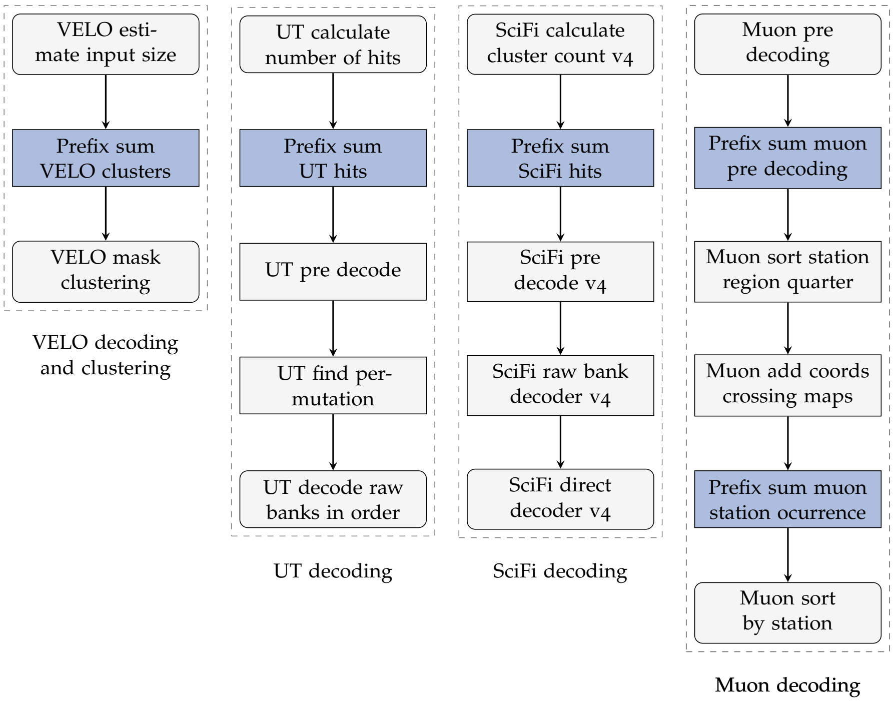

VELO estimate input size
Prefix sum VELO clusters
VELO mask clustering
VELO decoding and clustering
UT calculate number of hits
Prefix sum UT hits
UT pre decode
UT find permutation
UT decode raw banks in order
UT decoding
SciFi calculate cluster count v4
Prefix sum SciFi hits
SciFi pre decode v4
SciFi raw bank decoder v4
SciFi direct decoder v4
SciFi decoding
Muon pre decoding
Prefix sum muon pre decoding
Muon sort station region quarter
Muon add coords crossing maps
Prefix sum muon station ocurrence
Muon sort by station
Muon decoding

Figure 3.1: Decoding sequences detail.

sensor. Each chip is composed of an array of 256×256 pixels that can individually fire when a particle leaves a signal.

The total number of pixels in the Velo subdetector is therefore 52 (modules) $\times 4$ (sensors) $\times 3$ (chips) $\times 256 \times 256$ (pixels per chip) ≈ 40M. If each pixel were to be encoded individually as raw data, each event would require 5 MiB solely to accomodate Velo raw data. This would be inadmissible given the average LHCb event size of 150 kB.

Since the average occupation of the Velo subdetector is under 0.1%, the array of pixels in each chip is grouped in *superpixels* (SPs) of 8 pixels, and only active superpixels are encoded, alongside its location coordinates relative to the containing sensor. Figure 3.3 shows the layout of a SP. The order of the pixels is increasingly from the bottom in y, and increasingly from the left in x.

Velo raw data is encoded in 208 raw banks. Each raw bank corresponds to a distinct sensor. Each raw bank encodes an *SP header* followed by *SP words*. The number of SP words in the sensor is encoded in the 16 least significant bits (LSBs) of the SP header. Figure 3.4 shows a detail of the formats of the SP header and the SP words. There are up to 384 columns and 64 rows where the SP could be placed, and thus 9 and 6 bits

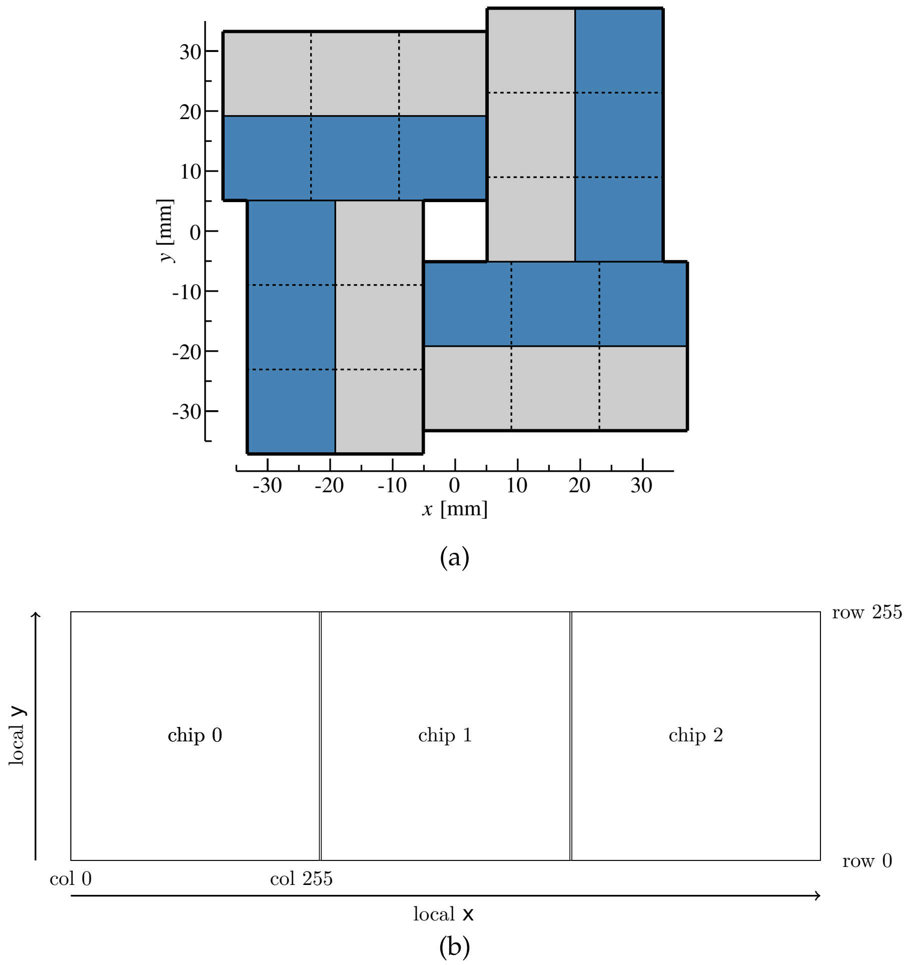
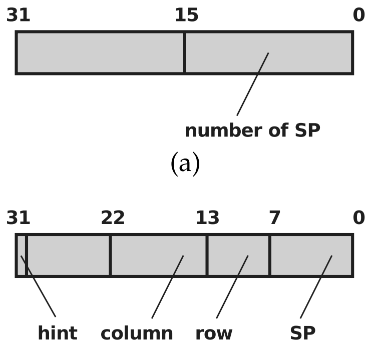


Figure 3.2: Top: Detail of pair of Velo modules. Bottom: Detail of Velo chips within a sensor.

3	7
2	6
1	5
0	4

Figure 3.3: Superpixel of Velo subdetector.

are sufficient to respectively encode the column and row in the SP word. The *hint* field refers to neighboring SPs. If its value is 1, then *it is guaranteed there are no neighboring active SPs to the current one*. If it is 0, then *it is not guaranteed there are neighboring active SPs to the current one*. This sort of indetermined state when the hint is 0 is due to the limitations of the parallel pipeline of the FPGAs [1] producing the format [46].



(a)

(b)

Figure 3.4: Top: SP header. Bottom: SP word.

3.1.1 *Velo clustering*

When particles produce signals in the Velo subdetector, they often fire several neighboring pixels. The reconstruction of the Velo takes this phenomenon into account. Pixels in neighboring cells are considered connected following the 8-connectivity definition, whereby a pixel is connected to either of the adjacent horizontal, vertical or diagonal pixels (in essence, a chess king's reach). The problem of Velo clustering is a variant of *Connected Component Analysis* (CCA): given a set of SPs, the average columns and rows of 8-connected pixel clusters are sought.

The Velo clustering produces as a result a set of pixel clusters or *hits*. For each hit, its spatial coordinates are stored, alongside a unique identifier known as the *LHCb ID*. Figure 3.5 shows the composition of the Velo LHCb ID format.

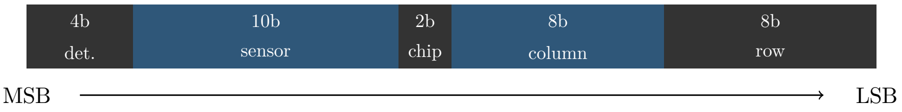


Figure 3.5: Unique hit identifier known as LHCb ID. Format of Velo detector.

A straightforward manner to implement the Velo clustering sequentially is to process sensors one by one, keeping a representation of the sensor 768×256 pixel space in memory. First,

[1] A *Field-Programmable Gate Array* (FPGA) is an integrated circuit that can be programmed, usually employing a hardware description language. FPGAs produce the raw data format that is then decoded in processors.

all pixels in the SPs of the sensor are loaded onto the sensor pixel space. A stack (LIFO) can then be used to traverse the pixels and find all connected pixels by inspecting sequentially each of the 8 neighboring locations, following a breadth-first search and flagging visited pixels to avoid revisiting pixels. The method is described in detail in [46].

```
for (auto sensor : sensors) {
  vector<int> pixel_indices;
  array<768 * 256, bool> sensor_pixels;

  // Initialize sensor_pixels to false.
  // Initialize pixel_indices with pixels from
  // SPs, and sensor_pixels of those indices to true.
  [...]

  vector<int> stack;
  for (auto pixel_id : pixel_indices) {
    if (sensor_pixels[pixel_id]) {
      sensor_pixels[pixel_id] = false;
      int x = 0, y = 0, n = 0;
      stack.push_back(pixel_id);
      while (!stack.empty()) {
        const auto index = stack.back();
        stack.pop_back();
        x += column(index);
        y += row(index);
        n++;
        for (auto neigh_pixel_id :
          neighboring_pixels(pixel_id)) {
          sensor_pixels[neigh_pixel_id] = false;
          stack.push_back(neigh_pixel_id);
        }
      }
      add_hit(sensor, x/n, y/n);
    }
  }
}
```

Listing 3.1: Sequential Velo clustering pseudo-code.

A pseudo-code of the sequential implementation is presented in Listing 3.1. For every sensor, `pixel_indices` are populated with the individual pixels from the superpixels. `sensor_pixels` keeps a view of the entire sensor. It is initialized to `true` for the individual pixels, and `false` otherwise. From line 11 onwards, pixel indices are consumed one by one. If the condition of line 12 is met, then said pixel is still not processed. The CCA instance consists in finding the average row and column in the cluster formed from the current pixel. To this end, variables `x`, `y` and `n` keep a sum of the column, row and number of pixels in the cluster, respectively.

The method employs a stack to navigate the neighboring pixels. Elements from the stack are traversed and popped in lines 17 and 18. `x`, `y` and `n` are updated from line 19 through 21. Neighbouring pixels are added to the stack in the for loop of line 22. Finally, once the stack has been consumed, the resulting hit is added to the set of reconstructed hits.

A visual walkthrough the algorithm is shown in Figure 3.6. In (a), the working green pixel is inspected. In (b), the pixel is marked as visited (black), neighboring active pixels are added to the stack (blue), and the next pixel in the stack becomes the working pixel (green). Pixels are subsequently added to the stack and processed one by one, as shown in (c) and (d). Once the cluster is processed and no active pixels remain in the stack, the cluster is created.

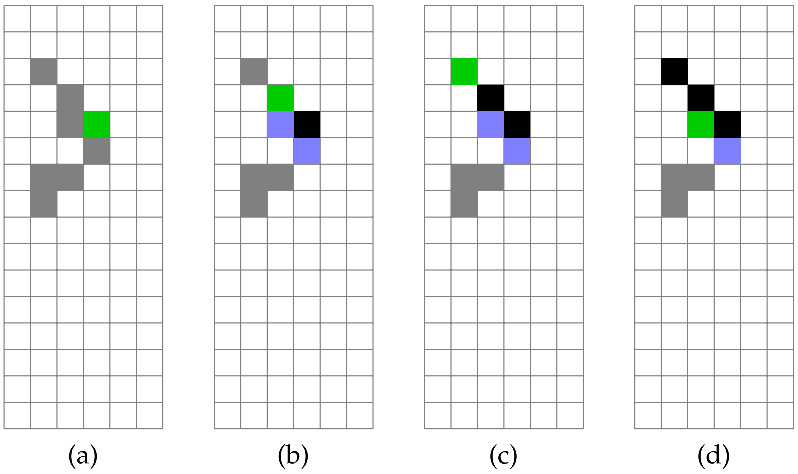


Figure 3.6: Four successive steps of the sequential clustering algorithm.

The presented implementation is simple and readable. A natural optimization can be achieved by processing SPs where the *hint* is 1 separately – information local to those SPs is enough to reconstruct the cluster of pixels. Given that an SP is composed of eight pixels, 256 possible variations of the contents of the SP exist. A lookup table suffices to decode single SPs in a constant time of `log(256)`.

However, limitations arise when this method is attempted in parallel. The `sensor_pixels` array would need to be extended to all the sensors, as it is required to process each sensor. A complete 5 MiB map of the pixels in the detector would be required per event, restricting the number of events processable in parallel. In addition, the algorithm contains numerous Read-After-Write (RAW) dependencies. `sensor_pixels` is updated with every traversed pixel, and the stack state is locally needed from a control flow standpoint. In order to overcome these shortfalls an algorithm has been designed from scratch.

3.1.2 *Velo estimate input size*

First, the number of hits in the Velo subdetector must be calculated in order to be able to allocate a buffer to hold hit data. Obtaining an estimate of the number of hits instead of the exact number of hits would also solve the allocation issue, as long as it is guaranteed the number of hits is overestimated and never underestimated.

Single SPs, that is, SPs where the hint is 1, can at most contain two clusters. As Figure 3.7 shows, two configurations can lead to two clusters in a single SP: (a) SP bits 1 and 5 inactive, at least one of either bit 0 or 4 active, and at least one of either bit 2, 3, 6 or 7 active – (b) SP bits 2 and 6 inactive, at least one of either bit 3 or 7 active, and at least one of 0, 1, 4 or 5 active. The number of clusters in a single SP can be determined with a constant-time lookup operation.

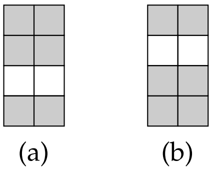


Figure 3.7: Superpixel configurations leading to two clusters.

Clusters that span several SPs pose a more complex problem. A cluster is usually composed of no more than 4 pixels, but there is no hard limit to the number of pixels in a cluster, other than the sensor extension. Figure 3.8 shows a distribution of the expected cluster sizes. The probability of a cluster consisting of many pixels is very low, but no assumption as to the number of pixels in a cluster can be established a priori.

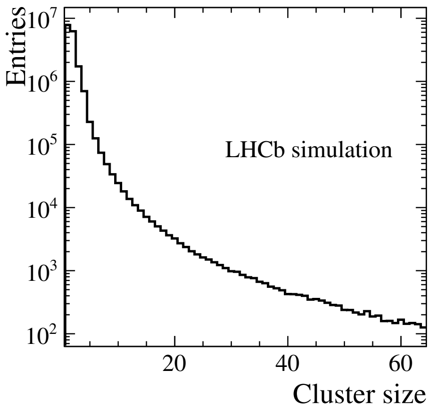


Figure 3.8: Distribution of cluster sizes for 10 000 simulated minbias, $\nu = 7.6$ events.

In order to estimate the number of clusters in multi-SP cases, the following observation is made:

Lemma 3.1.1. *Any cluster presents at least one pixel whose neighboring pixels north, north-east, east and south-east are either inactive or out of bounds.*

Proof. It can be demonstrated by *reductio ad absurdum*. Let us consider the set of east-most pixels in a cluster, and let us assume no pixels in that set fulfill the condition. In order for a pixel not to fulfill the condition, there must be pixels either to the north, north-east, east or south-east of it. Since the set of east-most pixels is being considered, it is only possible that a pixel be on its north cell. However, since the cluster is finite, there is a north-most pixel in the set, either with not active neighbors on its north cell or at the limits of the matrix. □

This condition is depicted in Figure 3.9a. Three clusters are shown in Figure 3.9b. Only one pixel fulfills the condition for the top left cluster and the middle cluster, whereas the cluster on the right has four pixels that fulfill the condition. The bottom right pixel fulfills the condition as some of the pixels are out of bounds.

An estimation of the number of clusters in the event can be made by searching for these patterns and assuming they refer to a single cluster. As has already been shown, there

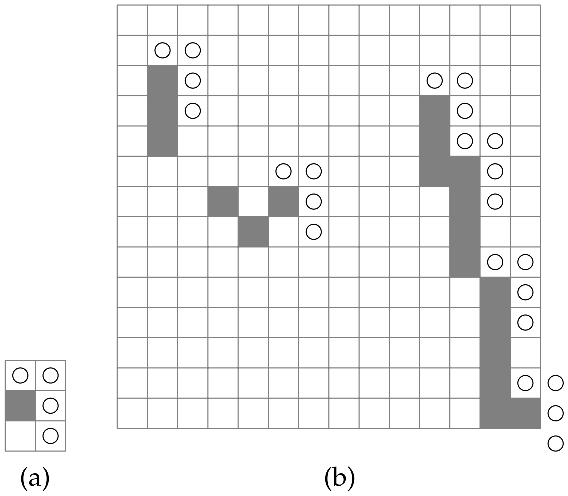


Figure 3.9: (a) Condition at least one pixel in a cluster must meet. (b) An example in an arbitrarily sized array of pixels containing three clusters.

are cases where the number of clusters will be overestimated. In practice, this overestimation was found to be in average 1.5%, not impacting severely the amount of memory required. Furthermore, this method is guaranteed to always yield either the exact number of clusters or overestimate it. The pixels found with this method are referred to as cluster *candidates*, and will be used in a posterior clustering stage (see 3.1.4).

This method can be parallelized by having each thread look for the cluster pattern on a different SP. Given an SP, neighboring SPs to the north, north-east, east, south-east and south must be loaded in order to test the aforementioned cluster condition. Figure 3.10 shows various patterns that can be encountered when exploring an SP for the cluster condition. (a) depicts the external pixels to the SP that are required to be able to test the condition. The need for five neighboring SPs is justified with examples (b) through (e). Is is possible to encounter several clusters that fulfill the condition, as (f) shows.

Testing the condition can be done in parallel within an event, for all sensors and all SPs. The amount of memory required to hold the SP and the required neighboring pixels is 17 bits, held in a 32-bit register. A known drawback to this technique is that each thread must iterate all SPs in the sensor, until the required pixels are populated. However, as threads in a block process the same sensor, there is a high chance the SPs are in L1 cache due to locality.

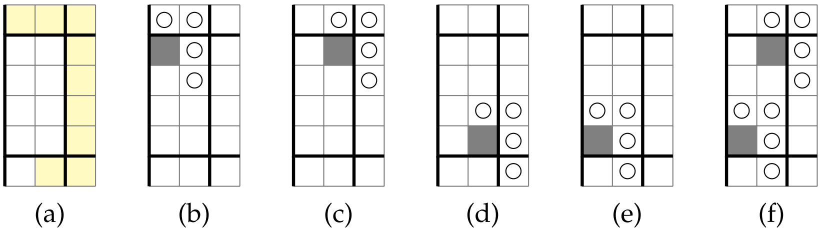


Figure 3.10: (a) Superpixel with neighboring pixels (yellow) required for testing the condition. (b) to (f): Pattern examples.

Each pixel fulfilling the condition is kept for posterior processing as a seed for a cluster. The algorithm *Velo estimate input size* performs this process and outputs a contiguous array of estimated sizes per module and a list of potential clusters per Velo module.

3.1.3 *Prefix sum Velo clusters*

The array of estimated sizes contains in contiguous locations the estimated sizes of each module in one event, followed by the sizes of each module of the next event. A prefix sum of this array permits to transform the input array, obtaining the total number of clusters in all events alongside an offset for every particular module in every event.

The prefix sum algorithm has been implemented following the Blelloch scan [47] algorithm, including the parallel optimizations in [48]. The scan has been implemented in three steps, and a visual aid is shown in Figure 3.11:

1. `Block scan` – The input array is subdivided in blocks of 2048 elements. A parallel exclusive scan is performed over each of the blocks, using the *reduce / downsweep* operations described in [48]. The sum of each block is stored in order in an auxiliary array. Each block is processed by a CUDA block with 1024 threads. The number of threads and number of elements in each block are chosen so that the workload is balanced across the executing threads.
2. `Auxiliary scan` – A parallel inclusive scan is performed over the auxiliary array with a single CUDA block.
3. `Add` – The auxiliary arrays are added to the blocks scanned in the `Block scan`. The last element of the auxiliary scan

is added to the input array, effectively increasing the size of the array by one element.

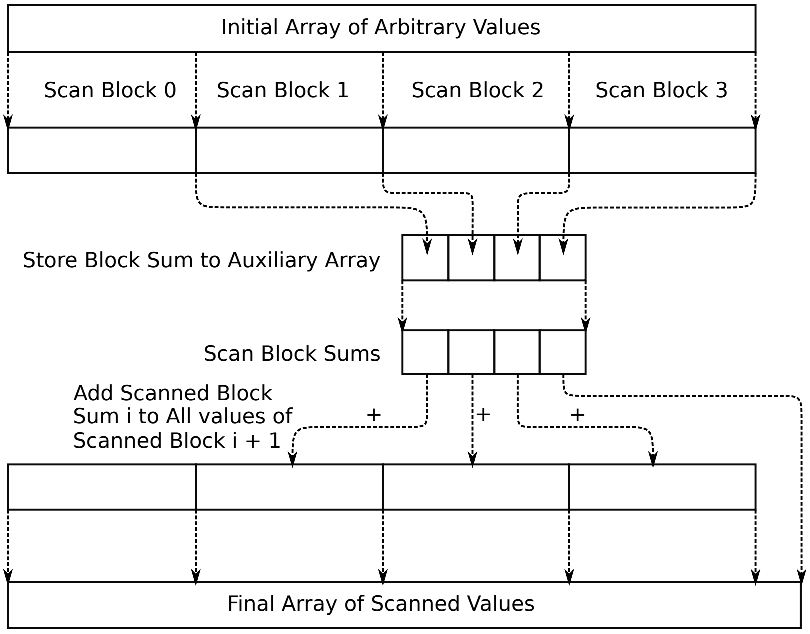


Figure 3.11: Parallel Blelloch scan.

The prefix sum can be further optimized by performing successive scans if the auxiliary array consists of many elements. Also, specialized libraries provide implementations of general purpose algorithms such as the prefix sum, for example [49]. These may be implemented in the future as improvements to the existing codebase.

A CPU implementation is also provided for all prefix sums. The CPU implementation fetches data from the GPU memory, performs the prefix sum, and ships the result back to the GPU. The CPU implementation is sequential, and it can be enabled at runtime with the toggle `-cpu-offload`.

Once the prefix sum over the module sizes is done, a buffer with as much memory as needed by the datatypes produced by the subsequent algorithms is allocated with the memory manager.

3.1.4 *Mask clustering*

Finally, cluster candidates are visited in parallel and expanded into clusters. Since memory requirements must be kept low, only the vicinity of the SP containing the cluster candidate is loaded, consisting in a map of three columns of four SPs each: one column to the west, the column including the cluster candidate, and one to the east. The SP containing the cluster candidate is placed on the second row starting from the north. Figure 3.12a depicts the SP containing the candidate and the neighbors loaded. The entire map is composed of 96 pixels. For the sake of explanation, let us assume a 96-bit datatype that can hold the entire map in a single object.

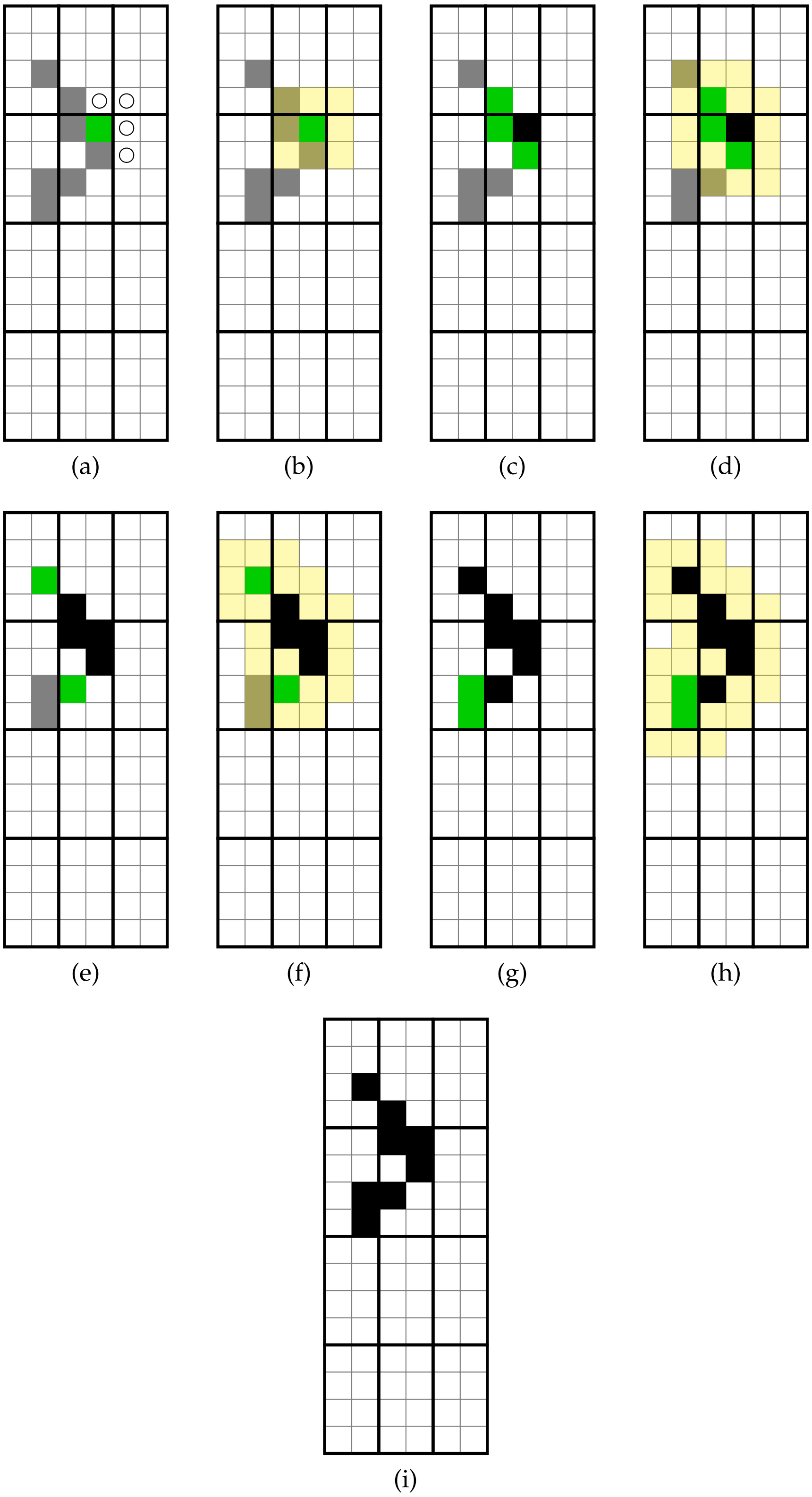
In order to reconstruct the cluster, a novel masking method is described. The method processes a cluster candidate, marked in green in Figure 3.12a. A bit-mask is created around the cluster candidate, including the cluster candidate pixel and the 8-connected neighboring pixels, as shown in yellow in (b). Then, the primitive *logical and* operation is applied between the mask and the SP map. The green pixels in (c) constitute the new pixels in the forming cluster. A mask is created in (d) around the forming cluster, containing the pixels in the cluster. The resulting cluster is taken for the next iteration. The whole process finishes when there are no new pixels in the cluster after applying the mask, as shown in (i). At this point, the entire cluster has been constructed.

A pseudo-code of the *Mask clustering* algorithm is shown in Listing 3.2. After initialization of the datatypes, the main loop of the method is in lines 15 to 18. The `cluster` is updated with the contents of `next_cluster`. The next cluster is then calculated with the logical and between the pixel map and a 8-connectivity mask of the cluster. The creation of this mask is done in constant time – regardless of the contents of the cluster, a mask can be created with 8 shift operations applying the *logical or* operation.

It is possible that the cluster contains pixels with *precedence*. In order to avoid cluster *clones*, the following precedence rules are followed: (a) if the cluster consists of pixels north in the same column of the candidate, or north-east, east or south-east of the candidate, the cluster is discarded; (b) else, it is kept. The reasoning for (a) is presented in the following lemma:

Lemma 3.1.2. *If a cluster is composed of a candidate and pixels located strictly to its north, north-east, east or south-east, then it contains another candidate located strictly to its north, north-east, east or south-east.*

Proof. Let us consider a candidate c in a cluster. Let S be the set of pixels on the same cluster located strictly to the north,



(a) (b) (c) (d)

(e) (f) (g) (h)

(i)

Figure 3.12: Mask clustering iterations.

```
int96_t pixel_map;
int96_t cluster;
int96_t next_cluster;
int8_t candidate_pixel_in_sp;
int column, row;

// Initialize:
// pixel_map with SPs.
// cluster to zero.
// next_cluster just with candidate to one.
// candidate_pixel_in_sp with candidate number in SP.
// column and row to starting column and row of map.
[...]

while (next_cluster != cluster) {
  cluster = next_cluster;
  next_cluster = pixel_map and 8con_mask(cluster);
}

const int96_t pixels_with_precedence = pixel_map and
    precedence_mask(cluster, candidate_pixel_in_sp);
if (pixels_with_precedence == 0) {
  const auto n = popcount(cluster);
  const auto x = column * n + x_sum(cluster);
  const auto y = row * n + y_sum(cluster);
  add_hit(sensor, x/n, y/n);
}
```

Listing 3.2: Mask Velo clustering pseudo-code.

north-east, east or south-east of c. If S is not empty, then it must contain at least one 8-connected subset of pixels. Let T be one of such 8-connected components. Then, T must contain at least one candidate, as per Lemma 3.1.1. □

The pseudo-code reflects this with the condition `pixels_with_precedence` be zero. Finally, the number of pixels in the cluster must be calculated. The `popcount` instruction was introduced in the *SSE popcount* instruction extension set and is commonly available in up-to-date hardware across architectures. The instruction counts the number of active bits (1s) in a number. The number of pixels composing the cluster is calculated applying the function to the entire cluster. In addition, the weight of each row and column can be calculated by applying popcount to specific parts of the cluster, achievable through masking, or by iterating active pixels in the cluster with the `count leading zeros` operation. The condition finishes with the addition of the hit.

Mask clustering can be parallelized across sensors and SPs. The implementation requires just a subset of the SP data that can be locally stored in thread registers.

3.1.5 *Physics efficiency*

In spite of the precedence rules set for the Mask clustering algorithm, it is possible that a single cluster composed of many pixels be interpreted as several clusters, which can lead to inefficiencies in the Velo track reconstruction. There was no working definition of *clustering efficiency* in the LHCb community, since clustering algorithms have typically performed *perfect clustering*, that is, found all correct clusters. In order to evaluate the algorithm, the following definitions are proposed:

- **Reconstruction efficiency**:
$$\frac{\text{Number of correctly reconstructed clusters}}{\text{Total number of reconstructible clusters}}$$
- **Clone fraction**:
$$\frac{\text{Number of clone clusters}}{\text{Number of reconstructed clusters}}$$
- **Fake fraction**:
$$\frac{\text{Number of fake clusters}}{\text{Number of reconstructed clusters}}$$

What constitutes a *correctly reconstructed cluster* is typically defined as finding the exact row and column of the cluster. It could be argued that this definition is strict: a cluster placed in the close vicinity is likely to be assigned to the same track and produce similar track parameters in the Velo tracking algorithm. A cluster is considered to be correctly reconstructed if the row and column are at maximum at a distance of two pixel units, as shown in Figure 3.13.

Given these definitions, the algorithm shows a reconstruction efficiency of 99.27%, a clone fraction of 0.004% and a fake fraction of 0.734%. In spite of the Velo tracking validator requiring

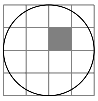


Figure 3.13: Error tolerance on average center of Velo clusters.

a strict LHCb ID equivalence, in other words, requiring rows and columns to be exactly correct, the impact of the clustering inefficiencies is negligible. The Velo reconstruction algorithm developed as part of this thesis presents a high reconstruction efficiency (see chapters 5 and 9).

3.2 UT DECODING

The UT detector consists in four planes with staves of sensors. Figure 3.14 shows a detail of the UTaX plane. The UTaX plane is composed of 16 vertical staves. Each box represents a sensor, and the density of strips composing each sensor accomodates the expected higher hit count in the region immediately closer to the beam line hole, depicted in the center. Green boxes represent sensors consisting of 512 strips, whereas yellow and orange boxes represent sensors consisting of 1024 strips. The orange sensors in the middle region span half the height of the other sensors, further increasing its precision.

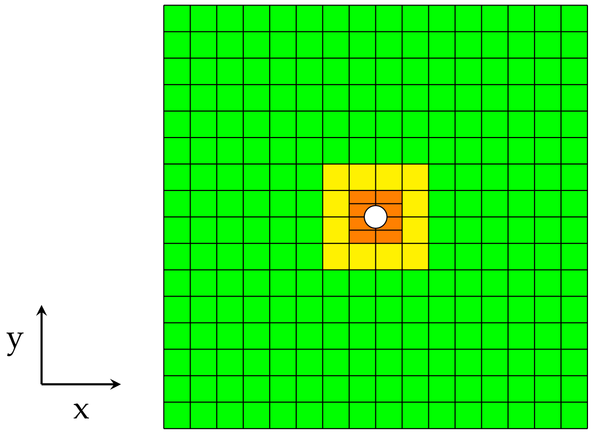
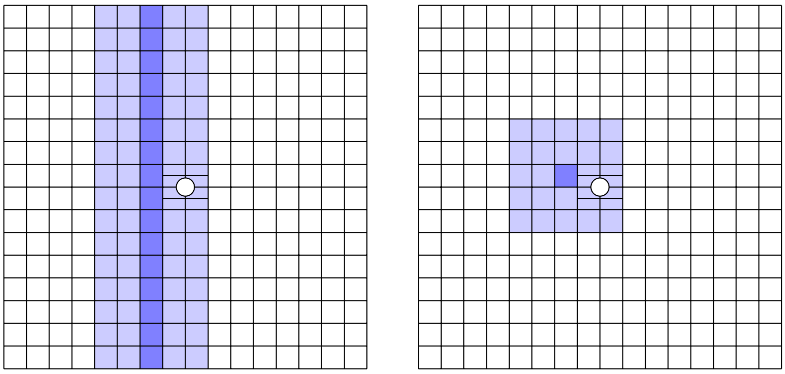


Figure 3.14: UTaX plane. Each box represents a silicon sensor.

Data generated with Monte Carlo samples for the upgrade LHCb detector encode UT hits with a pre upgrade data format, as per the elaboration of this thesis. At present, it is also unknown whether it will be necessary to perform a clustering algorithm to UT data. Therefore, the current baseline architecture decoding has been parallelized.

Irrespective of the input data format, the decoded hits are sorted in two dimensions, following a KD-tree-like structure [50]. Hits are grouped in sensor groups by their x, and within a sensor group, hits are sorted by y. This method allows efficient searches in the posterior UT tracking algorithm. Figure 3.15 depicts a search in the target data format in two stages. For a given Velo track extrapolated to a UT layer location $\{x, y\}$, the closest hits will be searched. The sensor group is identified by locating the x coordinate in the data structure, in darker blue in (a). Next, neighboring sensors are located by performing an additional search in y (b).



(a) (b)

Figure 3.15: Binary search steps for a given location onto a UT layer. (a) Search of sensor group. (b) Search of first and last hit within each sensor group.

The UT decoding algorithms achieve this data format in five stages, which are succinctly described in the following.

3.2.1 *Overview of UT decoding*

First, the number of hits in the subdetector is calculated by traversing the UT headers in algorithm `UT calculate number of hits`. The correspondance between sensors and sensor groups is statically defined, and thus each sensor group hit counter is incremented accordingly upon parsing the raw data structures. The number of hits of each sensor group and every event are then prefix summed, yielding a contiguous array of offsets (`Prefix sum UT hits`). A data buffer that can accomodate all UT hits in the events is allocated.

UT hits can be decoded at this stage. However, the order in which they should be inserted is still not known. In particular, all hits that are fired in the same sensor yield the same y information. Figure 3.16a shows six fired hits in two consecutive sensors in a sensor group. The y of each hit spans the entirety of each sensor, and thus the decoded y (`y begin` and `y end`) cannot be used to obtain a deterministic order within a sensor. In order to overcome this limitation, hits are sorted in a sensor group according to the hit y ($\frac{\texttt{y begin} + \texttt{y end}}{2}$), and if it is the same, to the hit x, as shown in Figure 3.16b.

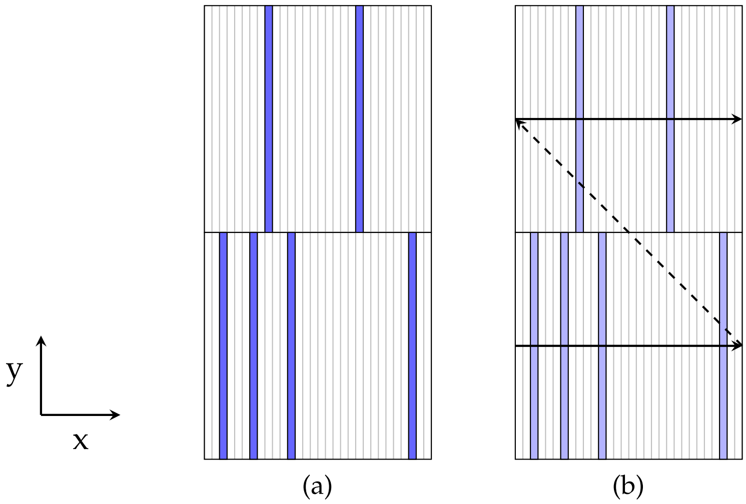


Figure 3.16: (a) Fired hits in two sensors in the same sensor group. (b) Sort order within sensor group. Only 30 strips are depicted for each sensor to aid with visualization, instead of the 512 or 1024 strips that would compose a sensor.

This logic is efficiently implemented in three steps. `UT pre decode` decodes the x and y of every hit, alongside the raw bank index of the hit. `UT find permutation` sorts sensor groups by the insertion sort method, producing a permutation. Finally, `UT decode raw banks in order` utilizes the permutation to access the relevant hit, and decodes UT hits in place. A coalesced data access pattern is achieved by storing contiguous decoded hits in neighboring CUDA threads. Decoded UT hits consist of six 32-bit quantities, each of them decoded into their own SOA.

3.3 SCIFI DECODING

SciFi raw data is already clustered and sorted by x prior to being processed by the HLT1 sequence. The headers and banks composing SciFi raw data are depicted in Figure 3.17. Raw banks are encoded in 16-bit words. They start with a header and are followed by clusters that can either refer to:

- Single clusters – Encoded by a single word.
- Fragmented clusters – Encodes consecutive fired clusters with two consecutive words.

A bit in the word identifies single clusters from fragmented clusters. SciFi raw data can be decoded in a precise or in a *lossy* manner. The lossy decoding interprets all clusters as single clusters. Various revisions of the SciFi format have appeared in the last years. The lossy decoding *v4* has been implemented. Two more recent versions of the decoding format have been produced since, and efforts in supporting them are currently undergoing. The impact in efficiency of using a lossy decoding as opposed to a precise one is expected to be around 1%.

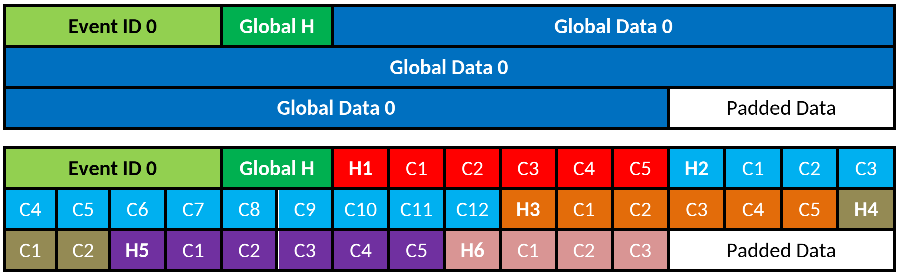


Figure 3.17: Format of SciFi raw data headers and banks.

The SciFi decoding first calculates the cluster count in the algorithm `SciFi calculate cluster count v4`. As a side effect of using lossy decoding, the number of words in every raw bank gives an exact count of the number of clusters that will be posteriorly decoded. The number of clusters is prefix summed in `Prefix sum SciFi hits`, yielding offsets and sizes for every SciFi *mat* [51]. A buffer for the clusters is prepared in GPU memory.

Decoding SciFi hits can then be performed. However, if decoding of raw banks is performed in parallel with an increasing atomic counter, the order of clusters would not be preserved, resulting in an inefficient decoder.

The first two SciFi stations encode four mats on every raw bank whereas the last SciFi station encodes five mats on every raw bank. Data is sorted either monotonically increasingly or monotonically decreasingly every four mats. This implies that data is sorted between raw banks only in the first two stations, and requires a more fine grain processing for the third station, where the *four mat in one raw bank* equivalence does not hold. The first two station hits are stored either monotonically increasing or monotonically decreasing in x. The top 8 zones, corresponding to the last station, do not follow this pattern.

The pattern of the first two stations is exploited in the parallel implementation of the SciFi decoding. `SciFi direct decoder v4` decodes clusters in the first two stations. It determines the location of each cluster by its current raw bank offset, obtained from the prefix summed datatype produced in the previous algorithm and the local index of the cluster word with respect to the beginning of the raw bank.

The last station is decoded in two steps. `SciFi pre decode v4` stores 32-bit *pointers* to the location of the raw data and the words for every cluster in order. `SciFi raw bank decoder v4` iterates the pointers, gathers the data and stores it in a coalesced manner, maximizing the efficiency of store operations.

The SciFi decoding produces SOAs for each datatype required by subsequent algorithms. The `x0`, `z0`, `channel` and `endPointY` of each cluster are decoded. These variables describe the position and properties of each cluster. In order to reduce memory bandwidth of these algorithms, the *fraction*, *plane code*, *pseudo size* and *mat* of each cluster have been encapsulated into one single 32-bit datatype named `assembled datatype`.

3.4 MUON DECODING

Similarly to other detectors, muon raw data is organized in banks. Each bank contains four batches of data of a variable size, encoding tiles in 16-bit words. Each tile's logical position, consisting of its station, region and quarter, is decoded. Tiles in the same station, region and quarter are partitioned into two sets depending on their layout, and the following logic is applied:

- If a tile represents a pad, a hit is added.
- For every two tiles representing strips that cross, a hit is added. These tiles are flagged as used.
- Every tile representing a strip that was not flagged is considered an uncrossed tile and encodes exactly one hit.

Therefore, the final number of hits cannot be calculated until all crossings of all tile partitions have been processed. The muon decoding is thus implemented as follows. The number of tiles is decoded in the `Muon pre decoding` algorithm, together with station, region and quarter placement, encoded as a single 32-bit integer. The number of tiles is prefix summed in `Prefix sum muon pre decoding`.

Then, all tiles are sorted in the `Muon sort station region quarter`, employing a parallel insertion sort and utilizing shared memory as temporary storage. Tiles are decoded following the above logic in the algorithm `Muon add coords crossing maps`. In order to avoid the high memory footprint of decoding tiles only to reshuffle them later, a 64-bit long integer encoding hit-identifying information is stored as `muon compact ids`.

The final hit count is calculated in `Prefix sum muon station ocurrence`, and hits are sorted and compact ids are decoded in place in `Muon sort by station`. Data is stored in the appropriate SOAs in this last step.

4

TRACK RECONSTRUCTION

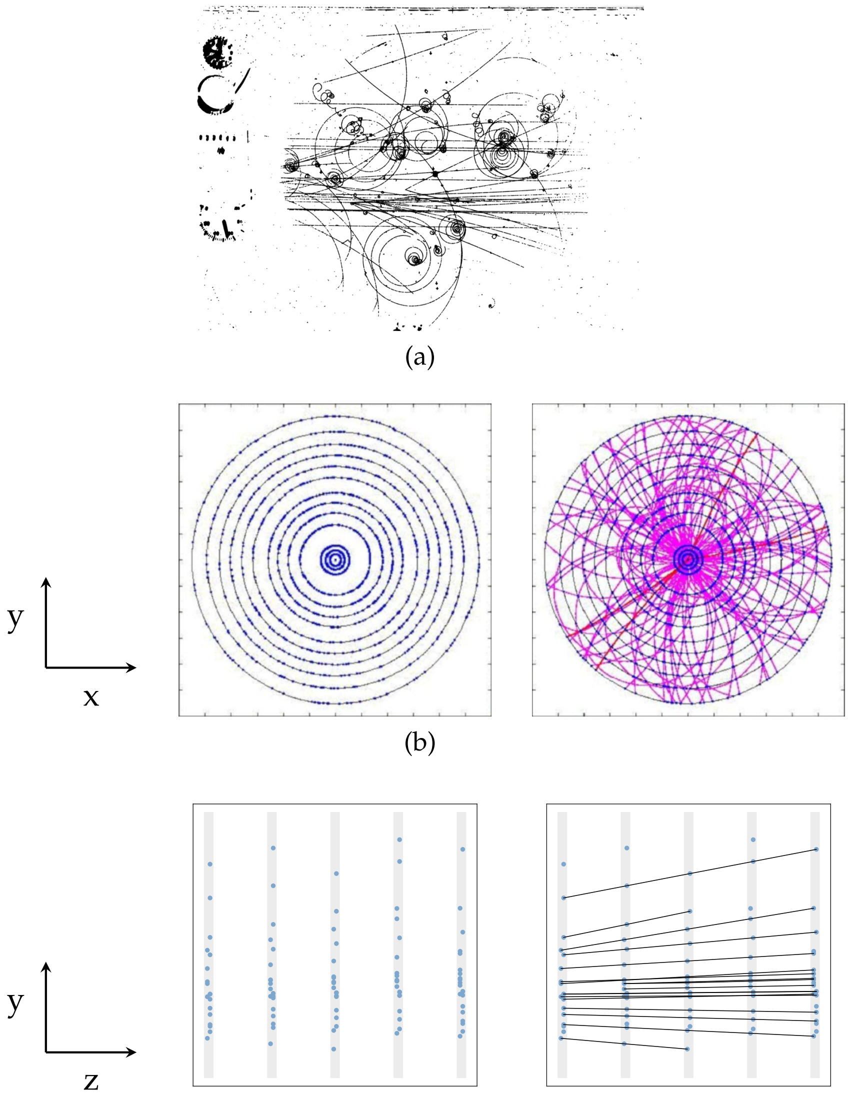
Track reconstruction, or *tracking*, is a pattern recognition problem consisting in finding particle trajectories from measurements in detectors alongside their path. The problem is equivalent to finding a partition of disjoint sets of measurements, accounting for the fact that some measurements may be noise, and some particle trajectories may not be of interest to the researcher.

The trajectory of particles may be deviated by a magnetic field, as a function of the electric charge and the momentum of each particle. The detecting material may also interact with the particles in a process known as *multiple scattering*, affecting the trajectory of the particles. Multiple scattering is usually regarded ultimately as a physical limitation of the detecting apparatus and is not accounted for, other than the expected statistical inefficiency caused by it.

Various track reconstruction problems are shown in Figure 4.1. Early examples of track reconstruction would be done in *bubble chambers* [52], whereby a particle would create ionized tracks in its pass through a *superheated liquid*, causing the liquid to vaporize and generate bubbles. Bubbles would then be photographed and rudimentary digitized, yielding images such as (a). Tracks are visible to the naked eye with this early technology.

Two more recent examples are shown in (b) and (c). A cylindrical detector with measurements and tracks is shown, where the presence of a magnetic field deviates particle trajectories into helical trajectories. A YZ projection of a section of the Velo subdetector is shown in the bottom, where trajectories are straight lines. In both of these examples, measurements are shown on the left, whereas trajectories are shown on the right.

While the mathematical models describing particle trajectories have not changed, track reconstruction has become increasingly complex due to the increase in collision rates and the increase in event hit multiplicities. In some cases like the ALICE *time projection chamber* (TPC) or the silicon tracking system of CBM,



(a)

(b)

(c)

Figure 4.1: (a) Bubble chamber event. (b) Tracking detector with cylindrical layers. (c) YZ projection of section of Velo.

the high rates produce overlapping events, adding a time dimension to the problem [53]. It is therefore crucial to explore the different techniques and adapt them to parallel processors, in order to keep up with the demanding requirements of modern detectors.

4.1 EFFICIENCY INDICATORS

Particles that pass through a detector are considered *reconstructible* when they meet a criterium specific to the detector.

The criterium depends on factors particular to the technology of the detector, such as its topology, the detecting medium, and the kind of particles expected to be reconstructed. Within the LHCb tracking system, a particle is reconstructible in a subdetector if:

- Velo – The particle generates at least three different hits (clusters of pixels).
- UT – The particle produces at least one hit in an x layer, and one hit in a u/v layer.
- SciFi – The particle generates at least six hits, with at least one hit on each station x layers, and at least one hit on each station u/v layers.

The validation of a track reconstruction algorithm is performed against Monte Carlo simulated samples, where reconstructed tracks should match Monte Carlo particles, which establish the ground *truth*. The matching of tracks with particles is done on a hit by hit basis. In LHCb, this is accomplished by comparing their LHCb IDs.

The physics quality of found tracks can be evaluated according to five indicators [12]:

- The track *reconstruction efficiency* can be determined by the ratio between the reconstructed tracks of reconstructible particles, over all the reconstructible particles:

$$\frac{N_{\text{reconstructed and reconstructible}}}{N_{\text{reconstructible}}} \tag{4.1}$$

- A *fake track* (*ghost track*) is created when a percentage of hits in a track are not from a real track. In LHCb, at least 70% of the hits in a track must be in a Monte Carlo particle to be associated in the validation process. The *fake track fraction* is the ratio between the fake tracks and all the reconstructed tracks:

$$\frac{N_{\text{fake tracks}}}{N_{\text{reconstructed tracks}}} \tag{4.2}$$

- The *clone track fraction* refers to the fraction of tracks associated to the same Monte Carlo particle as another reconstructed track:

$$\frac{N_{\text{clone tracks}}}{N_{\text{reconstructed tracks}}} \tag{4.3}$$

- The *purity* in a track refers to the fraction of track hits that belong to the true particle:

$$\frac{N_{\text{track hits in true particle}}}{N_{\text{track hits}}} \tag{4.4}$$

- Finally, the *hit efficiency* yields the number of hits correctly found out of the true particle hits in a track:

$$\frac{N_{\text{track hits in true particle}}}{N_{\text{true particle hits}}} \tag{4.5}$$

Validation numbers are typically listed in separate categories. Table 4.1 shows a sample validation output from a run of the GPU sequence over Monte Carlo data. Global fake particle rates are presented at the top. For each category, the reconstruction efficiency is shown, followed by the clone fraction, the purity and the hit efficiency.

	Fake fraction
TrackChecker output	7.61%
for $p > 3$ GeV, $p_T > 0.5$ GeV	6.62%

Track type	Reconstruction efficiency	Clone fraction	Purity	Hit efficiency
Long	45.49% (46.41%)	0.66%	96.81%	92.30%
Long, $p > 5$ GeV	62.74% (63.85%)	0.64%	96.98%	93.39%
Long strange	29.86% (30.19%)	0.80%	96.15%	91.21%
Long strange, $p > 5$ GeV	51.88% (52.77%)	0.70%	96.47%	92.91%
Long from B	67.66% (68.34%)	0.66%	97.32%	94.06%
Long from B, $p > 5$ GeV	78.29% (78.52%)	0.66%	97.39%	94.48%
Long electrons	12.59% (13.02%)	2.56%	95.52%	91.30%
Long electrons from B	28.37% (30.41%)	1.83%	95.86%	91.97%
Long electrons from B, $p > 5$ GeV	41.91% (44.59%)	1.65%	95.96%	92.66%
Long from B, $p > 3$ GeV, $p_T > 0.5$ GeV	80.06% (79.05%)	0.66%	97.38%	94.32%

Table 4.1: Physics validation output.

The quantities shown strictly follow the definitions above. It would also be possible instead to calculate a weight per event and average it over the number of events processed. In particular, the weighted reconstruction efficiency is:

$$\frac{1}{n}\sum_{i=0}^{n}\frac{N_{\text{fake tracks}_i}}{N_{\text{reconstructed tracks}_i}} \tag{4.6}$$

Densely populated events can outweigh low populated events simply because the number of tracks in those is higher. Weighting by event allows to mitigate this issue. Table 4.1 shows both reconstruction efficiency and weighted reconstruction efficiency (between parantheses).

4.2 OVERVIEW OF TRACK RECONSTRUCTION METHODS

Due to the interest in tracking by many physics experiments, various track reconstruction techniques have been documented in literature [54]. *Local tracking methods* find tracks iteratively, whereas global methods adapt an equivalent formulation of the problem, typically including all measurements, where solutions map to tracks.

4.2.1 *Local methods*

The most common local tracking method consists in finding a *track seed* and extending it to other detector planes in a process known as *track forwarding* or *track following*. The track seed is usually formed by a segment of two or three hits, and the search starts in a region where the hit multiplicity is lower and thus signal is cleaner, which usually corresponds with the furthest region to the expected interaction point. Track seeds are extrapolated (*forwarded*) to detector regions closer to the interaction point by applying an extrapolation accounting for the presence of a field if necessary. A model of the track can be formed from the track hits, and this model can be employed to select among a list of candidate hits the best fitting one. This extrapolation process may account for missing hits in detector parts, according to the hit inefficiencies of the physical detector and *dead* regions without sensitive detectors. Once a track is fully built, its constituent hits can be flagged so they are not revisited in further seed or forwarding steps.

The LHCb baseline Velo reconstruction algorithm is based on a track forwarding technique. The algorithm *Search by pair* [8] constructs seeds of pairs of hits initially in the furthest modules from the interaction region. Track seeds are forwarded to neighboring modules, allowing for one consecutive missing module on any one side of the Velo subdetector. Tracks of four or more hits flag all of their hits, reducing the search time of further seeding and forwarding steps, and avoiding the creation of clone tracks. Tracks consisting of at least three hits are stored, in accordance to the reconstructibility condition of the Velo subdetector.

Track forwarding has commonly been used in conjunction with the Kalman filter estimator [55, 56]. The Kalman filter provides an iterative method to update the prediction of a track model (see chapter 7). The χ^2-statistic determines the best measurement with regards to the prediction of the model.

Other local tracking methods are *track roads* and *track elements*. The track roads method consists in forming candidates with two hits situated in the extremes of the detector, and creating a path or *road* between both hits by interpolation. In case the model of the track be curved, a third hit should be added. The width of the road determines the accepted error in the model, and it depends on the characteristics of the detector [57]. The track elements method has two phases: (1) seeds are made up from neighboring points, straight lines or parabolic lines. Each seed is converted into a *master point* (a weighted average of the points) and a direction. (2) The seeds, instead of the original hits, are used to perform tracking. This method reduces the number of hits to consider in the tracking phase, at the cost of a loss in precision [58].

Hit multiplicity is often a concern in real-time reconstruction environments, where track reconstruction must be performed at a high throughput in order to keep up with the collision rate. Spatial reductions can be employed to reduce the search time of hits under consideration for local tracking methods. This envolves a data preparation step prior to the application of the tracking method.

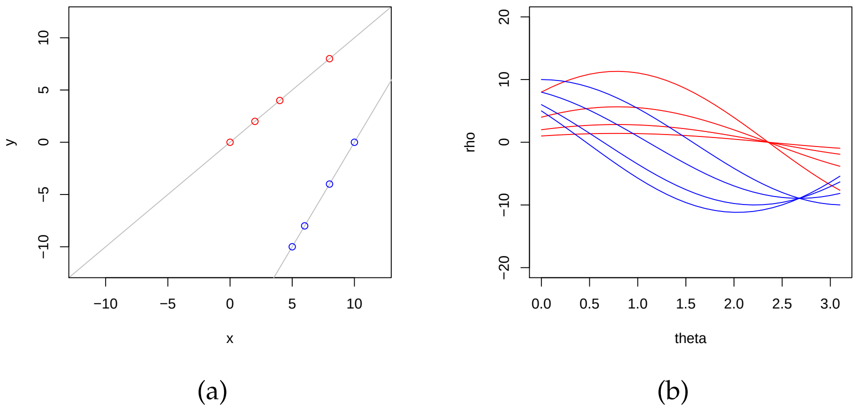
One simple spatial reduction is achieved by sorting hit arrays by a particular coordinate. Hit arrays are sorted on a per module basis prior to performing the Search by pair algorithm. This ensures every iterative search of compatible hits can be performed with a binary search, reducing the overall number of memory accesses of the algorithm. The dimensionality of hits under consideration can be reduced by employing R-tree structures [59] or KD-tree structures [50]. The specifics of the geometry of the detector yield in some cases a natural subdivision of the problem, as in the ALICE TPC sectors [60].

4.2.2 *Global methods*

Track reconstruction can be adapted to global formulations of the problem. The *histogramming method* plots all possible lines that pass by each point. A line that passes through all points will be visible in the histogram produced. In a simplified formulation, all possible lines containing a point of coordinates x, y can be expressed with the polar coordinate representation of a line:

$$\rho = x \cos(\theta) + y \sin(\theta) \tag{4.7}$$

The histogramming method applied to 2D tracks and straight lines is exemplified in Figure 4.2. Points on the left image are mapped to lines in the ρ and θ space. A 2D histogram of the right image would reveal where the lines cross, and what are the parameters of the compatible tracks.



(a)

(b)

Figure 4.2: (a) Points in space across two lines. (b) Representation in ρ and θ space of all possible lines traversing the points.

The histogramming method is a special case of the more generic Hough transform method [61]. Despite of the elegance of the Hough transform underlying principle, its application is more nuanced. Difficulties arise when dealing with high hit multiplicities, where binning and threshold of the histogram play an important role in avoiding excess of clones or fake tracks. Circular trajectories must be converted to lines prior to applying the Hough transform, which can be achieved with a *conformal mapping* transformation.

The *clustering method* consists in extrapolating hits onto a parameter space, according to the expected trajectory from a collision vertex. Once in the parameter space, hits pertaining to the same track appear close to each other. The tracking problem consists then in a clustering problem, that can be solved with any clustering method [62]. The collision vertex must be chosen with sufficient precision, and the resulting cluster must be verified to match reasonable track parameters [63].

Even in cases where the origin vertex is only estimated, the problem transformation yields tracks that share patterns in the parameter space. Figure 4.3 shows the extrapolation of all hits from the origin in the Velo, with clusters found by (a) *k-means* and (b) HDBSCAN [64]. Given that Velo tracks are straight lines, the expected pattern out of the parameter space extrapolation is either a cluster or a line of close points.

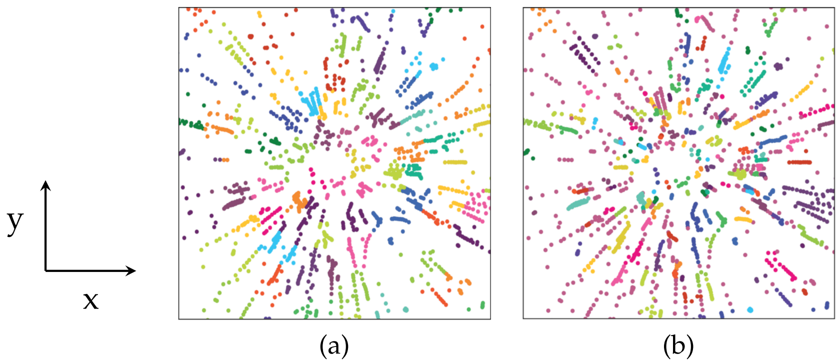


Figure 4.3: Extrapolated points from Velo onto parameter space with two solutions by clustering algorithms, obtaining the color-coded tracks. (a) Solution provided by k-means. (b) Solution attained by HDBSCAN algorithm.

The *automata method* [41] is a graph traversal method of a weighted directed graph representing the measurements[1]. Each measurement is a vertex, and directed edges between measurements in neighboring detector elements are created according to the expected track parameters in the region of acceptance of the detector, in the direction from outer to inner-most layer. The graph is then traversed following a Depth-First Search, assigning weights to visited edges according to the current depth level. If vertices are visited multiple times, the highest weight prevails.

[1] An equivalent formulation with automata is described in the paper [41].

In spite of its locality upon considering connected layers, the automata method has been successfully implemented in various detectors [43, 65]. In part, this is due to the separability of the solution into sub-algorithms, and its amenability to being efficiently implemented in parallel processors. Other graph traversal techniques such as a variation of a *Minimum Spanning Tree* have been applied successfully to track reconstruction problems [66].

Both the algorithm and the underlying hardware architecture must be considered when devising an efficient track reconstruction solution. It has been discussed that global methods are amenable by design for parallel architectures [54, 63]. In this thesis however, the author has investigated a novel local tracking method and presents various algorithms in the following chapters that demonstrate local methods may also be efficiently implemented in parallel architectures.

5

VELO TRACKING

VELO subdetector track reconstruction is one of the critical problems in the LHCb reconstruction in spite of the simplicity of particle trajectories therein. The high collision rates of the LHC combined with the increase in luminosity expected in the LHCb upgrade translates into an expected rate of 10^9 particle trajectories per second. In addition, the physics performance of the algorithm directly impacts the performance of all subsequent stages in the track reconstruction sequence.

The author has previously worked in the development of a parallel local method for the Velo reconstruction that showed promise [44]. Building on that foundation, the author has developed a fast local algorithm for track reconstruction on parallel architectures. The algorithm has been designed for the Single Instruction Multiple Thread (SIMT) programming model of CUDA. In addition, the algorithm has been translated to the SPMD programming model of ISPC [67], making it compatible with CPUs.

The algorithm, named *Search by triplet*, has been presented in the IPDPS conference, at the PDSEC workshop [1]. The publication is included in appendix A. The current chapter discusses the fundamental concepts, and extends on the results presented in the publication.

5.1 DISCUSSION

The sequential Velo tracking algorithm, implemented in the previous *Run* 2 of the experiment, has been briefly introduced in section 4.2.1. A local tracking method is employed, whereby hit pair seeds are created and forwarded to neighboring detector modules. Upon completion of a track, if the track consists of at least four hits, all hits composing the track are flagged and not considered in posterior stages. This method avoids traversing hits once they have been assigned to tracks, which is advantegous since (1) clone and fake rates are lowered, and (2) it reduces the multiplicity of hits to consider to fully reconstruct the detector.

The sequential algorithm has been validated, and it delivers the required physics performance of the LHCb physics program. However, from a parallel design standpoint the algorithm presents several shortcomings. Hits of Velo modules are sorted prior to the execution of the algorithm. Then, Velo modules are explored in order, finding compatible pairs and extending them, checking compatible hits in the sorted order, within some boundaries established by a tolerance condition. This process is deterministic, however, it depends on the order in which hits are considered. If a hit with a better fit χ^2 than the found one exists, it is not visited. Therefore, the processing order determines the solution. In addition, hits in tracks are flagged, imposing a visiting order to maintain determinism. These two conditions are implicit read-after-write dependencies, and prevent parallelization without blocking conditions.

Some parallel methods circumvent the stated dependencies. The seeding phase of the CMS cellular automaton algorithm [65] calculates all triplets in parallel, avoiding any flagging dependency. However, this comes at the cost of processing all possible hit triplets, making the method inefficient in densely populated detectors.

Global tracking methods on the other hand are parallelizable to some degree. Every bin can be populated in parallel in the histogramming method, and several independent histograms can be populated in parallel and later combined. The automata method has been successfully implemented in several SIMD parallel architectures [43, 68]. A variety of parallel clustering methods exist in literature [69].

Search by triplet is a realization of a local tracking method. In spite of the inherent data dependencies of local methods, the author has found sufficient parallel workload by processing triplets of modules at each step on both sides of the detector simultaneously. The algorithm is optimized for SIMD architectures, and runs efficiently both on multi and many-core architectures.

The physics quality of the algorithm is discussed in section 9.1. The performance of the algorithm has been further optimized, and additional architectures have been analyzed in section 10.2.2.

The *GPU sequence framework* presented in the publication [1] is discussed in greater detail in part iii.

The Search by triplet algorithm is a key component of the tracker system presented in this thesis. It historically stands out as the first algorithm to demonstrate the feasibility of any LHCb reconstruction algorithm on GPUs, leading to the posterior development of a full High Level Trigger 1 realization on GPUs (cref. chapter 8). It also demonstrates local tracking methods are not inherently inefficient on parallel architectures, and sets a design pattern that can be implemented for other tracking problems (cref. chapter 6). Search by triplet is the current state-of-the-art in LHCb Velo tracking in terms of performance.

6

FORWARD TRACKING

AFTER tracks with Velo and UT hits have been reconstructed, the *forward tracking* problem consists in extending those tracks with data from the scintillating fibre tracker. As has already been introduced in section 1.1.1, the SciFi subdetector is placed after the magnet in the forward direction, and it constitutes the last stage of the tracking system of LHCb. Particles are deviated by the magnet in their path to the SciFi detector. Figure 6.1 shows the tracking system with the trend of the magnetic field.

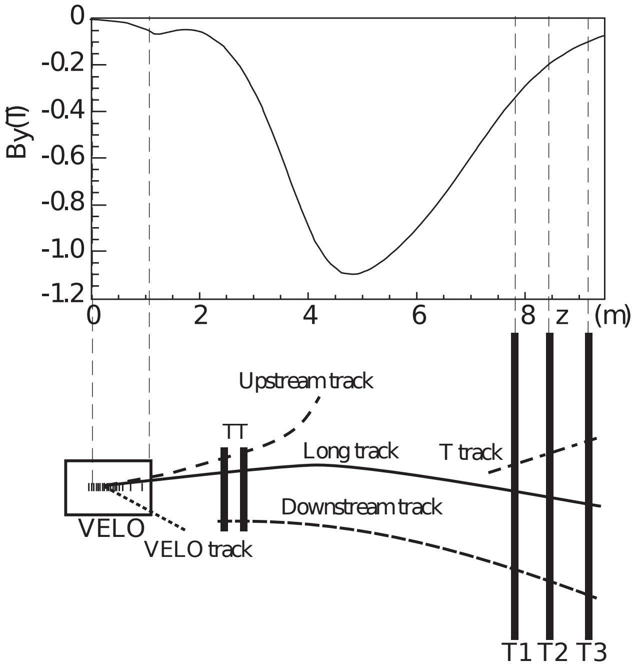


Figure 6.1: Tracking system of LHCb with magnetic field strength profile [70].

As particles cross through the SciFi detector, scintillating fibres in their path fire and the signals generated are detected by SiPM modules. SiPM modules can also produce thermal noise, and thus the thresholds to form measurements takes this effect into account. Figure 6.2 shows the values attained with various cluster threshold configurations, as a function of the SiPM position (shown as *SiPM channel*). The hit efficiency of the detector determines the probability a measurement will be produced given a particle leaves a signal, and it is a characteristic of the detector. The designed efficiency of the SciFi detector is $98.66 \pm 0.04\%$ [51].

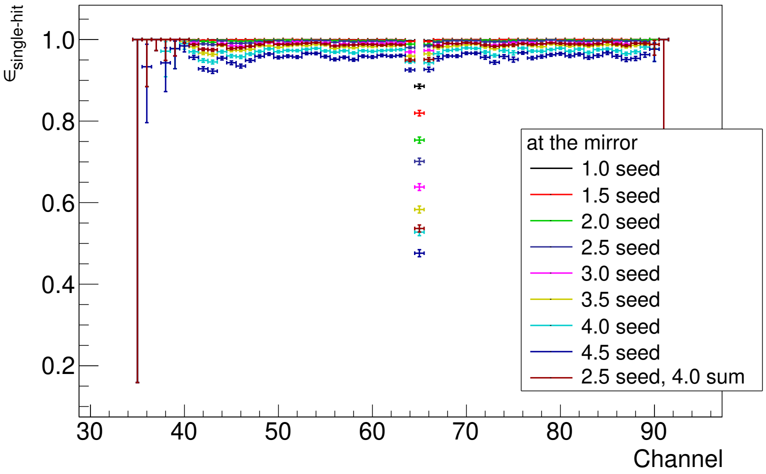


Figure 6.2: Hit efficiencies for different cluster thresholds.

The SciFi detector consists in three stations, each equipped with x, u, v, x layers. Energy deposits yield a precise x measurement, whereas the y component utilizes the $-5°$ and $5°$ tilt of the u and v layers. Particles cross the detector spanning its entirety, but the measurements that conform the input to the algorithm lose the y information, with the exception of knowing the upper or lower half was fired. In addition, the number of SciFi measurements stemming from tracks that left signals in both Velo and UT (*long tracks*) conforms roughly $\frac{2}{3}$ of the data [9].

All the factors above pose a real-time challenge in the LHCb reconstruction. Several tracking algorithms have been produced in the last few years in the LHCb codebase to solve this problem in face of the upgrade. A histogramming method *Seeding x layers* was developed in 2014 [71]. Significant improvements were brought with *Hybrid Seeding* [9], which follows a modular approach with several *cases*, providing a configurable variety of physics and computing constraints.

A parallel track forwarding method has been developed. A similar method to *Search by triplet* is employed by requiring triplets from *neighboring* x layer candidates, and forwarding tracks to other x layers and u, v layers. The code has been homogeneized avoiding branches and RAW control dependencies, and completes the GPU tracking system.

6.1 HISTOGRAMMING METHOD

The sequential LHCb reconstruction routine *Seeding x layers*, structurally similar to *Hybrid Seeding*, consists of the following stages:

- *x-hit preselection* – For every Velo-UT track, a candidate window is created by requiring a minimum transverse momentum (p_t). The momentum, charge and direction of the track are taken into account when creating the candidate window, resulting in a single side of the SciFi detector being explored for each track, reducing combinatorics by half. A list of candidate hits for each of the six *x layers* is built by requiring for each hit a compatible hit in the immediate U or V layer in its vicinity. This requirement implies the effective hit efficiency of each x layer is the hit efficiency of the x layer multiplied by the hit efficiency of the neighboring u/v layer.
- *x-hit cluster search* – The momentum estimation from the UT does not yield precise enough information to predict within the candidate window the expected trajectory of the particle. Therefore, all candidates are extrapolated to a reference plane, where they are accumulated. Figure 6.3 depicts this process. Each of the hits in the window are projected onto the reference plane, producing the histogram on the right of the image. A threshold is defined in the histogram, producing track candidates.

  The extrapolation to the reference plane is made taking into account the effect of the magnetic field. The effect of the magnetic field can be calculated by integrating the path with the equations of motion through a magnetic field B of a particle with a momentum p, charge q and velocity v,

  $$\frac{d\vec{p}}{dt} = q\vec{v} \times \vec{B}$$

  A mathematical simplification is achieved by expressing the trajectory of the particle as two linear paths with a *kick* in the magnet center. The method is depicted in Figure 6.4, showing the kick at position z_M. This method

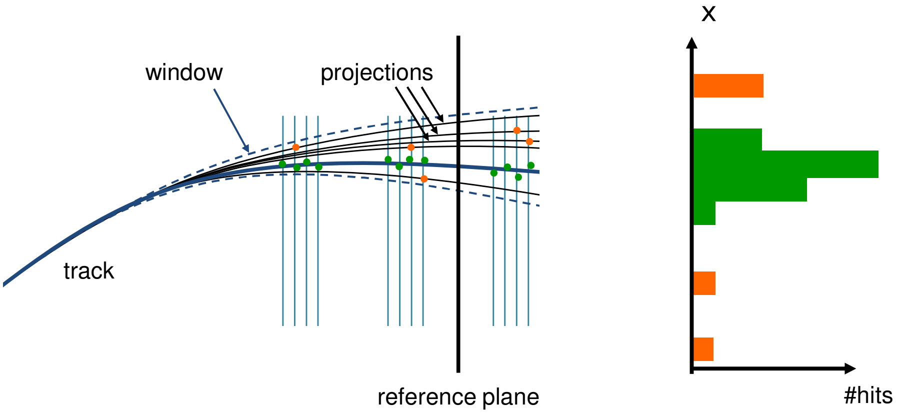


Figure 6.3: 1D histogram formed by accumulating measurements extrapolated in a reference plane.

permits calculating the trajectory of the particle through a homogeneous magnetic field. However, given the *fringe field* or peripheral conditions of the LHCb magnetic field, assuming a homogeneous magnetic field would lead to imprecissions of up to 20 cm in the particle trajectory. Instead, an empirical formula obtained from fitting Monte Carlo simulation data is used, involving the magnet center z_M and the track parameters t_x and t_y, in order to determine the $\Delta\mathtt{slope}$ (kick angle) of the particle,

$$z_M = \mathtt{zMagPar1} + \mathtt{zMagPar2} \cdot \Delta\mathtt{slope}^2 \\ + \mathtt{zMagPar3} \cdot t_x^2 + \mathtt{zMagPar4} \cdot t_y^2$$

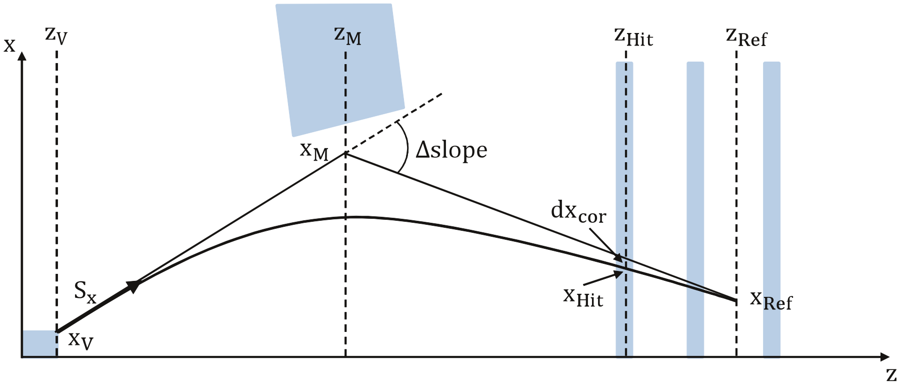


Figure 6.4: *Kick-method* for estimating the trajectory of a particle bent by a magnetic field.

- *Fit of x projection* – Hits from the 1D histogram are grouped iteratively with a *sliding window*. The group must have at least 5 hits and a maximum of 10 hits, and hits must not be over a threshold distance in the reference plane. Groups of contiguous hits that contain at least 4 hits on different layers constitute *track candidates*. The 1D histogram can be iterated several times depending on algorithm configuration flags. Track candidates found in previous iterations are flagged, to avoid being revisited in following iterations.

One hit in the candidate is chosen to obtain an updated charge q and momentum p of the track. The track candidate is then fitted linearly, and the worse χ^2 contributors over a predefined threshold are removed in an iterative *outlier removal*. Hits on missing layers in the candidate are added and fitted. A cubic fit is finally done on the resulting track candidate, imposing thresholds on the resulting χ^2.

- *Addition of u/v hits* – Even though u/v hits were required in the x-hit preselection stage, these hits were not further used in the creation of the track. u/v hits are selected by projecting them onto the x-axis and requiring them to be within a tolerance window. Then, the hit minimizing the distance to the expected x of the track fit x_{track} is kept, that is, minimizing $dx = x_{hit} - x_{track}(z_{hit})$.

  A fit of the stereo hits is also performed once the track has been fully built. A check is performed by which the extrapolated y from the Velo track to z_M should match the extrapolated y from the track hits. That is, $|y_{Velo}(z_M) - y_{track}(z_M)|$ should be under a threshold.

- *Complete fit and ghost / clone killing* – If all the above conditions are met, a complete fit of the resulting track is performed. The fit is done using the extrapolated coordinate onto the *reference plane*, similarly to the x projection fit. However, it includes all u/v hits x projections. After fitting the track, an iterative outlier removal is performed once more.

  Even though the reconstructibility condition of the SciFi detector requires only a total of 6 hits, Monte Carlo simulations show that in most cases the number of hits in a track is over 9 hits, as Figure 6.5 shows. Therefore, only tracks with at least 10 hits are kept. No clones are encountered by definition by only keeping one track per Velo-UT track.

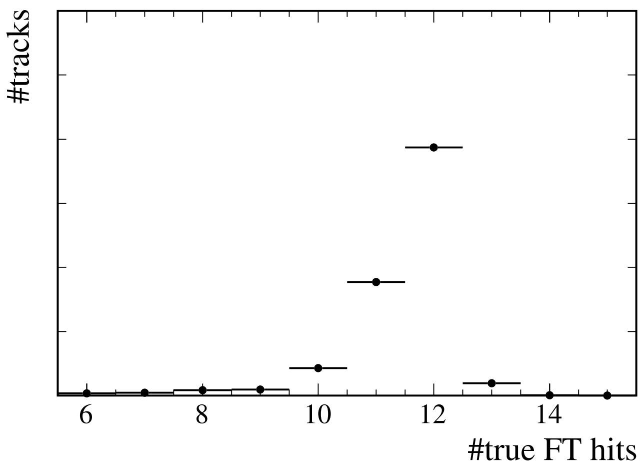


Figure 6.5: Number of measurements in SciFi detector for particles with $p > 5MeV$.

Finally, ghost rate is reduced by applying an Artificial Neural Network discriminator. The discriminator is fed with Monte Carlo training data, with features pertaining to the qualities of the track, such as the difference in the extrapolated location from the Velo to the track in x $|x_{Velo}(z_M) - x_{track}(z_M)|$, and in y $|y_{Velo}(z_M) - y_{track}(z_M)|$, the number of hits in different planes, and $\Delta\frac{q}{p}$. The quality of the discriminator is used finally to accept or reject the track.

6.2 LOOKING FORWARD

The LHCb Forward tracking problem requires dealing with numerous unknowns with respect to tracks momenta, hit inefficiencies and noise. The sequential solution processes tracks iteratively, with a code consisting in many branches that does not bode well with data-parallel processors. The sequential implementation was ported to CUDA by D. vom Bruch and V. Gligorov, confirming the hypothesis a new approach was necessary to run efficiently on parallel processors.

In order to efficiently use data-parallel processors, an algorithm that processes a homogeneous workload was instead developed, where all threads would follow the same control path, avoiding execution divergences in as much as possible. The *Looking Forward* method consists in twelve steps, as is shown in Figure 6.6. Each box represents an algorithm. Red boxes represent selection algorithms.

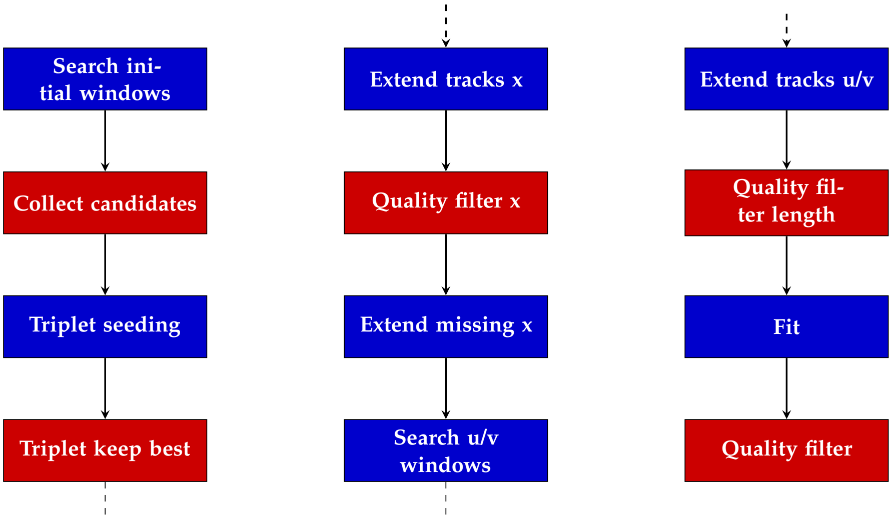


Figure 6.6: Forward tracking implementation *Looking Forward*.

In spite of the apparent increase in complexity, the presented design is modular and every algorithm has confined and well-defined tasks. The division of work into pieces allows optimizing kernel call configurations independently and decreases register usage.

The Looking Forward algorithm has been developed in close collaboration with F. Pisani and D. vom Bruch. The contributions of this thesis are the original parallel design of the entire algorithm into the steps presented here and the triplet seeding, as well as many optimization iterations to the original design.

Search initial windows, Collect candidates

Windows in x layers are formed by extrapolating UT tracks, taking into consideration the direction, momentum and charge of the tracks. Candidates are required to have a matching hit on the immediate neighboring stereo layer on the same station, similarly to the sequential approach.

A hard limit on the number of candidates is set to be 32 per x layer. Candidates are sought from the middle outwards, following a pendular order, giving preference to hits closer to the extrapolated position in average. Each track may therefore have up to $32 \cdot 6$ candidates.

The notation used in the following denotes by n the number of events under execution, and by t the average number of tracks in each event. The complexity of both Search initial windows and Collect candidates is therefore $O(n \cdot t)$.

Triplet seeding

Neighbouring layers in x are checked three by three for the existence of compatible triplets with a parabolic fit. Given there are six x layers and denoting them by ascending order, this implies searching for triplets in layers $\{0, 1, 2\}$, $\{1, 2, 3\}$, $\{2, 3, 4\}$ and $\{3, 4, 5\}$.

For any consecutive three layers, a parabolic fit χ^2 is minimized. In order to simplify the following arithmetic section, any three consecutive layers are denoted by 0, 1 and 2, with a hit on each with coordinates $h_0 = \{x_0, z_0\}$, $h_1 = \{x_1, z_1\}$ and $h_2 = \{x_2, z_2\}$ respectively. For every hit in layer 1, the hits in the first and last layer minimizing the following function are sought,

$$\begin{aligned}
\chi^2 &= \sum dx_i^2 \\
dx_i &= x_i - (x_0 + t_x \cdot dz_i + \text{param} \cdot \text{qop} \cdot dz_i^2) \\
dz_i &= z_i - z_0 \\
t_x &= (x_1 - x_0)/dz_1
\end{aligned}$$

The `param` is obtained empirically through Monte Carlo samples, and qop refers to the $\frac{q}{p}$ of the track. The above function can be simplified taking into account parameter canceling,

$$\begin{aligned}
dx_0 &= x_0 - (x_0 + t_x \cdot dz_0 + \text{param} \cdot \text{qop} \cdot dz_0^2) \\
&= x_0 - x_0 - t_x \cdot (z_0 - z_0) - \text{param} \cdot \text{qop} \cdot (z_0 - z_0) \\
&= 0
\end{aligned}$$

$$\begin{aligned}
dx_1 &= x_1 - (x_0 + t_x \cdot dz_1 + \text{param} \cdot \text{qop} \cdot dz_1^2) \\
&= x_1 - (x_0 + ((x_1 - x_0)/dz_1) \cdot dz_1 + \text{param} \cdot \text{qop} \cdot dz_1^2) \\
&= x_1 - x_0 - x_1 + x_0 - \text{param} \cdot \text{qop} \cdot dz_1^2 \\
&= -\text{param} \cdot \text{qop} \cdot dz_1^2
\end{aligned}$$

$$
\begin{aligned}
dx_2 &= x_2 - (x_0 + t_x \cdot dz_2 + \text{param} \cdot \text{qop} \cdot dz_2^2) \\
&= x_2 - x_0 - ((x_1 - x_0)/dz_1) \cdot dz_2 - \text{param} \cdot \text{qop} \cdot dz_2^2 \\
&= x_2 - x_0 - x_1 \cdot dz_2/dz_1 + x_0 \cdot dz_2/dz_1 - \text{param} \cdot \text{qop} \cdot dz_2^2
\end{aligned}
$$

Yielding the following χ^2 formulation,

$$
\begin{aligned}
\chi^2 &= (-\text{param} \cdot \text{qop} \cdot dz_1^2)^2 \\
&\quad + (x_2 - x_0 - x_1 \cdot dz_2/dz_1 + x_0 \cdot dz_2/dz_1 - \text{param} \cdot \text{qop} \cdot dz_2^2)^2
\end{aligned}
$$

Triplets are processed iterating over neighboring layers, three by three. The qop term is UT track dependent. Therefore, the following expressions can be precalculated for all triplets in three neighboring layers for a UT track,

$$
\begin{aligned}
\text{extrap}_1^2 &= (-\text{param} \cdot \text{qop} \cdot dz_1^2)^2 \\
dz_2dz_1 &= dz_2/dz_1 \\
\text{extrap}_2 &= \text{param} \cdot \text{qop} \cdot dz_2^2
\end{aligned}
$$

so the χ^2 can be simplified as a function of x_0, x_1 and x_2,

$$
\chi^2 = \text{extrap}_1^2 + (x_2 - x_0 - x_1 \cdot dz_2dz_1 + x_0 \cdot dz_2dz_1 - \text{extrap}_2)^2 \tag{6.1}
$$

All combinations of triplets from the candidates created in the previous step are calculated. This amounts to a total of $4 \cdot 32^3 = 131072$ χ^2s for every UT track. Two additional optimizations have been done to cope with the high number of calculations,

- A tiled data access pattern is employed when calculating triplets. All combinations between any hits in the first and last layer in the triplet are precalculated, in batches of 16 × 16 hits. Following equation 6.1, and for a pair of hits $\{h_0^i, h_2^j\}$,

$$
\text{partial\_chi2} = x_2^j - x_0^i + x_0^i \cdot dz_2dz_1 - \text{extrap}_2 \tag{6.2}
$$

$$
\chi^2 = \text{extrap}_1^2 + (\text{partial\_chi2} - x_1 \cdot dz_2dz_1)^2 \tag{6.3}
$$

Partial chi2s are precalculated following equation 6.2. For every hit h_l^k, the partial chi2 is incorporated into equation 6.3 to obtain the final χ^2.

This method preserves memory locality, as hits are likely close in memory. Partial chi2s are stored in shared memory, increasing throughput. Given that all triplets are calculated, the implementation follows a clear execution path with few branches, efficiently processing many χ^2s at a time.

- Tensor cores [72] are specialized functional units in modern NVIDIA GPUs that allow performing fast mixed precision *fused multiply-add* (FMA) operations. Tensor cores can be programmed with CUDA, and allow performing the matrix operation $D = A \times B + C$, for matrix sizes 16×16, 8×32 or 32×8[1]. Matrices A and B must contain half precision floating point numbers[2], and matrices C and D can either (1) be single precision matrices, in which case the operation $A \times B + C$ is performed and stored in single precision, or (2) be half precision matrices, in which case the operation is performed and stored in half precision.

  The partial chi2 arithmetic has been expressed using Tensor cores. Mixed precision mode is used (1). The precision lost by requiring matrices A and B be expressed in half precision are negligible and do not affect the results significantly. Hits are processed in a tiled manner, 16×16 at a time.

  Tensor cores allow us to calculate χ^2s up to 15% faster on GPU models supporting them. A fallback standard code has been developed for compatibility with older GPU models, and to allow translations to other architectures not featuring this specialized functional unit.

[1] Tensor cores are programmable since CUDA version 9.0 on NVIDIA GPUs of major version at least 7. The shown matrix sizes are allowed since CUDA 9.1.

[2] 16-bit (half precision) floating point, alongside 32-bit (single precision) and 64-bit (double precision), are numerical and arithmetical standards specified by IEEE 754.

$$A = \begin{pmatrix} 1 & -x_0^{i} & x_0^{i} & 0 & \dots & 0 \\ 1 & -x_0^{i+1} & x_0^{i+1} & 0 & & \\ 1 & -x_0^{i+2} & x_0^{i+2} & 0 & & \\ 1 & -x_0^{i+3} & x_0^{i+3} & 0 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ 1 & -x_0^{i+15} & x_0^{i+15} & 0 & & 0 \end{pmatrix}$$

$$B = \begin{pmatrix} x_2^{j} & x_2^{j+1} & \dots & x_2^{j+15} \\ 1 & 1 & \dots & 1 \\ dz_2 dz_1 & dz_2 dz_1 & \dots & dz_2 dz_1 \\ 0 & 0 & \dots & 0 \\ \vdots & & \ddots & \\ 0 & & & 0 \end{pmatrix}$$

$$C = \begin{pmatrix} -extrap_2 & \dots & -extrap_2 \\ \vdots & \ddots & \\ -extrap_2 & & -extrap_2 \end{pmatrix}$$

$$D = A \times B + C$$

The best triplet with minimum χ^2 for each middle hit is kept. A maximum of $4 \cdot 32$ track seeds per UT track are formed. The number of computations is constant, since all triplets are always checked for every UT track. Hence, computational complexity of the triplet formation is $O(n \cdot t)$.

Track formation in x layers

Triplets created in the previous step are sorted by χ^2, and the best are kept in algorithm `Triplet keep best`. The number of triplets to preserve is configurable – the configuration is a trade off between physics efficiency and performance. Insertion sort is used, with a shared memory buffer as temporary storage. The $\frac{q}{p}$ of the tracks is now updated, with the measurements from the SciFi detector. The best triplets are projected onto missing x layers in algorithm `Extend tracks x`, only considering the collected hits in step two, `Collect candidates`.

`Quality filter x` filters tracks by requiring at least 4 hits. All hits are projected onto a reference plane analogous to Figure 6.3, and tracks are sorted in decreasing order by the *spread* on the reference plane. The best two tracks per UT track are kept at this stage. Finally, since only a few tracks remain after this cut is applied, track hit efficiency is recovered by exploring all hits in the missing x layers, in algorithm `Extend missing x`.

Each of these processes is parallelizable. Individual events are assigned to different blocks of threads. Intra-event parallelism is exploited in the sort algorithm of `Triplet keep best`, and best tracks are chosen simultaneously immediately afterwards. Multiple tracks are projected and forwarded in parallel in `Extend tracks x`, considering hits in layers in a data-parallel manner. The projection and spread calculation in `Quality filter x` is performed in parallel. Finally, multiple tracks are extended in parallel, iterating over missing layers in a data-parallel fashion in `Extend missing x`.

For each UT track, the best track according to a fitting condition is kept in `Triplet keep best`. `Extend tracks x` may extend tracks at most 5 iterations, over a constant number of hits each time. `Quality filter x` selects tracks based on a constant time calculation of the spread of the hits. The complexity of these procedures is $O(n \cdot t)$. `Extend missing x` considers hits in missing layers, for a maximum of 5 iterations, performing a binary search to find compatible hits. The average number of hits in a layer is denoted as m. Even though a binary search is used to find the first and last candidate, in the worst case all hits in the layer will be traversed, thus the complexity of `Extend missing x` is $O(n \cdot t \cdot m)$.

Track forwarding to stereo layers

Tracks are forwarded to the u/v layers taking into account the $\pm\ 5^\circ$ tilt of the layers. The Velo track is used to predict the position in y, as is shown in Figure 6.7. A similar method to the histogramming extrapolation is used, by combining the prediction in x with the Velo extrapolation in y, *y(track)*.

The search is done in two steps. Compatible hits are searched through a binary search with a tolerance window in `Search u/v windows`. Tracks are then extended to the best fitting candidate in `Extend tracks u/v`. Both the search and the extension

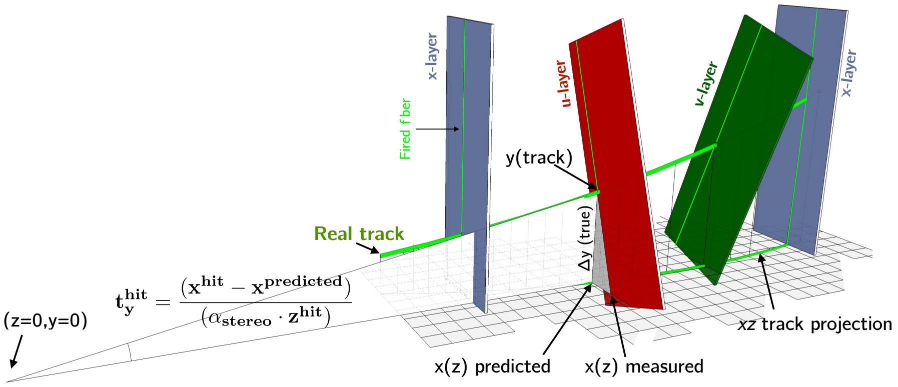


Figure 6.7: Track extrapolation to u/v layer. Image from [9].

of tracks iterate all six stereo layers, and consider all hits on the track region. Even though it is possible that tracks of an upper region produce measurements on to the lower region, the current tracking method does not account for this effect. This feature, commonly referred to as *triangle search*, will be considered in future iterations of Looking Forward.

Both methods exploit intra-event parallelism by assigning tracks to CUDA threads. For every track, six binary searches are performed in the window search, and all windows are further used to extend tracks. The separation in two algorithms favors code branching homogeneity. The complexity of the search is $O(n \cdot t \cdot \log(m))$. The extension in the worst case is $O(n \cdot t \cdot m)$.

Track selection

Fully built tracks are selected by first requiring a minimum number of hits in algorithm `Quality filter length`. In spite of the evidence supported by Figure 6.5 suggesting a lower bound of 10 hits, tracks with multiple hits on the same layer are not allowed – this limitation is circumvented by requiring at least 9 hits. The remaining stages of the algorithm are similar to the sequential incarnation. A full fit of the remaining tracks is performed in `Fit`, and the set of parameters defining the tracks is passed on to an ANN discriminator in `Quality filter`.

The track selection is parallelized assigning CUDA threads to tracks. The complexity of the `Quality filter length` and `Fit` are $O(n \cdot t)$. The topology and weights of the ANN in `Quality filter` are static and its amortized time contribution are constant. Therefore, the `Quality filter` is $O(n \cdot t)$.

7 KALMAN FILTER

KALMAN devised an algorithm [73] that is capable of estimating the *state* of an object, integrating information both from observations in the trajectory of the object, and from a mathematical description of the process driving the propagation of the object, known as the *Kalman filter*. In its discrete form, the Kalman filter can be described as a process consisting of two stages applied iteratively, *predict* and *update*.

The LHCb use case and terminology are described in the following. Particles that propagate through the detector at a certain z position along the beam line are described with a 5-element state $\hat{x}$,

$$\hat{x} = \{x, y, t_x, t_y, \frac{q}{p}\} \tag{7.1}$$

where x, y are the position of the particle in the X and Y planes, t_x and t_y correspond to the slope in X and Y in the parametric formulation of the line at position z, q refers to the charge of the particle, and p to its momentum. In addition, the error associated with the state is defined by a 5×5 symmetric matrix P known as the covariance matrix, which stores the correlation between all elements in the state.

Starting with a seed state, the Kalman filter alternates predicting and updating the state of the particle using detector measurements. In the Kalman filter predict stage of location k, the predicted state of a particle with state $\hat{x}_{k-1|k-1}$ and covariance $P_{k-1|k-1}$ is calculated:

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k \tag{7.2}$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k \tag{7.3}$$

F_k is the *transport matrix* that transforms the previous state and covariance into the predicted ones. External factors are considered by including a control matrix B_k and a control vector u_k. External uncertainty is considered in the Q_k term, also known as the *noise matrix*, which refers to the noise caused by multiple scattering in the detector material.

The Kalman filter update stage incorporates a measurement into the prediction of the state of the particle,

$$
\begin{aligned}
z_k &= H_k x_k + v_k && (7.4)\\
K_k &= P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} && (7.5)\\
\hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k (z_k - H_k \hat{x}_{k|k-1}) && (7.6)\\
P_{k|k} &= P_{k|k-1} - K_k H_k P_{k|k-1} && (7.7)
\end{aligned}
$$

At location k, the measurement z_k is observed (eq. 7.4), where x_k is the true state, and v_k is the observation noise, assumed to be Gaussian-distributed centered at zero with an uncertainty R_k. H_k is the *projection matrix*, which projects measurements into the space of the state. The *Kalman gain* K_k determines the relative weight of the measurement with respect to the prediction.

The prediction and update may be applied as many times as needed, and yield an updatable state of the particle. The Kalman filter is used to estimate the goodness of a track and to determine a good linear estimate of the track parameters in regions of the detector. The Kalman filter may be performed in the *forward* or *backward* direction. The process of averaging states obtained from the forward and backward Kalman filter is known as *smoothing*. It is possible to iteratively input the smoothed state as the seed state of the Kalman filter, in order to improve the quality of the estimation.

The Kalman filter is a process utilized at several stages of the LHCb reconstruction. Since not all stages require the same degree of precision, simplified Kalman filter models exist, which trade off precision for speed. The author however was involved with optimizing the full-fledged Kalman filter, with the objective of improving hardware resource usage while delivering the canonical formulation.

The Kalman filter has been parallelized in a number of works. In [74], a parallel SIMD Kalman was developed optimized for vector processors, used in the track finding and track fitting procedures of the CBM experiment. In [75], track building is sped up by employing a vectorized Kalman filter. In this thesis a cross-architecture implementation of the Kalman filter is considered for the track fitting procedure, optimizing resource usage by employing a static scheduler.

The work carried out by the author has been presented iteratively in various conferences. In [76], the initial design and a comparison between CPU architectures is shown. An Intel Xeon CPU, Power8 CPU and Intel Xeon Phi architectures are considered. The performance of both single and double precision was analyzed. In [77], a more fine-grained study over Intel processors was carried out in collaboration with O. Awile and O. Bouizi, including modern Intel Skylake processors. In [78], an implementation for the LHCb Gaudi framework was developed, validating the results and evaluating the performance speedup obtained in the framework run conditions.

The publication [2], included as part of this thesis in appendix B, extensively reviews the method employed in the developed algorithm, and concisely presents the conclusions of the work carried out by the author in the Kalman filter algorithm. The rest of this chapter discusses the presented results.

7.1 discussion

Three independent Kalman filter implementations have been developed. *Cross-Kalman mathtest*[1] is a cross-architecture program, implemented in four different language technologies, including two vectorization libraries (UMESIMD and VCL), and implementations written in CUDA and OpenCL. The three Kalman filter implementations are compilable in a configurable floating point precision of 32-bit or 64-bit. The program has been tested in all the available technologies to the author, including x86, Power8 and ARM processors, and NVIDIA and AMD GPUs. The *mathtest* project implements solely the Kalman filter math, and serves as a demonstrator of the efficiency of the mathematical formulation with respect to the data requirements across architectures. As the roofline models show, the developed algorithm makes optimal usage of the resources under the data requirements imposed.

[1] The three projects are publicly available. Cross-Kalman mathtest: `https://gitlab.cern.ch/dcampora/cross_kalman_mathtest`. Cross-Kalman: `https://gitlab.cern.ch/dcampora/cross_kalman`. TrackVectorFitter: Available as part of `https://gitlab.cern.ch/lhcb/Rec`.

In addition, the project serves as an indication of the expected performance speedup attainable by the various architectures under analysis, by producing an equally optimized code for multi and many-core architectures. While it is a synthetic benchmark, it is a significative result in the context of physics reconstruction.

Cross-Kalman is a multi-core cross-architecture implementation of the Kalman filter algorithm inspired by the LHCb Gaudi *TrackMasterFitter* package requirements. This program serves as a self-contained analysis reproducing the conditions of the Kalman filter in the LHCb Gaudi framework. While preserving the data dependencies in the Kalman filter formulation, multiple particles are reconstructed in parallel, assigning *predict* and *update* primitives to computing resources (elements in vector units). A complex workload is considered by supporting the forward and backward Kalman filter stages and the smoother, with a control scheduler. Cross-Kalman results are validated by supporting LHCb Monte Carlo data and analyzing the deviation in the results obtained.

TrackVectorFitter is a Gaudi framework realization of the vectorized Kalman filter. It is a 1:1 replacement of the sequential *TrackMasterFitter*, optimized for vector units. The vectorized filter has been validated against Monte Carlo samples to produce equivalent results to the sequential filter.

The presented Kalman filter work is orthogonal to the development of the GPU sequence in this thesis. The GPU High Level Trigger 1 sequence includes a simplified Kalman filter instead. During the prediction step, the nominal LHCb Kalman filter employs a magnetic field map and a Runge-Kutta extrapolator for track propagation and a detailed detector description for determining noise from multiple scatterings. The simplified Kalman filter, on the other hand, replaces these calculations with parameterizations. Due to the nature of the physics requirements, the simplified Kalman filter results in a faster alternative for the HLT1 use case.

The performance obtained by the vectorized Kalman filter affects High Level Trigger 2 configurations of the LHCb Gaudi sequence. The vectorized filter reduces the processing time of the Kalman filter arithmetic to less than half, which results in speedups of up to 1.09x in the entire sequence. The change in data structures required by the vectorized filter, from AOS to AOSOA, improves locality and data access patterns, and serves as ground work for further improvements in components related to the Kalman filter, such as track propagation. Under the data requirements of the full mathematical formulation, the developed Kalman filter arithmetic obtains the theoretical peak performance of the processors analyzed.

Part III

FRAMEWORK

8

A FRAMEWORK FOR MASSIVELY PARALLEL PHYSICS RECONSTRUCTION

Fast algorithms on hardware accelerators have been produced before for the LHCb physics use case (cref. section 2.3.1). The implementation of the OpenCL Velo tracking was up to 1.53× faster than the reference implementation utilizing two *Intel Haswell E5-2630* CPUs.

A key requirement to achieve a speedup over the reference implementation was to run multiple events in parallel. Figure 8.1 shows the speedup obtained with the LHCb OpenCL Velo reconstruction as a function of the number of events in flight. GPU performance was significantly better when the number of events in flight was in the thousands.

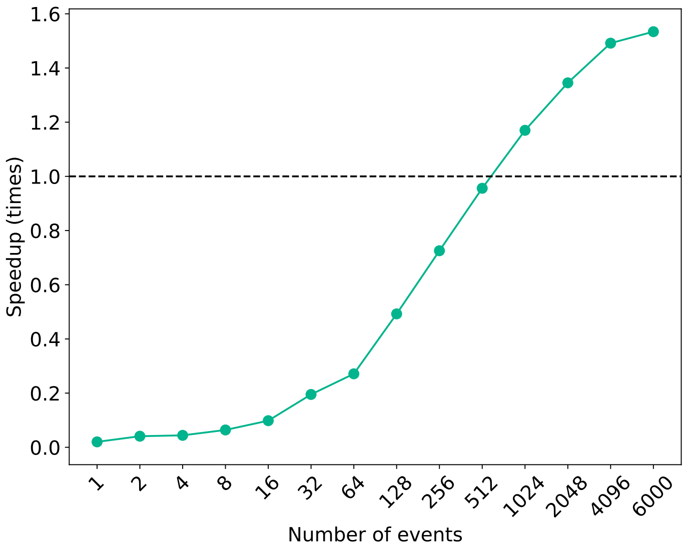


Figure 8.1: Speedup of OpenCL Velo tracking as a function of the number of events executed in parallel.

Since only the Velo tracking was developed in a first instance, data dependencies enabling the execution of the algorithm would be run using an LHCb *Gaudi application*. Prerequisites would be processed on the host, then the computation would be offloaded to the accelerator, and the result would be integrated back into the application upon completion of the offloaded work. In the Velo tracking case, the host would decode the detector data and prepare a binary blob to send to the accelerator, the accelerator would process the job and return Velo tracks. Finally, the host would populate Velo tracks in the required format. An offload engine for Gaudi known as the *Coprocessor manager* [79] was developed to this end, which enabled accelerators to interact with any Gaudi application.

In spite of these promising results and the development of the Coprocessor manager, fundamental design choices impeded any further developments. The OpenCL implementation would require to run thousands of events in parallel in order to have an adequate workload for GPUs to be efficient. However, Gaudi was designed to process individual events by each thread, one at a time. This design decision favors data locality, since event data of a single event has more chances to fit within cache memory limits. Given that the Gaudi HLT applications are memory bound, data locality severely impacts their performance. Therefore, the design requirements of Gaudi and offloaded GPU algorithms are irreconcilable, and other solutions must be sought.

In order to overcome the limitations encountered in Gaudi with respect to hardware accelerator utilization, a framework for massively parallel physics reconstruction has been created. Allen[1] is an extensible and modular software framework to perform the LHCb High Level Trigger 1 on hardware accelerators like GPUs. In its conception, it is meant to serve as a demonstrator of the feasibility of such a trigger in view of the upcoming LHCb Upgrade.

The framework is written in C++ with CUDA extensions. The CUDA language has been chosen as it has received continuous support from NVIDIA during the last ten years and provides a compelling development set of tools. The minimum CUDA capability required of Allen is the minimum supported by the latest release of the `nvcc` compiler. Some of the developed algorithms contain instructions that require higher compute capability. However, even in those cases an equivalent compatibility version is provided to maintain support for older graphics cards. This design decision ensures Allen is not locked to features only

[1] Allen is publicly available at `https://gitlab.cern.ch/lhcb-parallelization/allen`.

available in the CUDA language, allowing future translations to other GPU programming languages or middleware languages supporting a broad variety of accelerators [80].

8.1 FRAMEWORK DESIGN

Allen requires the presence of one CUDA-capable device in order to run. The Allen framework is multithreaded, and it utilizes the standard library `std::thread` functionality. Each thread spawns a CUDA *stream* in order to offload computation to the GPU. Each thread processes multiple events in parallel. Since events are independent to one another, there is no need for inter-thread or inter-stream communication.

Figure 8.2 presents an overview of the design of Allen. Each box represents a class. Singletons are represented with a single box, whereas classes with multiple instances are represented by stacked boxes. Lines represent relations between the objects. A description of the function of each item is presented. Certain relations are transitive and require more exhaustive explanations, which will be extended upon in dedicated subsections *Control flow* and *Data flow*.

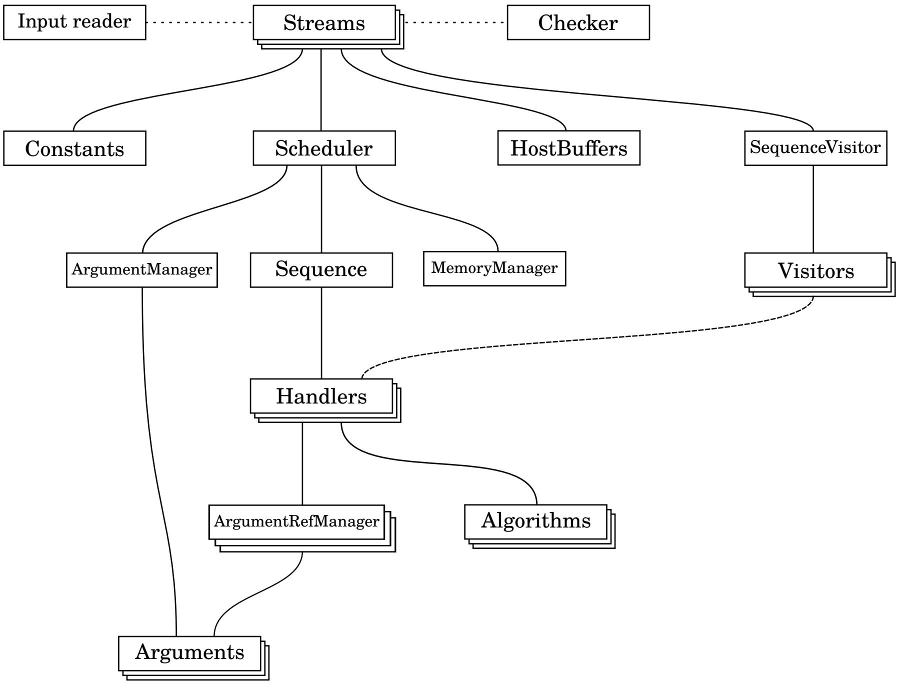


Figure 8.2: Overview of design of Allen.

- **Input reader** – Event raw data is supported through two formats: A custom LHCb format `MDF`, and a custom Allen binary format. Both can be produced with the Gaudi framework from Monte Carlo simulated particle collisions. The Monte Carlo truth[2] of the subdetectors involved in HLT1 are also read.
- **Checker** – A checker using Monte Carlo truth validates the reconstruction sequence within the framework. The functionality of the checker has been validated against the checker in the Gaudi framework.
- **Streams** – A GPU is able to execute programs, receive data and transmit data at the same time. The Allen application spawns a configurable number of threads to run the application, where each thread utilizes a CUDA stream to communicate with the GPU and run algorithms, in an non-blocking asynchronous manner. An effective pipeline is created when Allen is executed with at least three CPU thread - GPU stream pairs, so threads can send data from host to device, receive data from device to host and execute in the device concurrently (see section 10.2.1).
- **Constants** – Geometry constants, static lookup tables, magnetic field constants and others are populated only once and distributed to each individual Stream. Constants are kept both in CPU and GPU memory throughout the execution of the Allen application.
- **HostBuffers** – Data between executions of algorithms are kept in GPU memory (cref. 8.3). However, the size of certain memory buffers cannot be determined statically, and depends on the execution of prior algorithms. The singleton HostBuffers allocates a set of pinned buffers on the host, to allow retrieval of any required memory buffers from the GPU in an non-blocking asynchronous manner. If the validation is enabled, HostBuffers also holds pinned memory buffers for all produced objects that will be validated, such as tracks, primary vertices or Kalman filter states.

[2] Events generated with Monte Carlo simulation contain both the signals from the LHCb instruments detecting the event, and detailed information about the particles produced in the event, including features such as their mass, trajectory, velocity and momentum. The first constitutes the input for the event reconstruction, whereas the latter is commonly known as the *Monte Carlo truth*, employed for checking the efficiency of the reconstruction.

- **Algorithms** and **Arguments** – Developers can write normal CUDA or C++ functions, which are abstracted by using Allen Algorithms and Arguments. Allen allows the creation of both CPU and GPU algorithms. However, Arguments refer only to GPU arguments.

- **Handler** – A Handler is defined by specifying a *handler identifier*, an Algorithm and a set of Arguments. CPU and GPU handlers can be created. Handlers abstract the calling convention of CUDA by providing methods to set the CUDA invocation parameters and the function arguments. Similarly, CPU functions are encapsulated in `std::function`. Handlers provide unique identifiers to Algorithms in Allen.

- **Sequence** – A Sequence of Handlers is specified by the developer and is statically configured at compile time. Allen provides an extensible list of Sequences. Once Allen is compiled, the generated application is only able to execute the Sequence it was configured with.

- **MemoryManager** – GPU memory is allocated once per Stream (cref. 8.3). A MemoryManager manages the available global memory on each Stream by maintaining a linked list of free memory segments, and providing `allocate` and `free` implementations.

- **ArgumentManager** – The ArgumentManager stores an *offset* and *size* for each Argument required in the Sequence of algorithms to execute. The size is determined at runtime by any of the algorithm Visitors, and the offset is known once the MemoryManager has allocated the required memory of the Argument.

- **ArgumentRefManager** – Each Handler refers to the Arguments by employing an ArgumentRefManager of their own. The ArgumentRefManager only has access to the Arguments required by the Handler. This mechanism prevents any Argument to be accessed after it has been deallocated from memory by the MemoryManager.

- **SequenceVisitor** and **Visitors** – The Sequence is visited by employing the Visitor pattern. The Visitor pattern is defined by *the Gang of Four* [81] as:

  "[The Visitor pattern] represents an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

Developers can write two Visitors for each Handler: The size of arguments can be explicitly set in `set_arguments_size`. The execution of Algorithms, alongside any memory tranmissions and data preparations are expected to be run in `visit`.

- **Scheduler** – The Scheduler combines all the above functionality to steer the Allen application. The Sequence of Handlers is recursively inspected to generate a list of *in dependencies* and *out dependencies*. In dependencies is the list of Arguments required to be allocated prior to the execution of every Handler. Out dependencies is the list of Arguments that can be freed prior to the execution of every Handler. For every Handler in the Sequence, the Scheduler performs the following tasks:

  - Employ the Handler Visitor `set_arguments_size` to set the size of any Arguments that should be allocated prior to the execution of the Algorithm.
  - Use the Out dependencies to free Arguments employing the MemoryManager.
  - Use the In dependencies to allocate Arguments employing the MemoryManager. The size of the Arguments to be allocated should have their `size` in the ArgumentManager populated. After allocation, each Argument will have their `offset` in the ArgumentManager populated.
  - Finally, employ the Handler Visitor `visit` to invoke the function and perform any required data manipulations.

  Since the Sequence and its data dependencies are specified using the Type machinery of C++, the Scheduler is generated at compile time.

8.2 CONTROL FLOW

Allen requires developers to encapsulate logic into functions, in the same manner they would write conventional C++ or CUDA algorithms, and expose them to the framework by creating Allen specific types Algorithms and Arguments. In addition, the sequence of algorithms to be executed must also specified. The necessity of specialized framework code is kept to a minimum in order to avoid idiosyncratic practices that would prevent developers from being efficient. The Velo sequence is presented in Listing 8.1. Each line is a Handler identifier.

```
SEQUENCE_T(
  init_event_list_t,
  global_event_cut_t,
  velo_estimate_input_size_t,
  prefix_sum_velo_clusters_t,
  velo_masked_clustering_t,
  velo_calculate_phi_and_sort_t,
  velo_fill_candidates_t,
  velo_search_by_triplet_t,
  velo_weak_tracks_adder_t,
  copy_and_prefix_sum_single_block_velo_t,
  copy_velo_track_hit_number_t,
  prefix_sum_velo_track_hit_number_t,
  consolidate_velo_tracks_t)
```

Listing 8.1: Velo sequence definition.

Allen executes the sequence of algorithms in the order set by identifiers in the sequence, ensuring arguments lifetime requirements are met. Each thread runs the sequence in one CUDA stream, and as a consequence algorithms do not overlap within the stream.

Each algorithm in the sequence operates on multiple events in parallel. The number of concurrent events processed is kept constant throughout the sequence execution, and may decrease when filtering algorithms are applied. Figure 8.3 presents a flowchart of the algorithms in the HLT1 sequence of Allen *v0.6*. Allen implements an equivalent sequence to that of the HLT1 physics program of the Run2 of the LHCb experiment [82]. The order of the sequence of algorithms to be executed must fulfill the data dependency chart of Figure 1.14. There are different orderings of the algorithm sequence that would fulfill that criterium. The one presented here reduces the memory footprint of the application by performing clustering and tracking right after decoding of each subdetector.

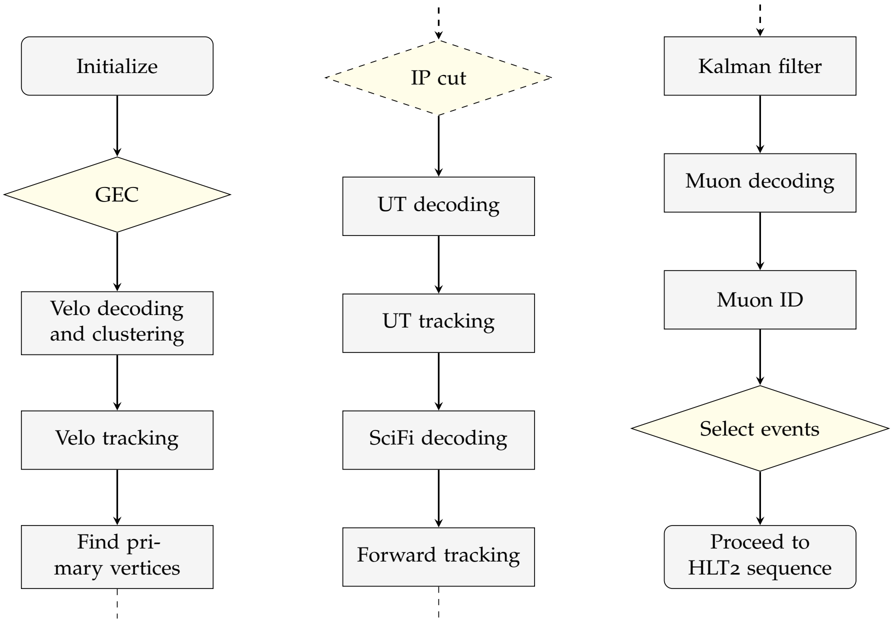


Figure 8.3: HLT1 control flow sequence overview.

Three decision algorithms are presented in the sequence. In those, data reductions are performed according to different criteria. The *global event cut* (GEC) only keeps events with less than or equal to 9750 hits combining the UT and SciFi subdetectors, which corresponds roughly to the busiest 10% of the events processed. The algorithm labeled as *IP cut* is disabled in the default sequence of Allen v0.6, depicted here as an optional algorithm with dashes in its surrounding box. The IP cut discards events with an *Impact Parameter*[3] over a selection threshold. The output of the sequence consists of a summary of the decision behind the selection of events alongside the selected events.

Simple and composite Handlers can be defined in the sequence. A composite Handler is defined with an identifier, a set of Handlers and a set of Arguments. Code repetition is avoided by encapsulating common tools such as the *prefix sum* or *sorting* into composite Handlers, which would otherwise require explicit individual instantiation and lead to a slower compilation due to the increased static analysis (see Figure 8.4).

Even though Allen is primarily a GPU framework, it is also possible to define CPU algorithms. Since Allen requires CPU

[3] The Impact Parameter of a produced particle refers to its distance in the XY plane to the collision vertex it is associated with.

threads to launch the computations in parallel streams, these CPU threads can potentially be used to run parts of the sequence. This allows developers to quickly develop and test CPU prototypes. As a side effect, it also allows to speed up computations by using CPU resources. In practice, CPU resources are considered *opportunistic resources* in Allen, and overcommitting to these impacts performance negatively. In the default sequence, only the GEC and the *Prefix sum* are available in CPU forms. These two algorithms are more naturally amenable to sequential processing, and are enabled to run on the CPU by default (`--cpu-offload=1`).

8.3 DATA FLOW

One of the central pieces of the design of Allen revolves around the idea of efficiently using memory, which addresses three issues found in GPUs:

- Memory available per core is very scarce.
- Thousands of events must be executed in parallel to obtain good performance.
- Memory allocation / deallocation is a blocking operation.

The amount of memory available varies between GPUs. Mid-end gaming graphics cards like the GeForce GTX 1060 exist in 3 GB and 6 GB GDDR5 memory models. High-end gaming cards like the GeForce GTX 1080 Ti and the GeForce RTX 2080 Ti come with 11 GB of GDDR5 memory. Scientific cards typically come equipped with more and faster memory. The low profile Tesla T4 features 16 GB of GDDR6, whereas the Tesla V100 comes in two versions, with 16 GB and 32 GB of High Bandwidth Memory (HBM2) respectively.

Regardless of the configuration, these numbers are around one order of magnitude behind the amounts of memory commonly seen in up to date servers, between 64 and 256 GB. The difference is bigger when considering the amount of memory available per core. In all the configurations tested throughout the elaboration of this thesis, servers were provided with O(1) GB per core, whereas the GPUs mentioned above have O(1) MB per CUDA core.

To give an estimate of the amount of memory required to process a single event, the Gaudi HLT applications use a fixed memory consumption of 517 MB, with an additional average 6.74 MB per thread [45] in a configuration where threads process events sequentially and individually. If the same approach would be taken on GPU, assigning the same amount of memory per CUDA thread, it would not be possible to run even one thousand events in parallel in some of the GPU configurations. The requirement of executing thousands of events in parallel competes with the necessity for a lower memory footprint in Allen.

In addition, memory allocation and deallocation is a blocking operation on GPUs. In other words, the entire device must be synchronized, stopping the operation on all streams and all transfers from and to GPU memory. Finally, even if a custom memory manager is provided, dynamic memory allocation within a stream would still require synchronization within all blocks and threads in the stream.

Allen provides a custom memory manager, implemented with the `MemoryManager` singleton. A fixed amount of data can be allocated upon startup with the `--memory` parameter. The configured memory amount will be allocated for each thread / stream pair, and defines the upper bound of memory that can be used at any point in the algorithm sequence. The memory manager can allocate and free memory independently on each thread within the bounds defined by `--memory`, set by default to 1 GB. Since the amount of memory is fixed, if the scheduler requirement surpasses the available memory, the application runs out of memory and cannot continue processing the events. Various contingency options are being studied to address out of memory situations, such as restarting the affected events execution in chunks of fewer events.

Dynamic memory allocations are not allowed within algorithm executions in Allen. As a consequence, the expected size of all buffers must be determined prior to the execution of algorithms. While most buffer sizes can be calculated, this introduces complications for certain buffers whose size is simply unknown prior to the execution of the algorithm. For subdetector decoding, a fast decoding is performed to calculate the size or estimate an upper bound of the memory required. For tracking, since repeatedly running tracking algorithms would be too slow, an upper bound is estimated using some subdetector-specific knowledge. This will be explored with more care in the algorithm sections 4.

Going through the tedium of defining static sizes for each and every one of the algorithms in the sequence pays off in terms of performance. Gaudi HLT applications currently spend around 14% of runtime in allocating / freeing resources [45]. The time spent in Allen is negligible, as will be discussed in subsection 8.4.

Last, Allen is required to execute thousands of events within a tight memory budget. This limitation was well known at the beginning of the design of Allen, and a common strategy was followed within the creation of every algorithm. Following the design philosophy of CUDA blocks and threads, the work was divided in two dimensions. Inter-event parallelism is exploited by independent blocks, whereas intra-event parallelism has been sought following a multitude of ad-hoc techniques to exploit the inherent parallelism of independent processes (cref. chapter 4).

Every algorithm was created from scratch for GPGPUs. Sequential techniques that would require memory buffers like the Velo clustering (cref. 3.1) were scrapped and new methods were developed. After the reconstruction of every subdetector, a consolidation step is performed to store only required information pertaining to tracks, primary vertices and states. The memory manager frees memory accordingly and merges neighboring free memory segments, that are reused in subsequent algorithms.

Data transmissions between CPU memory and GPU memory are kept to a minimum. Raw event data is transmitted in the Initialize stage of Figure 8.3, and selections are retrieved at the end of the sequence. Data buffers that are required by subsequent algorithms are kept in GPU memory. Sporadic data transmissions occur to populate buffer sizes, and to enable the CPU opportunistic resource usage if enabled.

The challenging memory requirements of GPGPUs have been taken into account in all Allen algorithms, to the point differences in memory size between GPU models are not a decisive factor in the performance of Allen. Code reviews are carried out upon completion of algorithms to ensure memory utilization is kept to a minimum.

8.4 FRAMEWORK PERFORMANCE

Framework compilation times depend on the length of the sequence defined. A static analysis is performed on the specified sequence, meaning a longer sequence will lead to a more costly compilation. Figure 8.4 depicts compilation times for different sequence lengths. For each datapoint, the Allen sequence was compiled with the *DefaultSequence*[4] cut to a varying number of algorithms, up to the current size of the sequence. The command `cmake .. && make -j` was issued: it converts the cross-platform `cmake` representation into makefiles, and proceeds to compile them using all available cores in the server. The server is equipped with two *Intel Xeon CPU E5-2650 v3*, the compilation was done using cmake version 3.12.1, `nvcc v10.1` and `gcc 8.2.0`, and the entire compilation process was done in an SSD. The compilation times take between 1 and 3 minutes in this setup.

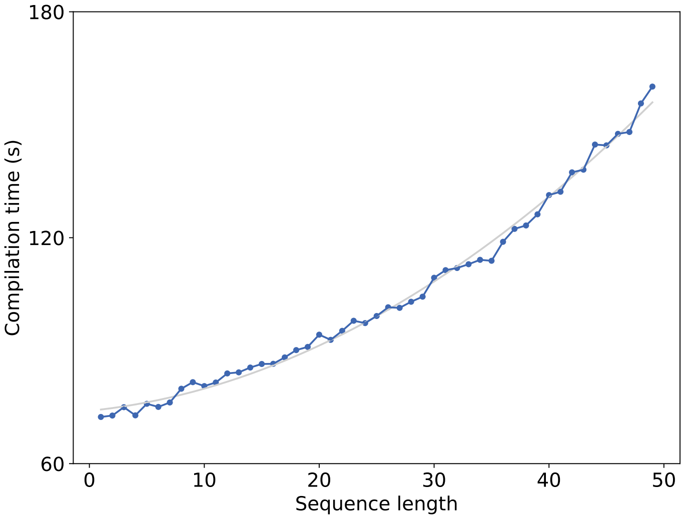


Figure 8.4: Compilation times of Allen for different sequence sizes.

A fit reveals the quadratic behavior of the compilation times. The current sequence includes only two Composite Handlers. Other single Handlers could be combined into Composite Handlers if the compilation time becomes unmanageable. The number of lines of code in Allen also affects the compilation time. Table 8.1 shows the composition of the current Allen codebase. A total of 47 thousand lines of code with 57% written in CUDA compose a fully functional HLT1 Allen codebase. It is noted

[4] The *DefaultSequence* contains a full LHCb HLT1 sequence.

that this number does not include the binary dumpers written in Gaudi to produce Allen input, nor the generation process of magnetic field input and other external tools used indirectly in Allen.

Language	Files	Lines of code	Percentage of total (%)
CUDA	405	27402	57.2
C++	66	11178	23.4
C/C++ Header	91	4807	10.0
Python	26	2419	5.1
CMake	37	1260	2.6
Total	612	47868	100

Table 8.1: Lines of code of Allen.

Several snapshots of the GPU memory consumption of Allen are shown in Figure 8.5, as reported by the memory manager. The sequence was run with 1000 minimum bias events. The memory consumption prior to the execution of respectively the Velo tracking algorithm identified by Handler tag `velo_search_by_triplet_t`, the forward tracking `lf_composite_track_seeding_t` and the parameterized Kalman filter `kalman_velo_only_t` are shown. Gray color identifies unused memory segments, whereas colored bands indicate memory segments actively occupied. All occupied segments have a description tag, but only those that are big enough are shown, to avoid clutter. The maximum size of the buffer is 500 MiB, indicating the sequence was run with that buffer size limitation. A dashed line is shown in each bar, indicating the maximum space required by the sequence so far. Note the Kalman filter algorithm is executed after the forward tracking, so even if memory segments `dev_scifi_hits` and `dev_scifi_lf_tracks` are liberated, the maximum amount of memory required so far is maintained in the memory manager, for debugging and information purposes.

As the sequence execution progresses, unused and active memory segments are interleaved. When new buffers are requested to be allocated, the memory manager attempts to locate an interleaved unused segment. If no interleaved buffers of enough size are located, the heap size indicated by the dashed line increases. In order to use memory more efficiently, memory defragmentation could be applied to join interleaved unused memory locations at the cost of moving memory buffers.

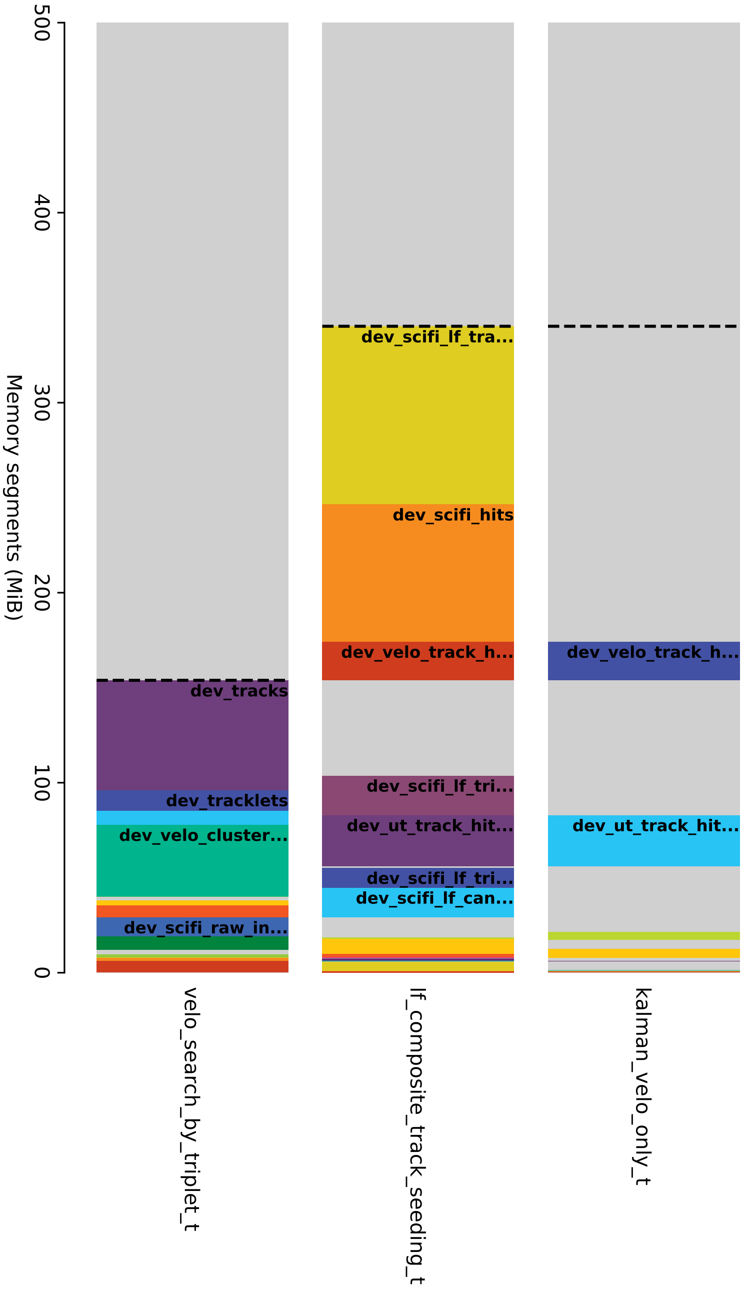


Figure 8.5: GPU memory utilization after several algorithms of the Allen sequence.

Memory transmissions from CPU memory to GPU memory and viceversa are depicted in Figure 8.6. Four run configurations are shown: *(No) CPU offload* refers to the `--cpu-offload` toggle for using opportunistic CPU resources, whereas *validation* refers to the `--validate` toggle, used for checking the produced data against Monte Carlo truth at the end of the sequence run. Two types of transmissions are depicted, *host to device* and *device to host*, where host refers to the CPU, and device to the GPU. For each setting, the transmission time is compared with the sequence execution time. The throughput of each configuration is also shown. The overhead of the output report that should be generated in a production environment has not been included, since the output format is still not decided at the time of writing (see Figure 8.3). The output report is estimated to moderately increase the device to host transmission.

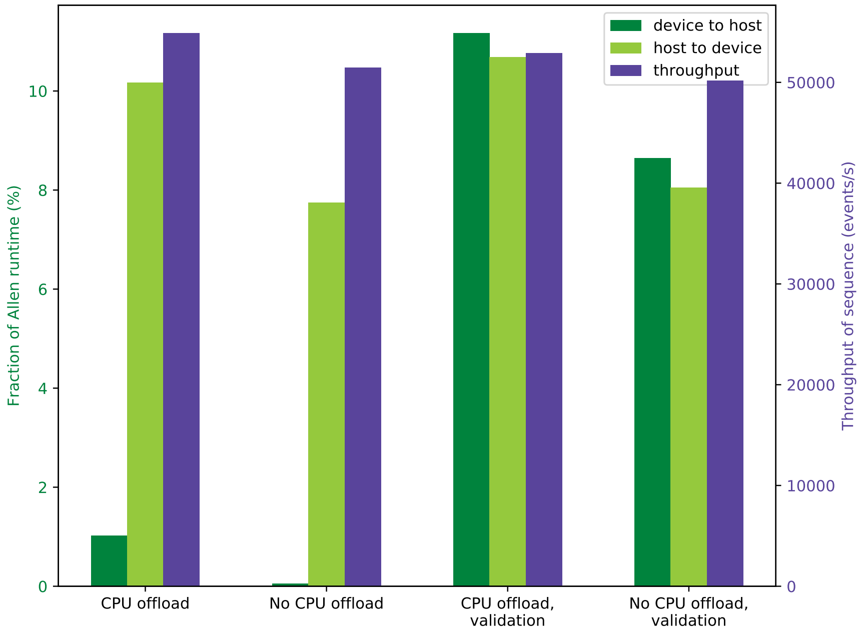


Figure 8.6: Left Y axis: Relative data transmission times with respect to sequence runtime. Right Y axis: Throughput of application.

The configuration that requires the fewest memory transmissions is *No CPU offload*, which populates raw buffers and all further computation occurs in the GPU, with sporadic buffer size transmissions back to main memory. *CPU offload* processes part of the sequence on CPU, which requires additional memory transmissions both ways in order to offload the computation. Validation sequences require transmitting to the host all data involved in the verification of the results. All data transmissions analyzed execute in under 12% of the sequence run time.

Allen runs the various worker threads asynchronously, which allows an effective pipeline of data transmissions and reconstruction execution. In spite of this, the differences in memory transmissions do impact the reported throughput of the application. The sequences tagged as *validation* were found to be 3% slower than their no validation counterparts, due to the device to host transmission required.

All processes in Allen that are not the execution of the reconstruction algorithms constitute the framework *backend*. The Allen framework backend resource consumption is of interest since it could negatively impact performance, and require more resources out of the hosting server, which would be included in a full system optimization (see section 10.3).

The resource consumption of the Allen framework backend was found to be negligible. An overview of CPU and memory usage during the execution of Allen is shown in Figure 8.7. CPU stabilizes at the equivalent of a single core 13% and 20% for the No CPU offload and CPU offload configurations respectively.

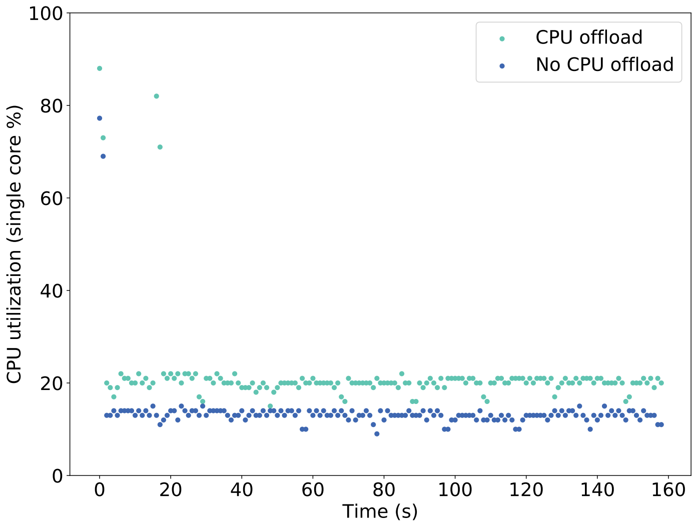


Figure 8.7: CPU resource utilization during Allen run.

A CPU heatmap of the processes that consume most resources in Allen is shown in Figure 8.8. The heatmap was generated using the tool valgrind, with the option `--tool=callgrind`, and the visualization was provided by kcachegrind. The *CPU offload* configuration was run, with 8 worker threads, 1000 events and 100 repetitions. The No CPU offload configuration did not yield any meaningful results and is not shown here.

The main process labeled as <*cycle 2*> has five children. The two processes on the top are unidentified. The CPU version of the global event cut takes 14.37% of the Allen CPU resources, whereas the various invocations of prefix sum accont to 72.64%. 1.03% of the time is spent waiting on CUDA events.

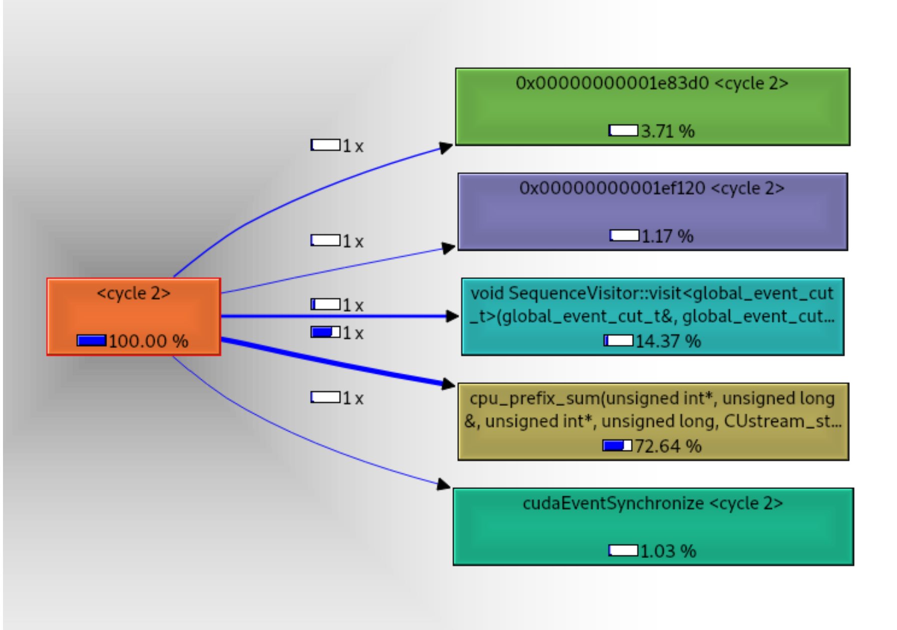

0x00000000001e83d0 <cycle 2>
3.71 %
0x00000000001ef120 <cycle 2>
1.17 %
void SequenceVisitor::visit<global_event_cut_t>(global_event_cut_t&, global_event_cut...
14.37 %
cpu_prefix_sum(unsigned int*, unsigned long &, unsigned int*, unsigned long, CUstream_st...
72.64 %
cudaEventSynchronize <cycle 2>
1.03 %
<cycle 2>
100.00 %
1 x
1 x
1 x
1 x
1 x

Figure 8.8: CPU heatmap of most resource consuming processes in Allen.

Finally, the memory consumption of Allen is shown in Figure 8.9. The valgrind application was run with the option `--tool=massif`, and the visualization is provided by `massif-visualizer`. The CPU offload configuration was used with the same settings as for callgrind.

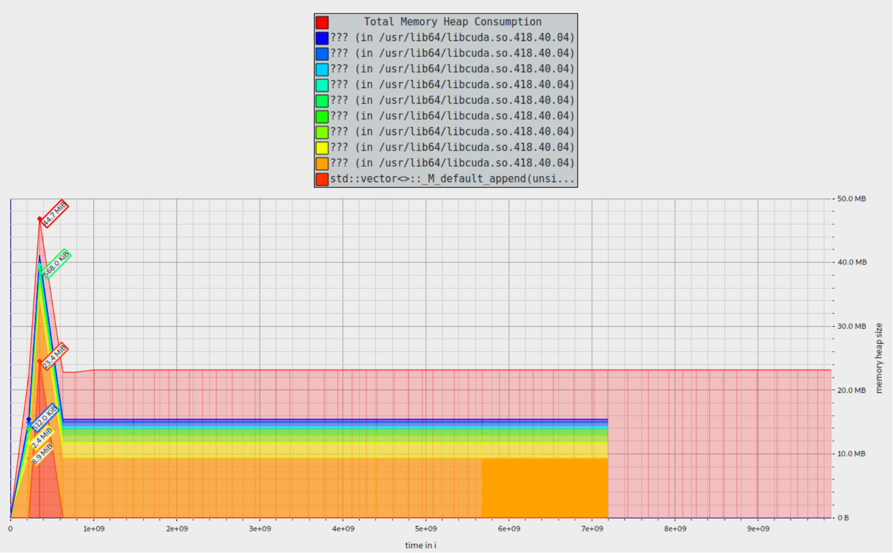


Figure 8.9: CPU memory consumption, as measured by `valgrind --tool=massif`.

The memory consumption spikes at the beginning of the sequence due to the readout of binary files and the initialization of streams data. Once all files are loaded and all streams are initialized, the memory consumption of the Allen application stabilizes at around 23 MB for the entirety of the run. The heap consumption during the sequence is composed mostly of CUDA-pinned host buffer memory locations. Such buffers are required for asynchronous data transmissions.

8.5 CONTINUOUS INTEGRATION

A set of scripts and a pipeline have been developed in collaboration with R. Schwemmer and P. Fernández Declara for use within the Gitlab repository that hosts Allen. This continuous integration work allows to:

- Check compilation works across various sequences, compilers and platforms.
- Apply a `clang-format` to all merged code to master, by invoking a common Gaudi script to that end.
- Run specific Allen sequences nightly, and publish performance results online.

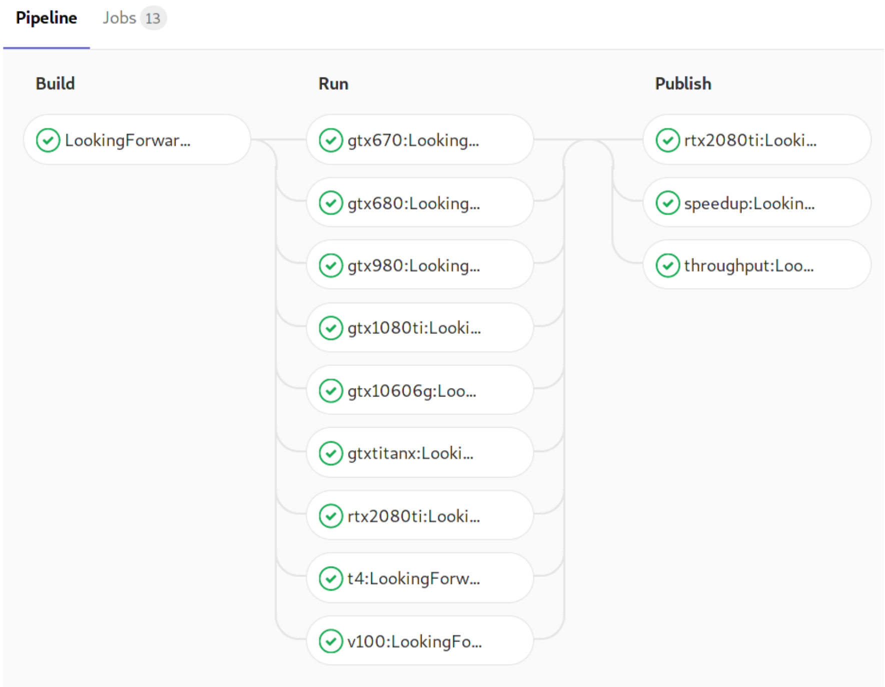


Figure 8.10: Continuous integration pipeline.

The pipeline developed is shown in Figure 8.10. The *Build* stage compiles a single binary per sequence, with all required GPU architectures as targets. The binary files are distributed, executed and profiled in the *Run* stage across different GPUs. Finally, the *Publish* stage publishes results online regarding speedup and throughput across GPUs, and a dissect of the times of the algorithms in the Allen reconstruction.

Results are published both to a mattermost channel in text form, and to a Grafana server that keeps track of the evolution in performance of the application. The continuous integration developed make collaboration and discussion easier among the growing number of collaborators of Allen.

9 TRACKING SEQUENCE PHYSICS EFFICIENCY

The physics efficiency of the developed tracking sequence has been analyzed in detail. A built-in validator in Allen has been developed and validated against the LHCb baseline software to this end. The validator and the producer of the efficiency graphs presented in this chapter have been developed in collaboration with D. vom Bruch and M. Schiller.

The validation has been run over 1000 events of the *BsPhiPhi* dataset. The physics efficiency of each individual subdetector is studied in the following sections. For completeness, the efficiency of the UT reconstruction [83] is included, even though its tracking algorithm has not been discussed nor contributed with this thesis.

9.1 VELO RECONSTRUCTION EFFICIENCY

The Velo reconstruction has three objectives: find tracks in the Velo, use those tracks to find primary vertices (PVs), and measure the distance of closest approach of each track to each PV (*impact parameter* or IP) in order to distinguish tracks produced in PVs from tracks produced in the decays of long-lived particles.

The Velo track finding is optimized to find both tracks traveling forward from the interaction region towards the magnet, and tracks traveling backwards from the interaction region out of the detector. While only forward tracks are relevant for the follow-up to the other tracking detectors, both forward and backward Velo tracks are needed to have an unbiased and efficient PV finding.

The Velo tracking efficiency obtained with *Search by triplet* is shown as a function of track momentum, ϕ, and η in Figure 9.1, where the ghost rate is also shown as a function of the number of primary vertices in the event. The same figures of merit are shown restricted to tracks coming from beauty and charm hadron decays in Figure 9.3. Beauty and charm hadron decays are shown as they are of particular interest towards the LHCb physics program. Efficiencies for electrons coming from beauty hadron decays, which are interesting for analyses, are also shown. Finally, Figure 9.2 shows the momentum distribution of tracks. Most tracks accumulate at low momenta.

9.2 UT RECONSTRUCTION EFFICIENCY

Following the Velo reconstruction, it is necessary to extrapolate the Velo tracks through the magnetic field in order to measure their momenta. The algorithm *CompassUT* creates search windows around the track extrapolated positions in the UT layers and finds compatible hits with the tracks. An efficient search is performed with the sorted decoded datatypes described in section 3.2. The extrapolation accounts for residual effects of the magnetic field, which allows for a 15-25% momentum measurement of the track. The momentum estimate will be used in the posterior Forward tracking algorithm, to obtain a narrower search window. In addition, the UT information reduces the fake rate of the LHCb tracking system. Due to the geometry of LHCb, most fake tracks are caused by mismatching genuine Velo and SciFi track segments which however come from different particles. The requirement of a matching UT hit along the extrapolated trajectory reduces the number of these ghosts.

To help comparing track efficiencies with the Velo, analogous efficiency figures are shown for Velo-UT tracks. The Velo-UT tracking efficiency is shown as a function of track momentum, ϕ, and η in Figure 9.4, with the ghost rate as a function of the number of primary vertices. Beauty and charm hadron decays efficiencies are shown in Figure 9.6, including efficiencies for electrons. CompassUT can be tuned to obtain better efficiencies, trading off algorithm speed. The chosen default configuration maximizes reconstruction efficiency within 1% of the best achievable efficiency tested [83].

Electron efficiencies are significantly worse than non-electron efficiencies. This is expected, as electrons are typically more difficult to detect: they may emit photons in an effect known as the *Bremsstrahlung*, losing energy in the process, making a precise parametrization difficult; and the Bremsstrahlung may also change the electron's trajectory.

The momentum distribution is shown in Figure 9.5. The *turn-on* curve refers to the gradual increase in efficiency (the steeper the better) as a function of momentum observed in the low momentum region. The LHCb physics program puts more emphasis on heavier particle decays, such as beauty and charm hadron decays, which produce typically high-momentum particles which are easier to reconstruct as they bend less and interact less with detector material. The observed turn-on curve and overall efficiencies are acceptable, and a better efficiency for the low momentum region and its impact on speed will be studied in the future.

9.3 FORWARD TRACKING EFFICIENCY

Finally, the Forward tracking algorithm *Looking Forward* completes the tracking sequence of Allen. The most critical Forward tracking algorithm parameter is the p_T threshold above which tracks are sought. Two such thresholds are used. The first is a cut applied on the momentum and p_T of the reconstructed Velo-UT tracks, to save time attempting to reconstruct tracks which would anyway fall below the forward tracking cutoff. The second, somewhat tighter, threshold uses the charge and p_T of the reconstructed Velo-UT track to calculate the size of the SciFi search window in the x (bending) plane.

The momentum distribution for different Long track types is shown in Figure 9.8. The Forward tracking efficiency and ghost rate are shown in Figure 9.7. The tracking efficiency for decay products of beauty and charm hadrons is shown in Figure 9.9, including electron efficiencies. The Forward algorithm uses an ANN classifier to reject ghosts before any Kalman filter is performed, similarly to the strategy followed in the Run 2 reconstruction of LHCb [84, 85].

The physics efficiencies of the tracking reconstruction of Allen, composed of algorithms *Search by triplet*, *CompassUT* and *Looking Forward*, meet the requirements of the HLT1 physics program of LHCb for the Upgrade, documented in [14].

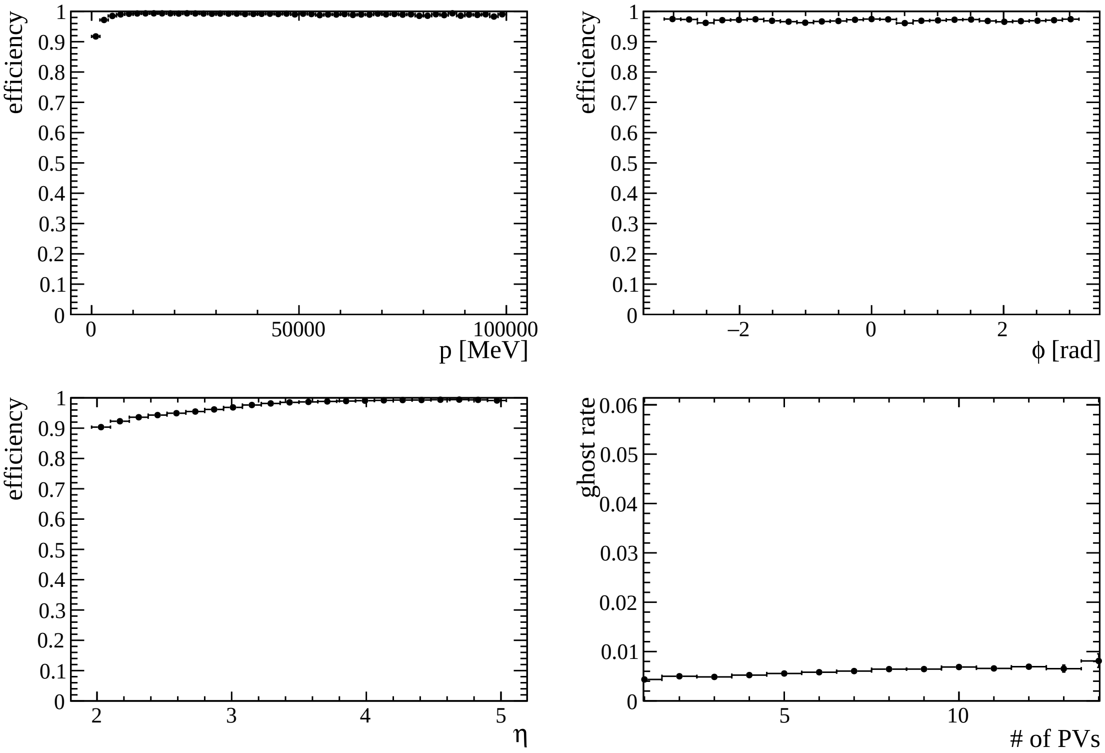


Figure 9.1: The Velo tracking efficiency for all Velo tracks in the event. It is shown as a function of track (top left) momentum (top right) ϕ and (bottom left) η. On the bottom right the ghost rate is shown as a function of the number of primary vertices in the event.

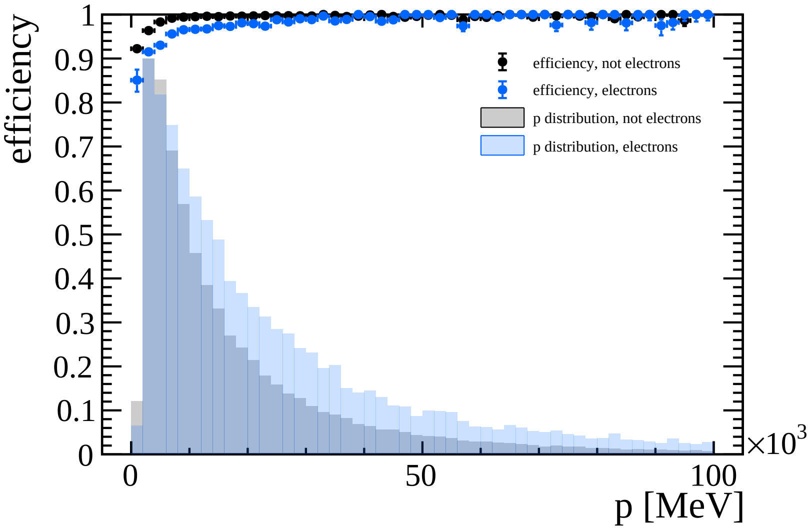


Figure 9.2: Velo tracking efficiency and momentum distribution of electron and non-electron long tracks coming from decays of beauty hadrons.

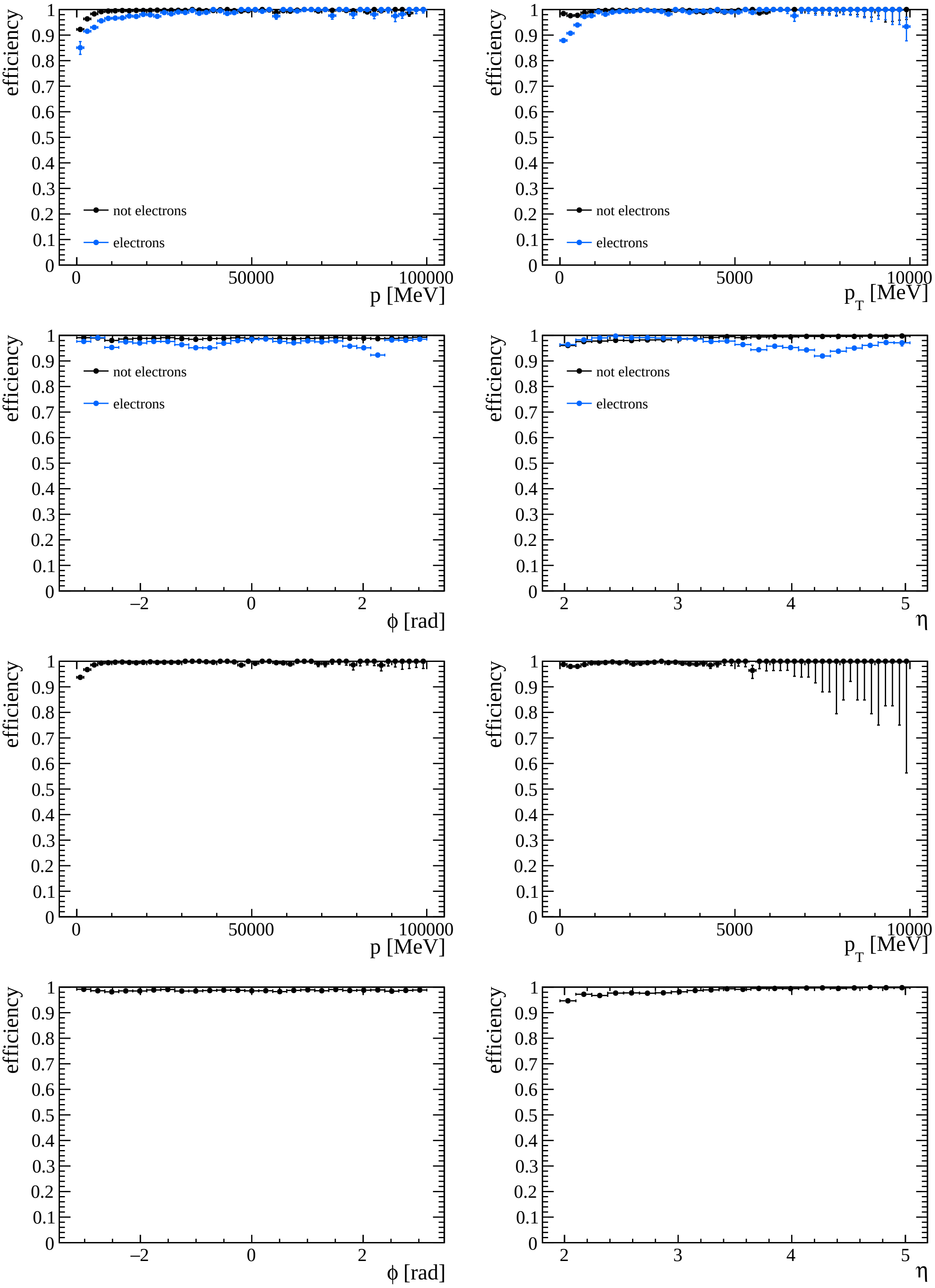


Figure 9.3: The Velo tracking efficiency for long tracks coming from the decays of (top quartet) beauty and (bottom quartet) charm hadrons. Within each quartet the efficiency is shown as a function of track (top left) momentum (top right) p_T (bottom left) ϕ and (bottom right) η.

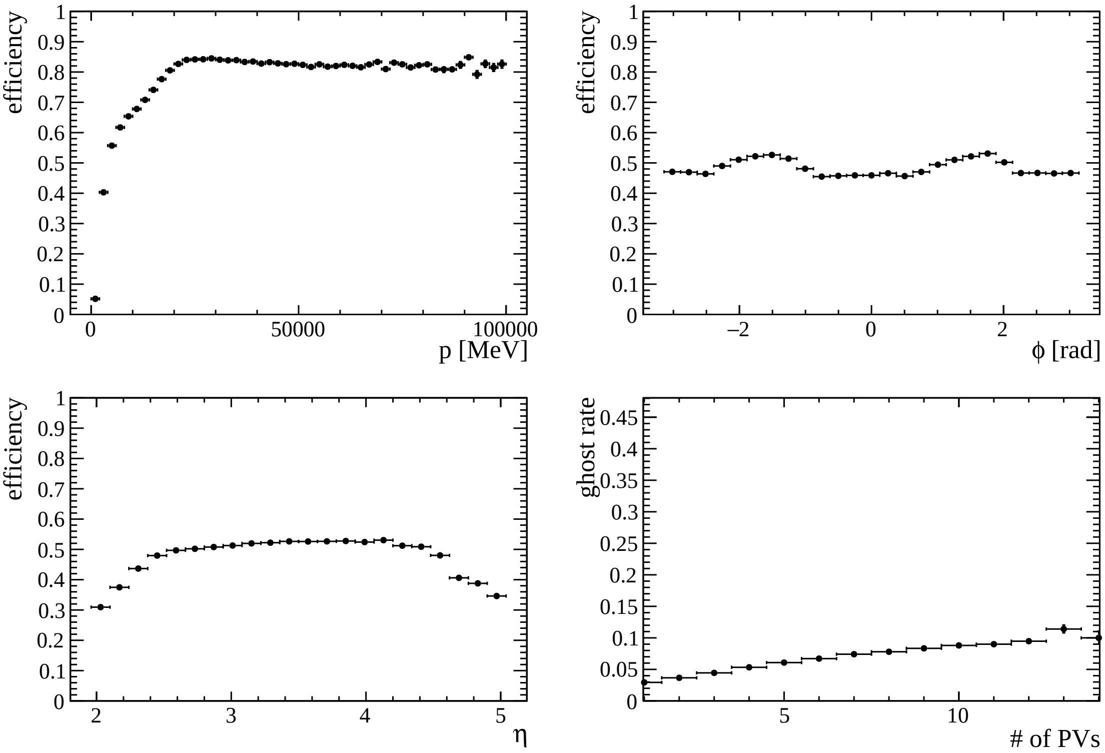


Figure 9.4: The Velo-UT tracking efficiency for all Velo-UT tracks in the event. It is shown as a function of track (top left) momentum (top right) ϕ and (bottom left) η. On the bottom right the ghost rate is shown as a function of the number of primary vertices in the event.

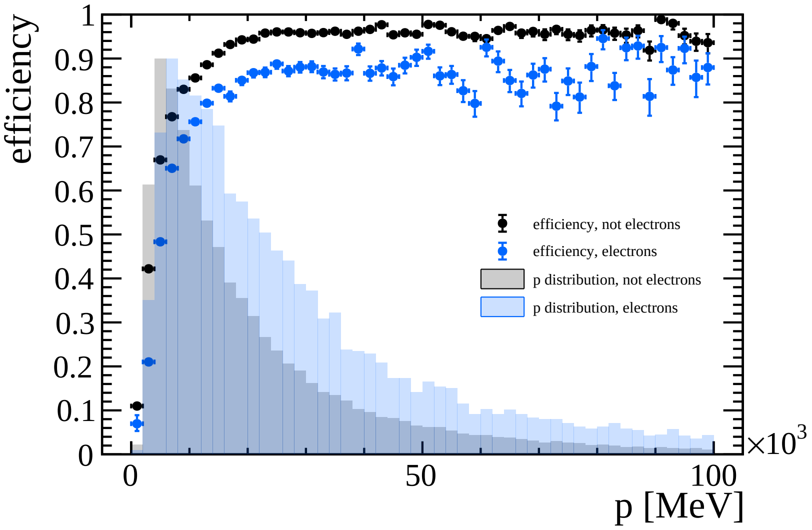


Figure 9.5: Velo-UT tracking efficiency and momentum distribution of electron and non-electron long tracks coming from decays of beauty hadrons.

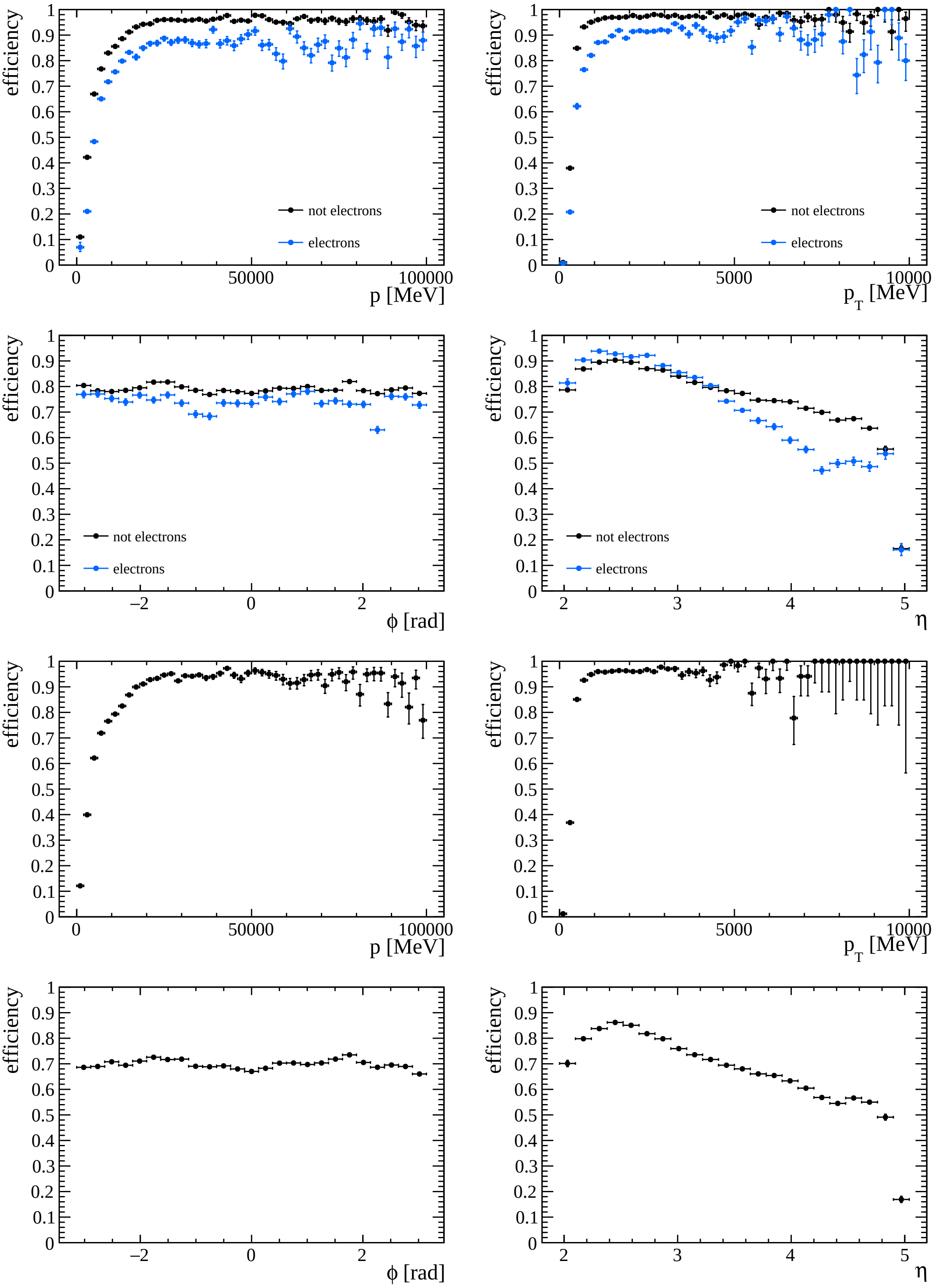


Figure 9.6: The Velo-UT tracking efficiency for long tracks coming from the decays of (top quartet) beauty and (bottom quartet) charm hadrons. Within each quartet the efficiency is shown as a function of track (top left) momentum (top right) p_T (bottom left) ϕ and (bottom right) η.

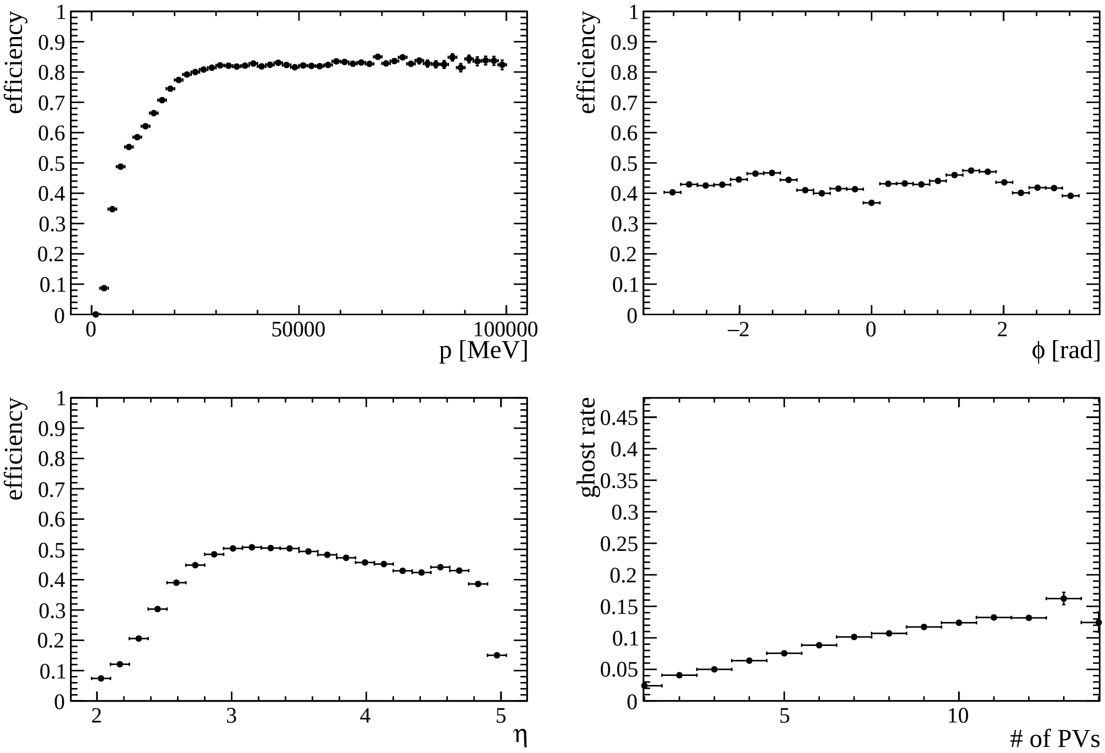


Figure 9.7: The Forward tracking efficiency for all long tracks in the event. It is shown as a function of track (top left) momentum (top right) ϕ and (bottom left) η. On the bottom right the ghost rate is shown as a function of the number of primary vertices in the event.

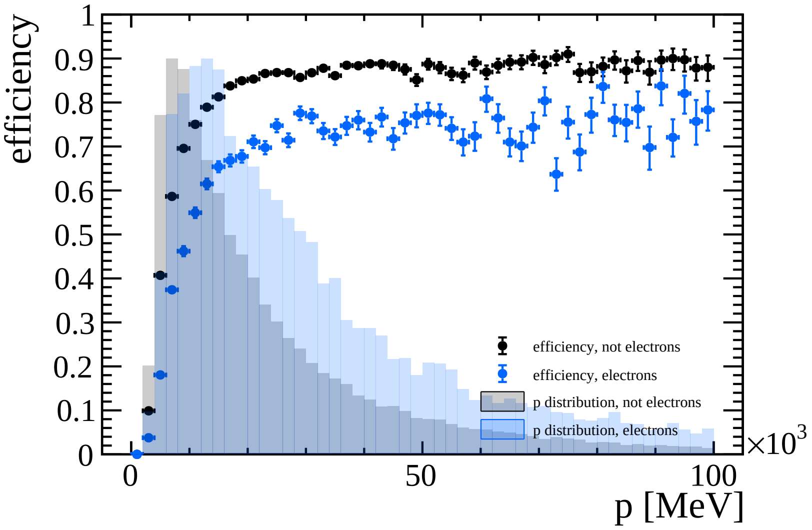


Figure 9.8: Forward tracking efficiency and momentum distribution of electron and non-electron long tracks coming from decays of beauty hadrons.

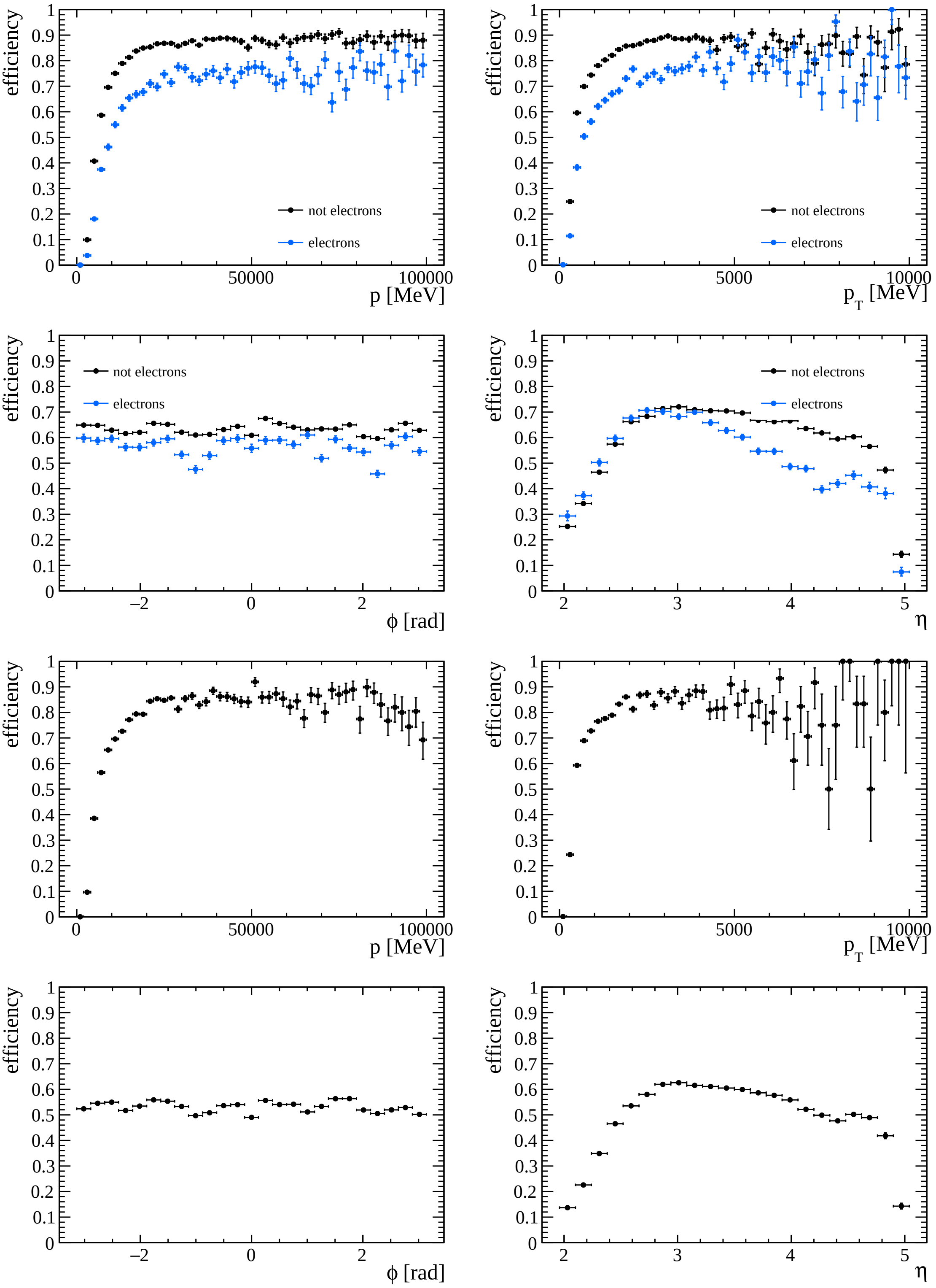


Figure 9.9: The Forward tracking efficiency for long tracks coming from the decays of (top quartet) beauty and (bottom quartet) charm hadrons. Within each quartet the efficiency is shown as a function of track (top left) momentum (top right) p_T (bottom left) ϕ and (bottom right) η.

10

PERFORMANCE ANALYSIS

Performance is a key component of the developments contributed with this thesis. For the Allen application to be used in a production environment, it should not only deliver the physics efficiency required by the LHCb physics program, but it should do so at a high performance. The figure of merit of the application is the throughput, measured in rate of events per second (Hz). The rate attainable in a single card determines the number of cards a prospective GPU HLT1 system would consist of, and plays a determining role in the feasibility of the system.

In addition, the scalability of Allen to newer hardware determines the expected gain in performance from upcoming hardware iterations. The sequence execution has been profiled. A detailed timing breakdown pinpoints the hot sections of the sequence. The performance of the Velo sequence is compared with the CPU SPMD reconstruction sequence (see chapter 5).

10.1 METHODOLOGY

The hardware used for the analysis in this chapter is listed in Table 10.1. The GPUs considered include a broad variety of graphics cards. NVIDIA produces the chips in *GeForce* cards, which are usually assembled and manufactured by graphics cards brands, targeted for the consumer market. The *Quadro* and *Tesla* cards are produced and manufactured by NVIDIA with no middle-man. Quadro is a high-end line of products targeted for professional designers and video editors. Tesla is the scientific line of cards, targeted for scientific computing use.

GeForce cards feature video output, high performance on single precision floating point operations (measured in FLOP per second or FLOPS) and active cooling. Quadro also features video output, active cooling and high performance single precision and *error-correcting code* memory (ECC). The scientific cards do not provide video output, however they support ECC

Card	# cores	Max freq.	Cache	DRAM	DRAM	CUDA
		(GHz)	(MiB, L2)	(GiB)	type	cap.
GeForce GTX 670	1344	1.06	0.5	1.95	GDDR5	3.0
GeForce GTX 680	1536	1.14	0.5	1.95	GDDR5	3.0
GeForce GTX 780 Ti	2880	0.93	1.5	2.95	GDDR5	3.5
GeForce GTX 980	2048	1.29	2	2.01	GDDR5	5.2
GeForce GTX TITAN X	3072	1.08	3	11.92	GDDR5	5.2
GeForce GTX 1060 6GB	1280	1.81	1.5	5.94	GDDR5	6.1
GeForce GTX 1080 Ti	3584	1.67	2.75	10.92	GDDR5	6.1
GeForce RTX 2080 Ti	4352	1.545	6	10.92	GDDR5	7.5
Quadro RTX 6000	4608	1.77	6	24	GDDR6	7.5
Tesla T4	2560	1.59	4	15.72	GDDR6	7.5
Tesla V100 32GB	5120	1.37	6	32	HBM2	7.0

Card	Peak TFLOPS	TDP	Release date	MSRP	Allen
	(32-bit)	(Watts)	(mm/yy)	($)	settings
GeForce GTX 670	2.63	170	05/12	400	Low
GeForce GTX 680	3.25	195	03/12	500	Low
GeForce GTX 780 Ti	5.34	250	12/13	700	Low
GeForce GTX 980	4.98	250	09/14	549	Low
GeForce GTX TITAN X	6.69	250	03/13	999	High
GeForce GTX 1060 6GB	4.38	120	07/16	249	Low
GeForce GTX 1080 Ti	11.34	180	03/17	699	High
GeForce RTX 2080 Ti	13.34	250/260	09/18	1199	High
Quadro RTX 6000	16.31	250	08/18	4000	High
Tesla T4	8.141	70	09/18	2295	High
Tesla V100 32GB	14.13	250	03/18	8999	High

Allen settings	Threads	Memory	Number of events	Repetitions	Validation
	(-t)	(-m)	(-n)	(-r)	(-c)
High	12	700	1000	100	0 (off)
Low	2	700	1000	100	0 (off)

Table 10.1: (Top) GPUs used for benchmarking. (Bottom) Benchmarking options set.

memory, high performance single and double precision, and passive cooling.

The Allen application is a full realization of the HLT1 sequence that only requires single precision arithmetic. This was an early design decision, knowing the various limitations of the graphics cards in the market, in an attempt to allow as many hardware options as possible.

ECC memory can protect against bit-errors which might affect long-running calculations or corruption. Since the Allen application reconstructs events which are physically independent one another, random bit-errors do not carry over other events. Furthermore, they are likely indiscernible from noise. No effect was observed by turning on ECC memory in the Allen application, however turning off ECC memory resulted in 10-20% performance increase.

The reliability and effectiveness of passive and active cooling should be studied over a period of time, in particular the reliability of active cooling, in terms such as mean-time-to-failures (MTTF). This study however is beyond the scope of this thesis. A commissioning of the system will be carried out in the upcoming months to determine the feasibility of Allen in a production environment.

The Allen tag `v0.6` has been used for all tests shown in this chapter. The application has been compiled with compilers `gcc 8.2` and `nvcc 10.1.168` using the flags listed in Table 10.2. *fast math* is employed. The loss in precision caused by using single precision and fast math throughout the sequence was found not to cause physics efficiency loss. The tracking algorithms discussed in this thesis require simple arithmetic operations. The nature of tracking (1) creating seeds of hits, (2) extending forming tracks with new hits, requires a set precision at every step, but the intermediate results consist of detector measurements, so no carry over of inefficiency occurs. The efficiency attainable by single precision and fast math is at least enough to cover the HLT1 use case (see section 9).

gcc compilation flags	nvcc compilation flags
`-march=ivybridge`	`--generate-line-info`
`-O3 -g -DNDEBUG`	`-O3 -g -DNDEBUG`
	`--ftemplate-depth=300`
	`--use_fast_math`
	`--expt-relaxed-constexpr`

Table 10.2: gcc and nvcc compilation flags used for Allen.

Throughput measurements are performed on dedicated machines with no other running user processes with the following methodology:

- Unless explicitly stated, the sequence under study is the full LHCb HLT1 sequence, described in section 8.2.
- The `CPU offload` configuration is used, whereby the global event cut and all prefix sum algorithms are run on CPU (cref. section 8.4).
- Experiments are run with the *minbias* dataset, which contains 1000 events.

- The sequence of algorithms is run on 2 or 12 concurrent threads (see table 10.1).
- A configurable number of events is run in parallel for each thread, typically around 1000. When this number is greater than the number of available minbias events, a round robin over the existing ones is performed.
- A number of repetitions inside each thread is run, typically around 100, to make the measurement less sensitive to fluctuations.
- The total number of events processed in parallel is given by the number of streams multiplied by the number of events multiplied by the number of repetitions.
- The implemented timing service is a wrapper using `std::chrono`. A single Timer is started prior to processing sequences in every thread, and it is stopped after all threads have returned (joined).

10.2 HLT1 SEQUENCE PERFORMANCE ANALYSIS

The performance of the full HLT1 sequence in Allen is shown in Figure 10.1. The cards GeForce RTX 2080 Ti, Quadro RTX 6000 and Tesla V100 32GB deliver a performance of roughly 60 kHz in a single card (57.93, 59.87 and 58.74 kHz respectively). The 30 MHz rate of inelastic particle collisions is therefore realizable with a system comprising 500 cards, irrespective of the product line. At the time of writing, these cards represent the top models in the consumer, specialist and scientific computing market, respectively.

A system with 1000 Tesla T4 cards would also meet the requirements, with less power draw than the other three configurations. The Tesla T4 model is the only one of the top four that does not require external power draw, and is fed exclusively through the PCI-express slot housing it. Older consumer models deliver a decreasing performance, and even though they are not considered realistically for an integrated GPU HLT1, they are however interesting to study the evolution of the technology.

Figure 10.2 shows on the top left the rate of each card versus its theoretical 32-bit peak TFLOPS. The performance of the application seems to linearly scale as a function of the peak TFLOPS of the cards under consideration. The GeForce GTX 1080 Ti is

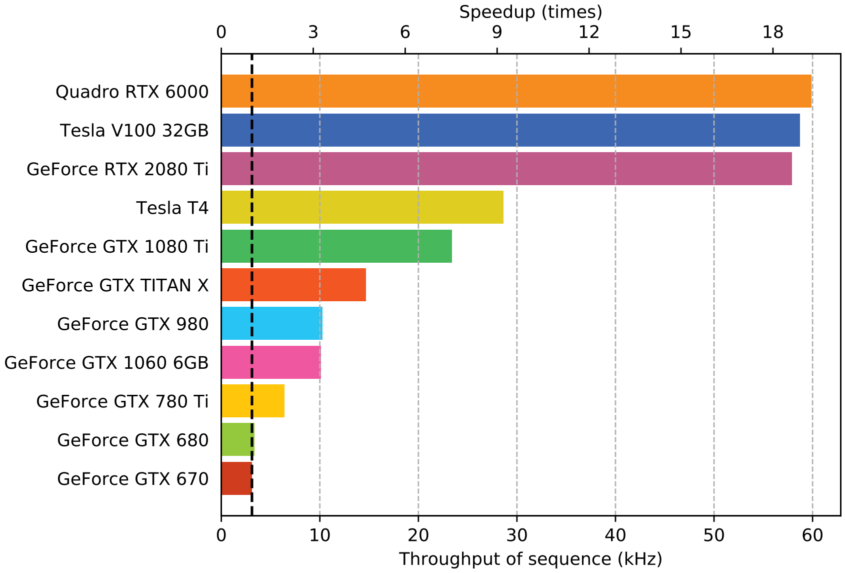


Figure 10.1: Performance of full HLT1 sequence across GPU architectures.

deviated from this tendency. The performance differential is attributed to a combined effect of the increase in CUDA cores, the increase in L2 cache (from 2.75 MiB to 4 MiB in the Tesla T4, and 6 MiB in the top three models), and the availability of tensor cores in later GPU generations, which is used for the hottest algorithm (cref. Section 6.2).

The rate as a function of release date is shown in the top right figure. The correlation between release date and performance is not clear, due to the existence of several product lines, such as the low profile Tesla T4, on the same release date as the top three cards. The GeForce GTX TITAN X delivers 14.68 kHz, well above other GeForce cards nearing its release date. Compared to the GeForce GTX 780 Ti, its core count is 6% higher, it has double the amount of L2 cache and its peak frequency is 16% higher. Unfortunately, no other TITAN card iterations were available for testing.

The *bang per buck*, or price performance of the cards under consideration are plotted against their release date on the bottom left of Figure 10.2. The bang per buck is calculated as the rate divided by the manufacturer's suggested retail price *MSRP* at release of each card. The price performance of all GeForce cards have been fitted linearly, and the data suggests a linear tendency on all tested consumer hardware. The Quadro and Tesla lineup

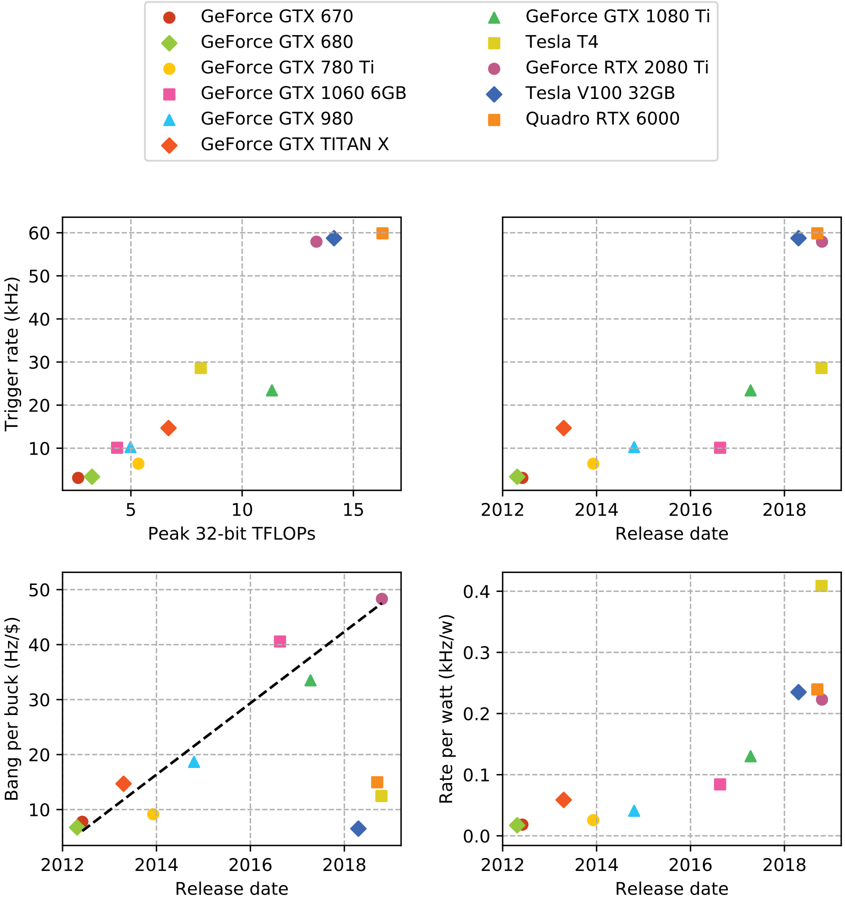


Figure 10.2: Trigger rate versus peak TFLOPS, cost and power envelope across graphics cards.

appear as outliers in this graph as one would expect, since they are marketed and targeted to a different demographic.

The rate per watt of each card as a function of release date is shown on the bottom right figure. Two models of different brands of the GeForce RTX 2080 Ti with a different TDP of 250 and 260 watts were considered. The numbers shown here refer to the 260 watt-model. Irrespective of this detail, the cards show an increase in power efficiency with time. Due to its low TDP, the Tesla T4 tops the chart with higher efficiency per watt than all other cards.

The timing fractions of all algorithms composing the full Allen HLT1 sequence are shown in Figure 10.3. Color indicates which subsequence the algorithm belongs to, and all subsequence timings are aggregated in the Figure legend. The timings were obtained with the `nvprof` tool running the sequence in a

GeForce RTX 2080 Ti. It should be noted the timings shown here indicate the aggregated time each respective algorithm took to finish execution. However, there is not always a correlation with the resource consumption of the algorithm. That is, no information is presented concerning how many *Streaming Multiprocessors*, registers, or cache were required by the algorithm in the presented figure.

The SciFi reconstruction dominates the sequence (55.83% of the time), followed by the Velo reconstruction (23.23%). The PV finder, UT reconstruction and muon subsequences roughly contribute 10% of the time to the sequence each. Finally, a parameterized and simplified implementation of the Kalman filter, and the selection algorithm `run_hlt1` complete the timing subsequences with less than 2% each.

The top consumers in the chart are the *Looking Forward triplet seeding* and the *Search by triplet* algorithms. The tensor core implementation of the Looking Forward triplet seeding was used. In spite of the resource consumption remark stated above, the graph shows an indication of where the biggest gains in performance may be obtained by optimizing specific algorithms, from an Amdahl's law standpoint [25]. An iterative optimization process has been followed for the top time consumers during the elaboration of this thesis, testing different algorithm approaches, arithmetic formulations, parameter configurations and GPU kernel configurations, choosing the best performing ones.

10.2.1 *Parameter scans*

The Allen application can be invoked with various flags. The flags that impact the performance of the application are listed in table 10.1. Out of these:

- `-c` – Setting the validation to *off* ensures no additional buffers are requested and fewer data transmissions from device to host throughout the execution of Allen.
- `-m` – The memory setting has no impact on performance other than enabling the creation of additional threads. The sole requirement is that sufficient memory buffer must be allocated to prevent allocation exceptions.
- `-r` – A higher number of repetitions increases the robustness of time measurements.

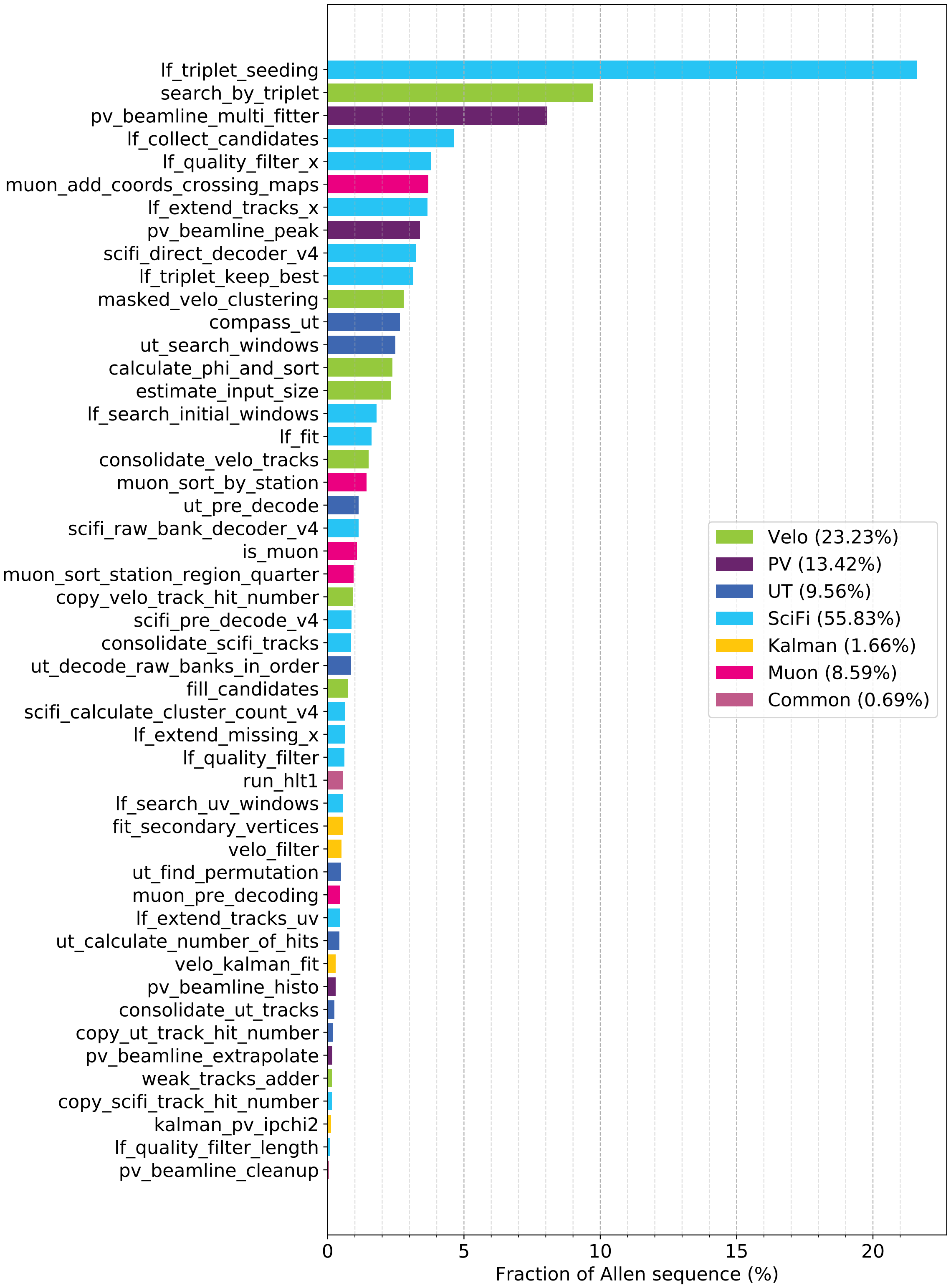


Figure 10.3: Timing fractions of all algorithms executed on the GPU composing the HLT1 sequence.

Therefore, the settings *number of events* (-n) and *number of threads* (-t) are left to tweak the performance of the application. It should be noted that it is not possible however to increase any of these parameters ad infinitum, since the memory required by the card increases linearly with both the number of events and the number of threads. A parameter scan of the two settings is shown in Figure 10.4.

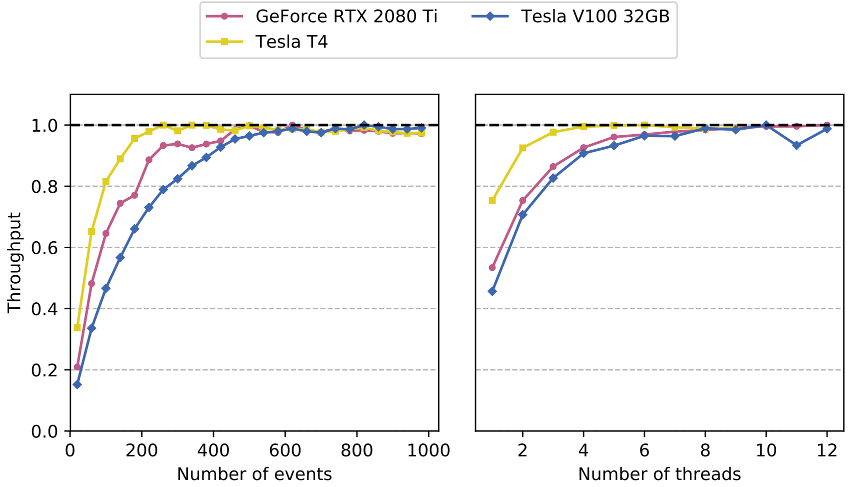


Figure 10.4: Scan of parameters number of events and number of threads for various graphics cards.

The peak throughput is normalized for every card shown in the figures. For the performance scans shown in the figures, validation was set to off (0), the memory buffer was set to 700 MiB per thread, and the number of repetitions was configured at 1000. On the left, the normalized throughput is shown as a function of the number of events. For this Figure, the number of threads was set to 12. A configuration of 600 or more events yields a peak performance on all cards shown. The Tesla T4 peaks with 240 events. On the right, the normalized throughput as a function of the number of threads is shown. The number of events was set to 1000 in this case. 98% of the peak performance is obtained with a configuration with 8 threads on the consumer and high-end scientific cards, whereas the Tesla T4 peaks early with 4 threads.

The performance of a configuration of 8 threads and 600 events has been tested: 96% of the peak performance is reached, using 203 MiBs per thread, amounting to 2097 MiB memory utilization in total, as reported by the Allen memory manager.

10.2.2 *Velo sequence performance analysis*

The Velo sequence has been developed both in CUDA and SPMD (cref. chapter 5), allowing us to compare performance with CPUs. The CPUs under consideration are depicted in table 10.3. All CPUs tested are server CPUs in dual-socket configurations. Three Intel Xeon CPUs are considered, including two Broadwell processors and a Skylake processor Intel Xeon Silver 4114. Broadwell processors feature 256-bit YMM registers and support up to the AVX2 vector-instruction set. The Skylake processor features 512-bit ZMM registers and supports a subset of AVX512 specifications [1]. An AMD processor has also been considered, the AMD Ryzen Threadripper 2990WX.

Processor	# cores	Max freq. (GHz)	Cache (L3, MiB)	SIMD cap.	MSRP ($)
Intel Xeon Broadwell E5-2650	20	3.0	25	AVX2	1166
Intel Xeon Broadwell E5-2687W	20	3.5	25	AVX2	2141
Intel Xeon Silver 4114	16	2.7	13.75	AVX512	694
AMD Ryzen Threadripper 2990WX	64	4.2	64	AVX2	2000

Processor	Vector extension	Mask size	Gang size
Intel Xeon Broadwell E5-2650	AVX2	32	8
Intel Xeon Broadwell E5-2687W	AVX2	32	8
Intel Xeon Silver 4114	AVX512	32	16
AMD Ryzen Threadripper 2990WX	AVX2	32	8

Table 10.3: CPU processors and configurations under test.

The number of cores of each processor refers to the total number of logical cores. In the case of Intel processors, the processors achieve this number by having half the number of physical cores and using the *HyperThreading* technology. The AMD processor consists of 64 physical cores.

When a SPMD program is compiled with the Intel SPMD Program Compiler (ISPC) [67], the targeting vector width and mask

[1] Concretely, it supports the extensions `avx512f avx512dq avx512cd avx512bw avx512vl`.

size must be specified. The vector width must be supported by the targeted processor, and the supported options cover a range of vector technologies, ranging from SSE2 to AVX512. The ARM NEON vector units are also supported. It should be noted that all `x86-64` processors (commonly referred to as Intel x86 64-bit processors) support at least the set of vector extensions SSE2. Therefore, a program written in SPMD is compilable for any Intel x86 64-bit processor.

The manner in which SPMD executes operations is similar to how NVIDIA GPUs execute code using *warps*. A *gang* of processor elements executes each operation in a masked manner, and the mask determines whether the operation is *written back* or ignored. The gang size and mask size can be configured at compile time, and several options are available for each vector width. The configurations chosen for the processors under analysis are shown in the bottom table of 10.3.

In contrast with GPUs, ISPC guarantees a common *Program Counter* be kept throughout the execution of the program by all gang elements. This removes the need for control synchronizations (such as CUDA `__syncthreads()` or OpenCL `barrier()`). ISPC also provides a threading model, however the model available since C++11 with `std::thread` is instead used.

The Velo reconstruction SPMD implementation is compared with the CUDA implementation in Allen in Figure 10.5. The dual-socket CPU processors are outperformed by all graphics cards with the exception of the GeForce 670 and 680. No effort was put in optimizing the SPMD translation in the comparison shown in this Figure. An effective and fully functional SPMD code was produced from the CUDA implementation with a low effort translation in an estimated time of less than a week.

The Skylake processor outperforms the Broadwell 2650 processor, in spite of its lower core count, frequency and cache. This is due to the wider vector unit and the consequent availability of twice as many vector registers of any width. Out of the Intel processors, the Broadwell 2687W outperforms the others, due to its higher frequency. The AMD Ryzen Threadripper 2990WX leads the CPU competition in the chart, with a 1.27× over the Broadwell 2687W. A speedup of 6.93× between the Tesla V100 and the AMD Ryzen Threadripper 2990WX is observed.

Some problems are more amenable to be solved sequentially than in parallel. CPU processors, with higher frequency clocks,

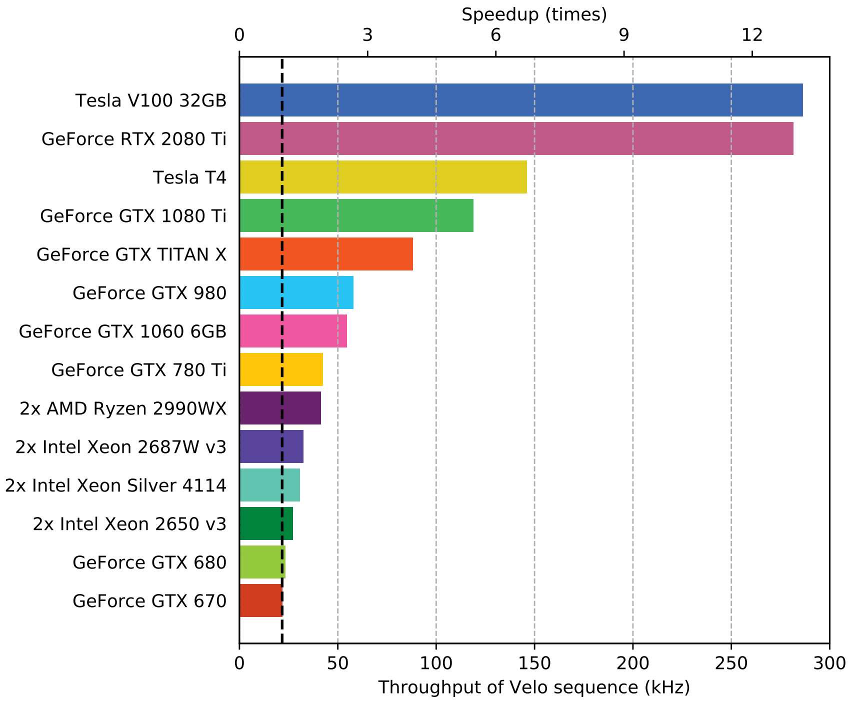


Figure 10.5: Performance of Velo sequence across architectures.

bigger caches and better branch prediction, provide flexibility upon solving problems not found on many-core processors. The case of the Velo clustering problem is a clear example where compromises were made to accomodate the algorithm to the limited amount of memory of GPUs. The flexibility of CPUs permits a sequential algorithm to be more efficient than the data-parallel many-core algorithm in CPU hardware (cref. section 3.1).

On the other hand, it was found that whenever an algorithm mapped efficiently to a many-core architecture exploiting its features rather than being limited by them, a translation to SPMD delivered a good performance as well, which could only be topped with architecture-oriented tweaking. A performance speedup of the SPMD translation with respect to the latest publicly available result of the LHCb baseline has been shown [1, 86]. Since then, further optimizations have been done to the LHCb baseline algorithms, and a detailed analysis will be carried out in the future.

The CPU Velo clustering could be sped up by implementing the sequential algorithm instead of the SPMD mask clustering. A hybrid program has been developed, consisting of a sequential clustering that prepares data as Structure of Arrays, followed by the SPMD tracking algorithm. Figure 10.6 shows a comparison between the hybrid CPU programs and the GPU Velo reconstruction.

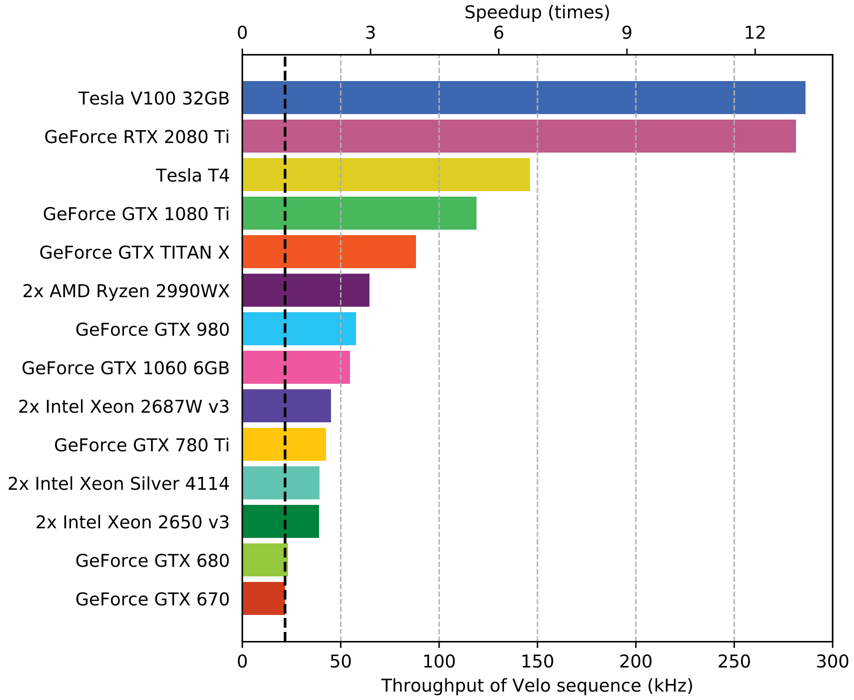


Figure 10.6: Performance of Velo sequence across architectures. CPU performance obtained with *hybrid* program.

A speedup of around 1.5× is observed across the CPUs under consideration with respect to the SPMD-only program. The performance of CPUs falls behind that from GPUs. A performance of 64.56 kHz has been obtained on the better performing CPU AMD Ryzen Threadripper 2990WX, a factor 4.43× behind the Tesla V100.

SPMD has allowed us to perform low-effort translations to CPU architectures, while preserving the SIMD data-parallel design of the application, with minimal intervention required on the algorithm code. An SPMD version of the Allen framework is under development, and an LLVM [87] plug-in to effectively automate algorithms translations. Although other multi-target middleware languages exist [80, 88], SPMD uses vector-units delivering a scalable program to future CPU architectures.

10.3 INTEGRATION IN DATA ACQUISITION SYSTEM

It has been demonstrated that the GPU HLT1 application meets the physics requirements at a high performance of up to 60 kHz in a single card. In order for the developed application to make it into a production environment, it will be necessary to integrate the application in the LHCb infrastructure, optimize the system and validate its capability to sustain long runs of the experiment. This section points out several possible configurations and throughput requirements of a production system. It is out of the scope of this thesis to validate the requirements of a full-fledged GPU trigger system.

GPUs need servers that host and power them. The form-factor of the top performing GPU cards, GeForce RTX 2080 Ti, Quadro RTX 6000 and Tesla V100, is *PCIe Gen3*, 16x, full-height, double-slot, with a power draw of 250 W. In addition, the Tesla T4 is a PCIe Gen3, 8x, half-height, single-slot card, with a power draw of 70 W. The bandwidth delivered by PCIe Gen3 16x is up to 100 Gb/s full-duplex, while PCIe Gen3 8x is half the capacity, 50 Gb/s full-duplex.

GPUs require the necessary throughput for data to flow in and out of GPU memory. It should be noted that due to the nature of HLT1 applications, the data reduction is such that the bandwidth required is uneven in the two directions, where the influx is roughly a factor 30 higher than the outflux. Therefore, one limiting factor to deploy GPUs is the available (influx) bandwidth on PCIe slots. This restricts the usage of PCIe slots that share bandwidth through the use of a *PLX switch*, or that don't deliver the full 16x (8x) bandwidth. Table 10.4 depicts possible scenarios relating the GPU HLT1 performance and the influx and outflux bandwidth requirement. The figures overestimate event size to 200 kB, providing a safety margin of 33% over the peak event size of 150 kB.

The bandwidth requirement applies to all components in the path of the data delivery to the GPUs. For instance, assuming a GPU capable of processing 60 kHz of events and 2 GPUs in a server, the network must sustain a 200 Gb/s influx traffic to the server, and the server must be able to sustain such traffic, both in terms of network throughput and memory throughput.

In order to deploy servers with GPUs to perform the HLT1, the entire data acquisition system (DAQ) must be considered.

Number of servers	GPUs in each server (1 GPU = 60 kHz)	Bandwith in (Gb/s)	Bandwidth out (Gb/s)
1000	0.5	50	1.67
500	1	100	3.34
250	2	200	6.67
125	4	400	13.33
84	6	600	20

Table 10.4: Relation between number of servers, number of GPUs in each server and required throughput to satisfy the 30 MHz collision rate.

For reference, a simplified schematic of the baseline LHCb DAQ is shown in Figure 10.7 (cref. section 1.2). Only the relevant components to this discussion are included.

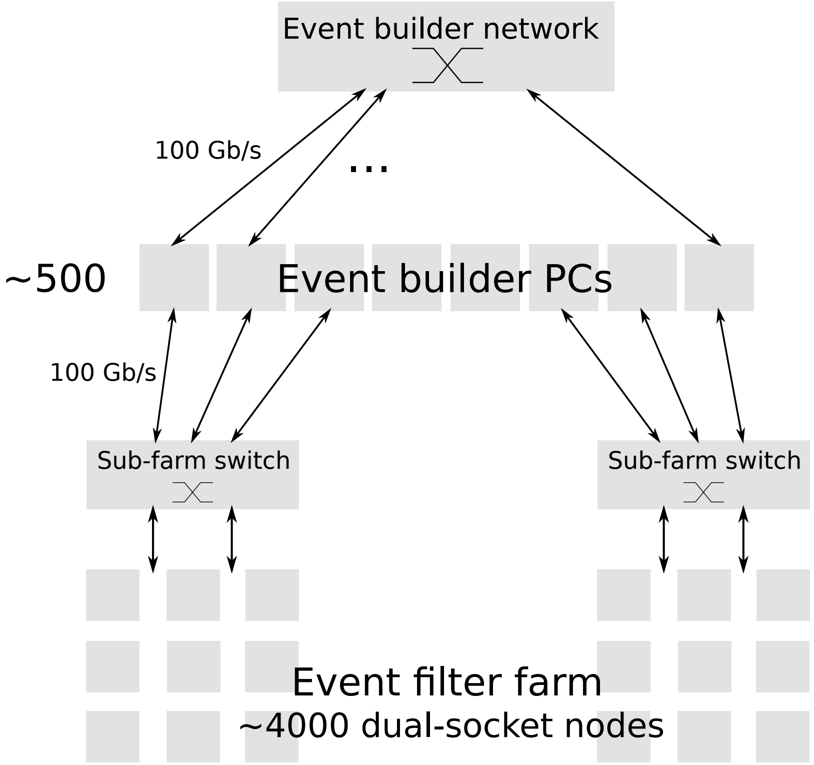


Figure 10.7: Baseline LHCb DAQ architecture.

The event builders (estimated to be 500 in the figure) perform this task by distributing event fragments all-to-all in a load-balanced manner, where every server builds events at an estimated throughput of 100 Gb/s. Built events are then sent through the sub-farm switches to the event filter farm, where events are filtered (HLT1 and HLT2 are performed).

The sole prerequisite imposed by the GPU HLT1 application is that events are built. Hence, it would be possible to alter the baseline DAQ design to equip servers with GPUs at two places: (1) the event builders, or (2) the event filter farm. The first option is depicted in Figure 10.8.

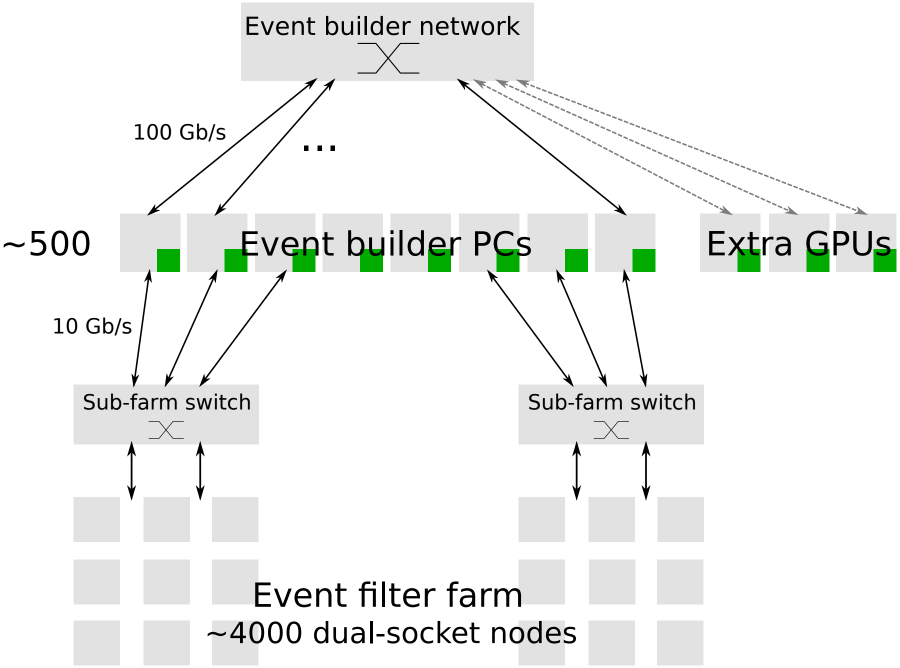


Figure 10.8: DAQ with GPUs in event builders.

In this configuration, every event builder server is equipped with GPUs, while the event filter farm suffers no hardware alteration. Due to the data reduction of the HLT1 application, the bandwidth requirement from each event builder to the event filter farm is reduced by a factor 30. Therefore, a link of 10 Gb/s to the event filter farm suffices to sustain an influx of up to 300 Gb/s in a server. Some motherboards come equipped with 10 G ethernet, effectively freeing the PCIe slot the event filter farm network card would occupy.

Following the estimation of 500 event builder servers, this would imply it would be enough to replace one network card on each event builder server with a GPU that obtains sufficient performance. As it has been shown in table 10.4, any GPU delivering 60 kHz meets this criterium. It is possible to add extra GPU cards to the system, subject to the overhead cost of a server with network connectivity to the event builder and event filter farm networks. This provides a means of scalability to the system (subject to rack space and event builder network port availability) that can also act as contingency in case of GPU failure. Another means of scalability would be to replace or add GPUs to the event builder nodes. However, if the DAQ application requires a perfect load-balancing (as is currently the case), it would be necessary to extend the GPU capabilities of all event builder nodes at once to scale in this manner.

The event building server would need to sustain the necessary memory throughput in order to be able to run the event readout, the event building application, the GPU HLT1 application and the sendout to the event filter farm. Furthermore, the GPU HLT1 application expects data in a different format to the event building application, which would require an additional data transposition. Tests will be performed in the upcoming months to validate this requirement.

There are several parameters that are being explored in face of the DAQ upgrade, which would impact this design, such as (a) the number of TELL40 readout cards required to read out the detector, (b) the possibility of a *compact* configuration with two TELL40s per event builder server, (c) the disponibility to use next-generation PCI-express, *PCIe Gen4*, alongside 200 G network cards, and (d) the availability of next-generation GPUs. The final composition of the event builders will determine the details of a feasible GPU-equipped event builder configuration.

A different possibility is to include the GPUs in the event filter farm, as shown in Figure 10.9. In this configuration, event builders suffer no alteration, and deliver the full 100 Gb/s throughput of events to the event filter farm. GPUs could be placed in different locations in the event filter farm.

GPUs could be coprocessors of a selected number of event filter nodes (tagged *GPU-equipped EF nodes* in the figure). These GPU-equipped event filter nodes would require meeting the GPU slot requirements, and possibly additional bandwidth, such that all the GPUs distributed across event filter nodes sustain the full 30 MHz of events combined. The selected events would then be distributed across all event filter nodes through the sub-farm switches.

Another option would be to prepare GPU accelerator nodes, configured with as many GPUs as possible and enough network bandwidth to support them. The validation of this node's memory throughput would be particularly critical if the additional data transposition is required. The configuration of

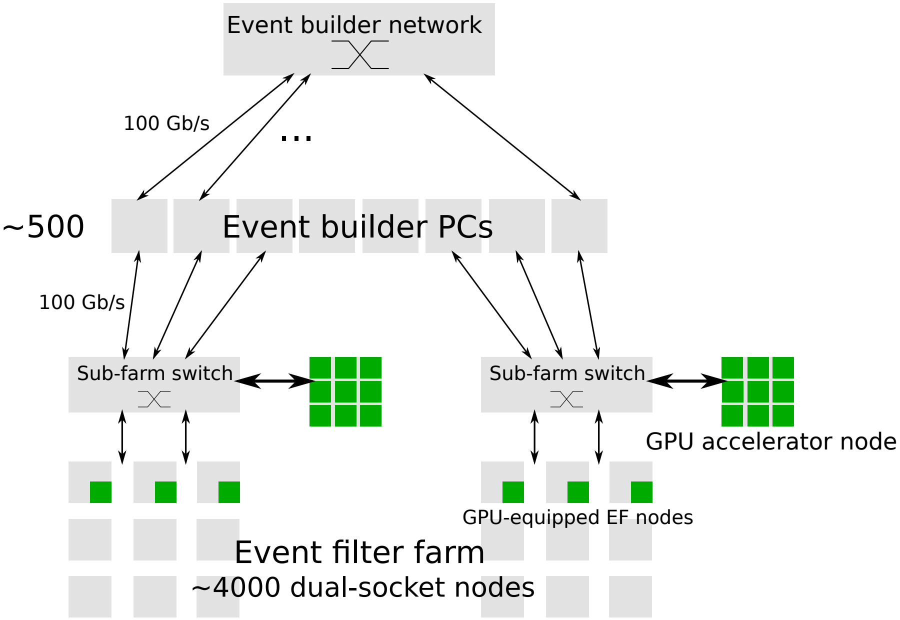


Figure 10.9: DAQ with GPUs in event filter farm.

number of accelerator nodes and their placement across sub-farms would have to be compatible with the event filter farm network. GPU accelerators would process data local to a sub-farm, which would require a matching between GPU resources and CPU resources at a sub-farm level.

The scalability of GPUs in the event filter farm would not have to follow the constraints of the event builders in terms of space, network ports or load-balanced throughput. However, other constraints are introduced: (a) only event filter nodes fulfilling the slot requirements will be able to host GPUs, with the exception of the Tesla T4 which can be hosted even in current LHCb farm nodes; (b) event filter nodes equipped with GPUs have different requirements in terms of network bandwidth, and would require a slot dedicated to a high-bandwidth network card; (c) the amount of GPU accelerator nodes is determined by the computing power of the sub-farm. Fulfilling these constraints, it would be possible to extend the system with any number of GPUs of a varying performance.

Regardless of the chosen configuration in either the event builders or the event filter farm, the GPUs would be available to be used opportunistically in periods of no collisions, as the connectivity to the GPU servers is ensured by definition by the full-duplex capacity of the links. It would be therefore possible to extend the applicability of GPUs to sections of the HLT2 workloads and other applications.

Considerations of the event builder integration

The integration shown in Figure 10.8 is discussed in the following. This is the configuration that is most likely to incur in cost-savings due to several factors: (1) The cost of the network cards and network infrastructure to sustain the 100 Gb/s network from the event builders to the event filter farm is neglected. The cost of the 10 G-based network replacing it is an order of magnitude more affordable. (2) The event filter farm would not need to process the HLT1 reconstruction stage. The server processing time saved is estimated to be 50%, according to Run 2 numbers. While this also holds true for the event filter farm integration options, the event builder integration has the advantage of not impacting parasitic resource consumption in the farm at all (such as the *CPU offload* resource consumption or use of memory bandwidth). (3) Similarly to the event filter farm configuration, GPU resources can be used opportunistically to perform R&D projects, analyses or HLT2 computation during LHC downtime.

The LHCb trigger and online technical design report (TDR) in 2014 [14] described an event builder machine like the one presented in Figure 10.10a. Each of the CPUs in the dual-socket server would be assigned the role of a readout unit (RU) or a builder unit (BU). The RU would receive data through a readout card, and send and receive data through a EB network card (cref. section 1.2). The BU processor would then build the events, and finally send data out through a HLT network card to the event filter farm.

Since then, a compact configuration has been proposed, shown in 10.10b. In this configuration, three PCIe 16x slots are required for each CPU. Each of the CPUs in the event builders would act as a RU and a BU, and perform the readout, EB network send / receive, event building and send to the HLT network card. The memory bandwidth required in this configuration is effectively double the one required in the original proposal.

It would be possible to add GPUs to either of these configurations. In 10.10c, up to two accelerator cards can be added on the slots of the BU CPU. Due to the data reduction, the HLT net-

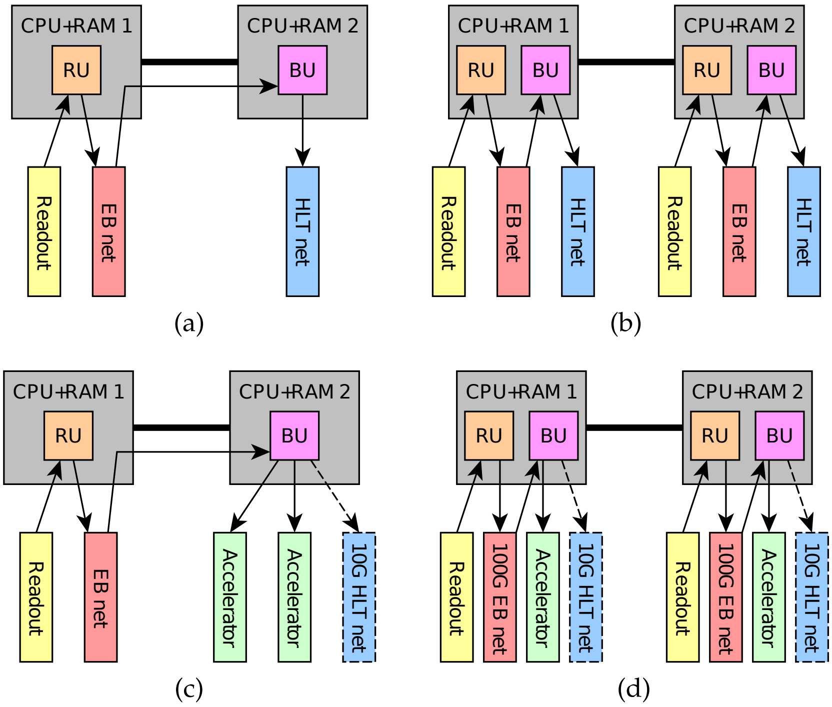


Figure 10.10: (a) (b) Event building server configurations. (c) (d) Event building with accelerator server configuration. Images reproduced with permission from T. Colombo.

work card becomes then a 10 G card, which the mainboard can equip, freeing a slot. A compact EB-HLT1 node configuration is shown in 10.10c, where the accelerator effectively replaces the HLT network card. A data reduction of 10× is assumed in this Figure, thus requiring two 10 G network cards. However, in the more realistic scenario of 30-to-1 HLT1 data reduction, a single 10 G card would be enough to accomodate the 6.67 Gb/s maximum output rate of the node, as shown in table 10.4.

Memory throughput may be a concern in this last scenario. In order to not require any additional memory throughput, the Allen application should support the data format produced by the event building application, with no additional transposition requirements. If that were the case, the memory throughput required in the Allen application would be equivalent to the memory throughput required to send through the HLT network card, which will be commissioned by the LHCb Online team.

The GPU HLT1 to which this thesis has contributed can be run irrespective of the DAQ final implementation. A more detailed study and a test of a vertical-slice of the system will be performed to evaluate the options presented. A strategic placement of the GPUs in the event builders can lead to cost-savings in the final system, with no foreseeable drawbacks in terms of physics efficiency, producing a homogeneous system. Additional work to allow a full-evaluation and commission of a GPU-equipped DAQ will be carried out in the upcoming months.

Part IV

THESIS RESULTS

11 CONCLUSIONS

In this final chapter, the presented work in the document is analyzed and summarized. First, a summary of all chapters is presented. The publications and their content are discussed next. Finally, the future directions stemming from the work in this thesis are presented.

11.1 SUMMARY

The LHCb detector at the Large Hadron Collider at CERN, Switzerland, will be upgraded in 2020, alongside its Data Acquisition system and Event Filter farm. Bunches of particles will collide at a rate of 30 MHz and then be reconstructed and selected in near-time on a farm consisting solely of off-the-shelf components, in a process known as *High Level Trigger*. The data rate is estimated to increase by $40\times$. The LHCb upgrade reconstruction is therefore a massive data-processing problem with potential for parallelization on modern processor architectures.

Hardware and software improvements were projected in the planning stages of the LHCb upgrade [14]. However, the compute resource cost and code performance were underpredicted. In 2016, the shortfall in computing performance to cope with the required data rates was estimated to be between 6 and 10×, and code modernization and alternative algorithms were being studied to close the performance gap. This thesis contributes parallel algorithms and introduces the possibility to run part of the High Level Trigger in Graphics Processing Units (GPUs).

Chapter 1 introduces the upgrade LHCb detector and all the subdetectors that compose it. The Data Acquisition system is divided in three parts: (1) In the Event Readout, raw data are read from the detector. The data are packed and distributed to the event builders. (2) In Event Building, the collision fragments from every readout unit are combined into single coherent data packets known as *events*. (3) The last stage is Event Filtering, where events are reconstructed and selected according to physics indicators. The reconstruction process occurs in software, in two stages known as *High Level Trigger 1* (HLT1) and *High Level Trigger 2* (HLT2). The HLT1 will reduce data at a ratio of 30:1, and will need to sustain the full LHC collision rate at LHCb of 30 MHz. This thesis contributes mostly to the HLT1, which consists of decoding, clustering, pattern recognition, estimators and selection algorithms.

In chapter 2, the foundations of parallel computing are discussed. While hardware architectures behind central processing units (CPUs) have historically optimized *sequential computing*, the trend stopped around 2004 due to heat and power consumption issues. Modern processors offer an increasing number of cores, and parallelism can be exploited with data, instruction, thread and process parallelism. Memory plays a fundamental role in computing, albeit its performance has not increased at the same pace as that of processing power. As such, it is vital to efficiently use the hierarchical memory models of modern processors in order to obtain better performance. Tools such as the *Roofline model* help describe and understand the performance limitations according to memory access requirements of applications.

GPUs are a kind of parallel processors that appeared originally to deal with optimization of workloads involving image and video processing. Since the early 2000s, GPUs became programmable with generic *shader* programs that would allow execution of general purpose code, leading to the general purpose GPU computing programming model (GPGPU). Modern GPUs have further extended programmability of their components, and development environments similar to those of CPUs exist. GPUs are programmable with data-parallel programming models, and are efficient when dealing with massively parallel workloads.

Particle collisions at LHCb yield a rate of 30 million independent *events* that must be reconstructed in real time. The increase in number of collisions per bunch crossing and the removal of the hardware trigger lead to a data rate increase for which a computing hardware replacement does not suffice. The algorithms that conform the HLT reconstruction stages must be optimized to efficiently use the characteristics of modern processors. This thesis contributes in the area of parallel software for High Energy Physics problems. In addition, GPUs are explored as alternative architectures optimized for parallel workloads.

This thesis contributes as well to the fundamental pieces of a GPU High Level Trigger 1 for LHCb.

Part II discusses parallel design and implementations for reconstruction algorithms of the LHCb HLT sequences. All algorithms are parallelized in two dimensions: (1) Each event is a physically independent event, and therefore they are processed in parallel, independently; (2) the processing of each independent event is further parallelized, exploiting *intra-event parallelism*, where possible. The proposed implementations make the assumption the available memory per thread is small, of the order of a megabyte. This design decision permits to target multi and many-core architectures such as GPUs, where the available memory is orders of magnitude smaller than that of CPUs.

Chapter 3 introduces the problem of decoding as it appears in four LHCb subdetectors: Velo, UT, SciFi and the Muon stations. For all cases studied, the number of measurements in each subdetector is first decoded. Then, an accumulated sum yields offsets to the subdetector parts. Finally, the subdetector is decoded in parallel. While the UT, SciFi and Muon decoded data are sufficient to proceed to the next reconstruction stages, in the case of the Velo subdetector an additional *clustering* stage must be performed. A parallel algorithm for performing the Velo clustering is presented and validated.

Chapters 4, 5 and 6 discuss the pattern recognition problem of *track reconstruction*. The problem is defined in chapter 4, with references to existing detectors and a concise description of the efficiency indicators that dictate the goodness of reconstructed tracks, applied to the LHCb case. Track reconstruction are divided in two kinds: (1) local methods form track iteratively, extending one track at a time. (2) Global methods transform the problem into an equivalent formulation, possibly more favorable to the underlying executing hardware, where solutions map to tracks.

The author studies the sequential implementation and proposes a parallel local method for track reconstruction *Search by triplet* in chapter 5, applied to the Velo reconstruction use case. The method iterates every Velo module only once, processing all measurements in each module in parallel. First, track *seeds* are created from neighboring triplets of modules, checking whether the *used flag* has been set on every measurement. Then, tracks are *forwarded* to the next module of the detector, and measurements are flagged as *used* whenever a compatible measurement is encountered. These two steps are iterated until the entire detector has been traversed. A GPU implementation and a CPU implementation have been developed, and a variety of architectures are evaluated. The Search by triplet algorithm is a state-of-the-art tracking algorithm, both in terms of physics efficiency and performance.

The Forward tracking, whereby Velo and UT tracks are extended to the LHCb SciFi subdetector, is studied in chapter 6. First, the current sequential implementation is analyzed, based on a global histogramming technique. The shortcomings for parallelization are identified: the implementation deals with a complex variety of use cases, resulting in a branchy codebase that would perform inefficiently on data-parallel hardware architectures. Instead, a parallel algorithm *Looking Forward* is proposed. Similarly to Search by triplet, a parallel local method is employed by visiting the *x layers* composing the detector in search for neighboring triplets of measurements. A specialization of the triplet search is shown by using modern *Tensor cores* available in NVIDIA GPUs. The method yields a satisfactory physics efficiency and performance.

The Kalman filter is a widely used estimator for trajectories of objects. In LHCb, it is used at several stages in the reconstruction to better estimate the state of particles throughout the detector. A data-parallel implementation of the Kalman filter is proposed in chapter 7. Although the Kalman filter presents conceptually iterative stages with data dependencies, many particles can be reconstructed in parallel. The developed Kalman filter is cross-architecture, and allows for a configurable floating point precision. Three software packages have been developed: (1) *Cross-Kalman Mathtest* is an efficient implementation of the underlying arithmetic in the Kalman filter discrete formulation, optimized across architectures. (2) *Cross-Kalman* mimics the conditions in the LHCb reconstruction, making an efficient use of vector units in CPU processors through a static resource assignment scheduler. (3) *TrackVectorFitter* is a vectorized realization of the fitter in the LHCb framework, available as a 1:1 replacement of the previous fitter. The Kalman filters contributed have been validated. The performance of the implemented Kalman filter arithmetic has been shown to peak the attainable performance in the processors under analysis with Roofline models.

The preexisting LHCb framework *Gaudi* and its specializations are optimized for single event execution per thread, where individual threads are assigned to an event. Gaudi imposes no restrictions on the algorithms, permitting memory to be dynamically allocated, and a branchy control flow. As a consequence, it is not a suitable framework for the requirements of many-core architectures, which require thousands of events be executed in parallel to achieve an efficient use of the underlying hardware. In spite of previous work to enable Gaudi to use GPUs as accelerators, the thousand in-flight events requirement contradicted the cache-friendly single-event sequence execution of Gaudi.

Part III discusses the need of a GPU framework, its proposed design, implementation, and the results achieved with it. The groundwork is developed in chapter 8, where a framework for massively parallel physics reconstruction *Allen* is presented. The framework can be configured with a number of threads, events and repetitions. Each of the Allen threads spawns a *CUDA stream* that is able to offload work to the GPU in an asynchronous and non-blocking manner, and so each thread executes the framework sequence independently. The number of events configured determines the batch of events upon which the sequence will be executed in parallel. The number of repetitions allow to configure the framework to run continuously, allowing more precise time measurements with fewer fluctuations.

The Allen framework allows a sequence of algorithms to be configured, which is then executed in parallel over a number of events. Algorithms are not allowed to dynamically allocate memory, since the built-in GPU memory manager is synchronous and blocking. Instead, a custom dynamic memory manager has been developed. Each thread instantiates its own memory manager, which allocates a fixed memory at the beginning of the program execution. The memory manager handles memory requirements prior to each algorithm's execution asynchronously. The tight memory requirements of GPUs are optimized by developing each algorithm in the framework to use as little memory as possible. The framework allows executing GPU and CPU algorithms, and makes an efficient use of the architectures to achieve state-of-the-art performance.

During the course of this thesis, a full realization of the LHCb HLT1 on GPUs has been achieved with the Allen framework. The sequence consists of up to 70 GPU algorithms, and represents the first GPU-only High Level Trigger sequence ever realized for an LHC experiment.

The physics efficiency of the tracking sequence in the Allen framework is shown in chapter 9. The efficiency numbers were obtained using the framework, which allows validating the results after each run. The Velo, UT and Forward tracking efficiencies, to which this thesis has contributed have been discussed. The efficiencies obtained meet the HLT1 requirements of the LHCb upgrade physics program.

Finally, the performance of the Allen HLT1 application on various GPU models has been discussed in chapter 10. Their performance, power consumption and price have been discussed for all GPU models under consideration. The best performing cards, the Quadro RTX 6000, Tesla V100 32GB, and GeForce RTX 2080 Ti, achieve a performance close to 60 kHz, while the Tesla T4 reaches a performance of 29 kHz. The full collision rate of 30 MHz would be realizable with 500 cards (60 kHz) and 1000 cards (30 kHz), which give a notion of the size of a prospective production system.

The LHCb experiment faced an upgrade phase required by the evolution in detector technologies, physics goals and increase in accelerator's collision rate. Early attempts to meet those goals highlighted serious difficulties in achieving the required performance, where scaling the existing system was not a valid alternative.

This thesis has contributed the fundamental building blocks of a GPU LHCb HLT1, and has shown the feasibility to efficiently run HEP workloads in the LHCb reconstruction use case. It has been demonstrated that it is possible to implement the full HLT1 on GPUs, serving as an innovative drop-in replacement for classic server farms with potential to incur into cost-savings in the final system configuration, while preserving the physics and throughput goals of the experiment upgrade.

11.2 PUBLICATIONS

The following publications have been produced as a result of the work in this thesis:

- D. H. Cámpora Pérez, N. Neufeld, and A. Riscos Núñez. A Fast Local Algorithm for Track Reconstruction on Parallel

Architectures. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2019), pp. 698–707. Included as appendix A.

The parallel local tracking algorithm *Search by triplet* was presented as an efficient solution for the Velo reconstruction. The same design was carried over from the original GPU implementation to a CPU implementation using the SPMD programming model, obtaining a performance speedup on both architectures. In order to test the GPU implementation, a *GPU sequence framework* was presented. The performance was optimized for the architectures under analysis by performing parameter scans. Search by triplet is a state-of-the-art tracking algorithm for the LHCb Velo subdetector.

- P. Fernandez Declara, D. H. Cámpora Pérez, J. Garcia-Blas, D. vom Bruch, J. D. Garcia, and N. Neufeld. A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures. In: *IEEE Access* (2019), pp. 91612–91626.

  The UT decoding algorithm and the UT tracking algorithm were designed as parallel efficient algorithms for the GPU. The UT sequence contributed as part of this thesis decodes efficiently the UT detector into a sorted structure of arrays, which enables good performance on the UT sequence overall. The parameters of the algorithm allow a configurable trade off between performance and physics efficiency. The default configuration has been tuned to fulfill the physics program for the UT detector in LHCb.

- D. H. Cámpora Pérez. LHCb Kalman Filter cross architecture studies. In: *Journal of Physics: Conference Series* 898.3 (2017), p. 32052.
- D. H. Cámpora Pérez, O. Awile, O. Bouizi, N. Neufeld. Cross-architecture Kalman filter benchmarks on modern hardware platforms. In: *Journal of Physics: Conference Series* 1085 (Sept. 2018), p. 032046.
- D. H. Cámpora Pérez, O. Awile, and C. Potterat. A High-Throughput Kalman Filter for Modern SIMD Architectures. In: *Euro-Par 2017: Parallel Processing Workshops*. Springer International Publishing, 2018, pp. 378–389.

- D. H. Cámpora Pérez and O. Awile. An efficient low-rank Kalman filter for modern SIMD architectures. In: *Concurrency and Computation: Practice and Experience* 30.23 (Dec. 2018), e4483. Included as appendix B.

  A vectorized efficient Kalman filter was designed and implemented over a variety of architectures. Data-parallelism is achieved through fitting several particles at a time, and a static scheduler is employed to occupy processor units on each fit iteration. Three software packages were produced from this work: An arithmetic test allows checking the efficiency of the implementation; a cross-architecture Kalman filter of configurable precision was produced as a standalone application; and the vectorized Kalman filter was also integrated into the LHCb framework. The Kalman filter work has been presented iteratively on the above publications. The vectorized Kalman filter is an efficient parallel implementation which will impact the performance of the LHCb upgrade reconstruction. In addition, it serves as a self-contained cross-architecture LHCb benchmark.

11.3 FUTURE WORK

The work presented in this thesis has numerous implications, and opens research paths for many-core architectures in High Energy Physics.

The amenability of many-core architectures like GPUs to solve High Energy Physics problems has been amply discussed. Parallel decoding algorithms, which are typically solved sequentially, can be solved in parallel by following a strategy loosely consisting in a pre-decoding phase, a prefix sum, and finally a parallel decoding algorithm. Other subdetector decoding algorithms may be ported following a similar design.

The tracking algorithms developed exploit a local method in parallel. The triplet search design has been shown to efficiently reconstruct the Velo tracking detector, as well as the seeding phase of the Forward tracking. Other tracking detectors may also be efficiently reconstructed by employing the presented technique. In addition, the developed reconstruction algorithms may be reused for the tighter requirements of the High Level Trigger 2 reconstruction sequence in LHCb. Configuration options will be explored to extend the current algorithms, both to yield better physics efficiency and to be more generic to other detectors.

The vectorized Kalman filter work has been integrated in the LHCb Gaudi sequence. In order to use the vectorized Kalman filter as part of the HLT2 of the next LHCb run, the HLT2 sequence will be tested with various configurations. Further optimizations will be performed in components used by the Kalman filter, such as the Runge Kutta extrapolator.

Various alternatives are being considered to port the entire Allen sequence to run on CPUs. An automatic translator tool from CUDA to SPMD is under development, which currently allows converting simple CUDA algorithms into SPMD. Middlewares like Kokkos or Alpaka may be considered as alternatives to maintain a single codebase which can run on CPU hardware as well as GPUs. Alternatively, simple CUDA functions using a restricted set of GPU functionalities may be translatable if thread-strided loops are always used. In order to deploy a GPU HLT1 in LHCb, either of these solutions will required to be implemented.

A complete HLT1 on GPUs has been developed and its main components have been presented in this thesis. In order to enable a production-ready environment, work will proceed in optimizing the sequence and integrating with preexisting components of the LHCb software infrastructure. A cost comparison and full system study will be performed in the future to decide whether the GPU sequence is realized in the upcoming LHCb upgrade.

The Allen framework is extensible, and enables the LHCb community to port other algorithms in the reconstruction to GPUs. Other subdetector algorithms are likely to be portable to GPUs in similar manners and will be explored in the future. The presented GPU work demonstrates a full sequence on GPUs is possible under the real-time tight constraints of an LHC experiment reconstruction, and it may lead to other experiments taking inspiration in the presented work to explore utilization of coprocessors. The Allen framework is being considered to be extended to serve as a generic framework to support the reconstruction of any High Energy Physics workload.

APPENDICES

A

A FAST LOCAL ALGORITHM FOR TRACK RECONSTRUCTION ON PARALLEL ARCHITECTURES

A fast local algorithm for track reconstruction on parallel architectures

Daniel Hugo Cámpora Pérez*†
Niko Neufeld*
Agustín Riscos Núñez†

* CERN
CH-1211 Geneva 23
Geneva, Switzerland
Email: dcampora@cern.ch

† Research Group on Natural Computing
Universidad de Sevilla
ETSI Informática, Av. Reina Mercedes, s/n,
41012, Sevilla, Spain

***Abstract*—The reconstruction of particle trajectories, tracking, is a central process in the reconstruction of particle collisions in High Energy Physics detectors. At the LHCb detector in the Large Hadron Collider, bunches of particles collide 30 million times per second. These collisions produce about 10^9 particle trajectories per second that need to be reconstructed in real time, in order to filter and store data. Upcoming improvements in the LHCb detector will deprecate the hardware filter in favour of a full software filter, posing a computing challenge that requires a renovation of current algorithms and the underlying hardware.**

We present Search by triplet, a local tracking algorithm optimized for parallel architectures. We design our algorithm reducing Read-After-Write dependencies as well as conditional branches, incrementing the potential for parallelization. We analyze the complexity of our algorithm and validate our results.

We show the scaling of our algorithm for an increasing number of collision events. We show sustained tests for our algorithm sequence given a simulated dataflow. We develop CPU and GPU implementations of our work, and hide the transmission times between device and host by executing a multi-stream pipeline.

Our results provide a reliable basis for an informed assessment on the feasibility of LHCb event reconstruction on parallel architectures, enabling us to develop cost models for upcoming technology upgrades. The created software infrastructure is extensible and permits the addition of subsequent reconstruction algorithms.

I. Introduction

LHCb is a large particle physics detector operating at the CERN Large Hadron Collider [1]. From 2020 on it will produce data at a rate of 40 Tbit/s [2]. A data selection will be performed in order to record interesting events [1] from a particle physics standpoint. The data acquisition system will be upgraded [3] to process all events in a commodity processor farm, deprecating the current hardware trigger. The increase in data rate and the removal of the hardware trigger pose a real-time computing challenge.

Different solutions are being studied to be able to process this enormous volume of data. The current LHCb trigger farm is composed solely of Intel Xeon-based servers, however the recent adoption of alternative architectures and accelerators in other detectors' data acquisition systems are an indication that other solutions may also be feasible [4] [5] [6]. Software demonstrators are fundamental towards implementing new architectures to the LHCb trigger farm, where price performance and software maintainability aspects should be taken into account.

Track reconstruction consists in determining the trajectories of particles from the signal pixel *hits* left on their path. The upgraded vertex locator (Velo) detector will span 52 consecutive silicon pixel modules, placed very closely to the interaction point [7]. The Velo reconstruction constitutes the first stage of tracking in LHCb. Tracks created at this stage are used for determining the locations of the collisions, and serve as a seed and are extended to subsequent LHCb tracking detectors. Hence, the Velo reconstruction is fundamental for the correct functioning of LHCb.

Various track reconstruction techniques have been explored in literature. Local methods find tracks iteratively. The baseline LHCb Velo reconstruction algorithm consists in a *track forwarding* technique, based on finding candidate pairs and extending them over iterative detector modules [8]. The need for flagging visited hits sequentially makes this technique unsuitable without modification to parallel architectures. Finding all compatible triplets can be parallelized dropping the flagging mechanism, like in the seeding phase of [9]. However, this is inefficient for densely populated detectors. Local methods are commonly used in conjunction with an estimator like the Kalman filter [10] to fit forming tracks and select hits [11]. Spatial reductions like KD-tree structures [12] or search windows help reduce the dimensionality of hits under consideration.

On the other hand, global methods adapt an equivalent formulation of the problem, where solutions map to tracks. The Hough transform [13] [14] converts all hit points into a histogram representation in polar coordinates, where peaks are equivalent to compatible hits. The Retina algorithm [15] builds a heatmap for each hit to determine compatible tracks. The *automata* technique [16] [4] consists in creating a weighted graph representing the connectivity of every hit, and traversing it to find the best tracks.

We present *Search by triplet*, a fast local method optimized for Velo track reconstruction on parallel architectures. We sort hits in all modules and define tight search windows. We adapt the track forwarding technique to expose parallelization

[1] An *event* corresponds to a single crossing of the Large Hadron Collider proton beams.

with an iterative two-step tracking. We iterate over each detector module only once, maximizing temporal and spatial locality. We flag hits while maintaining parallelizability of each individual step, avoiding Read-After-Write (RAW) data dependencies. We employ a least-squares fit for track fitting, given the expected tracks in the Velo region are straight lines due to the lack of magnetic field interaction. We use *Monte Carlo* simulation of LHCb particle collisions. This allows us to validate our algorithm by comparing trajectories generated by the simulation, also referred to as *true particle trajectories*, against the reconstructed tracks obtained as output of our algorithm.

We develop our algorithm using the SIMT programming model [17], targeting GPGPUs. In order to efficiently use the resources available on GPUs, we create a software framework for performing data parallel event reconstruction. We employ a dynamic GPU memory manager to handle algorithmic data requirements, which allocates and frees GPU memory segments based on a data dependency tree. Our framework can run several GPU *streams* in parallel. We hide the latency of data transmissions by employing a pipeline that reconstructs events while performing memory read and write operations.

We translate our algorithm to the SPMD programming model [18], producing a vectorized algorithm suitable for CPUs. We discuss the design of our algorithm and assess its performance and scalability on modern CPUs and GPUs. We run our software in several streams and study how many concurrent streams are required for saturating our GPUs.

Our work will directly impact the decision on what hardware to acquire for the upcoming upgrade of the processing farm of LHCb. The developed GPU framework is extensible and allows for other parts of the reconstruction to be implemented and evaluated on many-core architectures.

II. Velo reconstruction

The upgraded Velo detector will be a pixel-based particle detector [7], spanning a total of 52 detector modules. A schematic of the detector is shown in Figure 1. The detector modules are placed in two sides, with 26 modules on each side. The interaction region marks where the collisions are expected to occur. The nominal acceptance angle of the LHCb detector is $15-300$ $mrad$ in the forward region. The Velo detector will detect by design all particles produced in *primary vertices*[2] in the LHCb coverage angle on at least 3 modules [19].

In the Velo region, the effect of the LHCb magnet is negligible. Particle tracks detected in the Velo detector are therefore expected to be straight lines. Reconstructed Velo tracks serve as seeds for reconstructing particle trajectories through the other LHCb tracking detectors, and allow the reconstruction of vertices where the collisions happened. Additionally, Velo reconstruction occurs early in the LHCb reconstruction process. Therefore, the Velo reconstruction is of paramount importance towards a successful trigger.

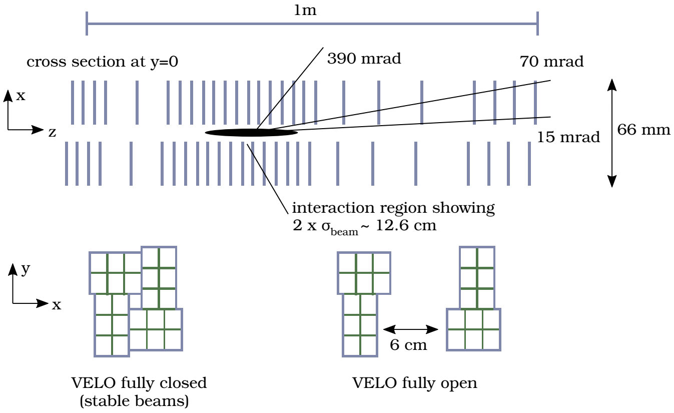


Figure 1: A schematic of the upgraded Velo detector. The top image shows a section in the XZ plane, with detector modules laying in two sides. The images at the bottom show a frontal view of each module in the XY plane, with subdivisions indicating detector chips. Each detector chip has a resolution of 256×256 pixels.

The physics quality of found tracks can be evaluated according to three indicators [20]. Particles are considered *reconstructible* in the Velo subdetector if at least three hits were left in different modules on its path.

- The *track reconstruction efficiency* is the probability to reconstruct a particle travelling through the detector, and can be determined by the ratio between the reconstructed tracks of reconstructible particles, over all the reconstructible particles:

$$\frac{N_{\text{reconstructed and reconstructible}}}{N_{\text{reconstructible}}} \tag{1}$$

- The *fake track fraction* is the ratio between the reconstructed tracks that are not associated to any Monte Carlo particle (*fake tracks*), and all the reconstructed tracks:

$$\frac{N_{\text{fake tracks}}}{N_{\text{reconstructed tracks}}} \tag{2}$$

- Finally, the *clone track fraction* refers to the fraction of tracks associated to the same Monte Carlo particle as another reconstructed track:

$$\frac{N_{\text{clone tracks}}}{N_{\text{reconstructed tracks}}} \tag{3}$$

In spite of the simplicity of Velo trajectories, Velo track reconstruction should maximize reconstruction efficiency, minimize fake fraction and clone fraction at a rate of up to 10^9 tracks per second. The Velo reconstruction algorithm is one of the main time contributors in the current first stage of software trigger [3], also referred to as *High Level Trigger 1*, and therefore it would have a high theoretical speedup if it were parallelized according to Amdahl's law [21].

[2] A primary vertex is the reconstructed location of an individual particle collision.

A. Sequential algorithm

Track forwarding is a local method consisting in finding track candidates and forwarding them over the rest of detector modules. The nominal LHCb algorithm [8] finds a candidate pair of hits fulfilling a *compatibility condition* in neighbouring modules on the same side. Then, the forwarding phase consists in extrapolating the candidate's trajectory to subsequent modules, finding hits that fulfill an *extrapolation condition*. Tracks are forwarded until either no modules remain, or no hits fulfilling the extrapolation condition are found on two consecutive modules on the same side. Hits are flagged upon finding tracks of 4 or more hits, so they are not considered for other tracks. The process is repeated until no candidates remain.

Additional design decisions specific to the Velo detector have been taken in the sequential algorithm. Tracks are required to consist of at least three hits. Three-hit tracks are required to have no flagged hits and to pass a fit cut, since they could potentially be formed out of noise. This is less likely on tracks with more hits, as each additional track hit has to fulfill the extrapolation condition.

A number of modules can be missed in the forwarding phase. This stems from a physical condition: A particle may not leave a signal on a module in its path. The probability of a track missing a signal in a module while having left signals in the precedent and posterior modules is under 1%. However, the probability of a track missing two consecutive modules on the same side is under 0.01%. Therefore, the sequential algorithm allows for a missing module on the last signal side.

The sequential algorithm has been validated to deliver the required physics performance. However, in our opinion there are some fundamental design shortcomings. It should be noted that the solution found by the algorithm is deterministic, although it depends on the order in which hits are considered. Hits are sorted prior to the reconstruction taking place, and the order must be strictly followed for the results to be reproducible. Additionally, hits are required not to be flagged before checking the compatibility or extrapolation conditions. These two facts are implicit RAW dependencies, and make parallelization in the algorithm unfeasible without blocking conditions.

III. Search by triplet

We propose a data parallel approach to Velo reconstruction. Events are physically independent, and can be reconstructed in parallel. Within an event, several tracks can be reconstructed in parallel. Also, events are sufficiently small that they are amenable to be processed by relatively small kernels, avoiding register spilling.

The *Search by triplet* algorithm is composed of five sub-algorithms that are described independently. For all complexity considerations, we generalize our algorithm to m consecutive detector modules, and an average number of hits in each module n.

Sort by phi

Given a list of module hits as input, no assumption can be made as to the order of hits inside each module. This algorithm sorts each of the module hit sets increasingly according to φ, calculated as the *2-argument arctangent* for each hit with respect to the origin of coordinates. Given the expected number of hits is small, a method employing *shared memory*[3] is used for storing the newly calculated φ and finding the sort permutation. The permutation is then applied to hit coordinates, yielding sorted *Structure of Arrays* for each module. A parallel insertion sort method has been implemented for calculating the permutation. The complexity of this algorithm is $O(m \cdot n^2)$.

Find candidate windows

In order to minimize the amount of candidates considered in subsequent steps, the first and last hits in the region of acceptance in the preceding and following modules are calculated for every hit. Figure 2 depicts this process. Hit c_0 would have one candidate on both the preceding and following modules, whereas c_1 would have one and two respectively. This process is repeated for every hit in every module that has a preceding and following module. All modules are processed in parallel. In order to find the first and last candidate, a binary search in φ is performed. The complexity of this algorithm is therefore $O(m \cdot n \cdot log(n))$.

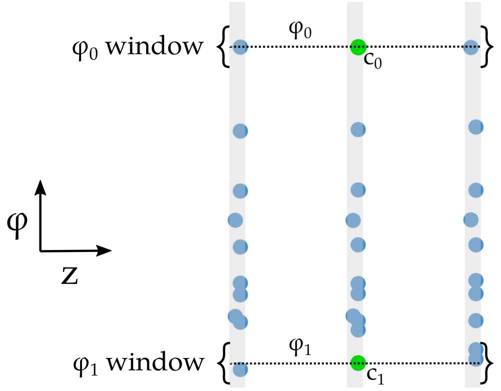


Figure 2: Three consecutive modules with hits are depicted. For hits c_0 and c_1, their respective φ angles and opening windows in the preceding and following modules are highlighted. c_0 has a compatible hit in the preceding module and another one in the following module, on the left and right respectively. c_1 by contrast has one hit in the preceding module and two in the following module.

Track seeding and track forwarding

The *track seeding* algorithm operates on three consecutive modules at a time. It assigns *threads*[4] to hits in the middle

[3] In our GPU implementation, the configurable L1-cache shared memory is employed, due to its low latency and high throughput. In our CPU implementation, main memory is employed.

[4] The CUDA terminology *thread* and *block* is employed here. Equivalently for the CPU implementation, *program instance* and *gang* [18].

module, and each of these threads checks the preceding and following modules for compatible hits. The previously calculated φ windows are employed to this end. For every hit in the middle module, all triplets in the search window are fitted and compared, and the best one is kept as a *track seed*. If there are no hits in either of the search windows, or the least-squares fit χ^2 is over a certain compatibility threshold, no track is formed for that hit.

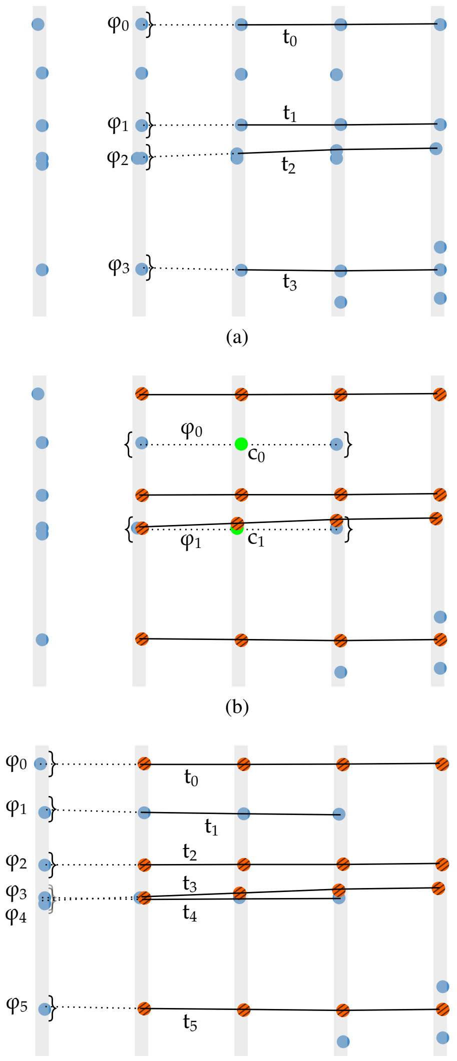
The multiplicity of triplets to be analyzed varies from hit to hit. A variable workload has a negative impact on performance in parallel architectures, as threads in a block would become idle until all workloads are finished. For this reason, multiple threads can be assigned to process the same hit. In this fashion, if there is one hit with a very high workload, its performance impact is diminished as it will be processed in parallel. The amount of threads assigned to each hit is configurable in our algorithm. Additionally, in cases where the number of hits is under a certain threshold, threads are dynamically reassigned to process one of the hits left, minimizing idle threads.

Since several threads may process the same hit, a synchronization mechanism is required in order to guarantee that only the best triplet for every one middle hit is kept as a track seed. This synchronization mechanism utilizes shared memory, where every thread stores its best found triplet, alongside its fit χ^2. Once all threads have computed their assigned triplets, the χ^2 values assigned to the same middle hit are compared, and only the best fits for each middle hit are kept. After all found triplets have been checked, threads assign to the next hit.

This tiled processing mechanism for finding triplets is applied in first instance to the modules that are further apart from the collision point, as they present the lowest hit multiplicity. This algorithm yields a deterministic solution, that is, the obtained set of triplets is independent of the order in which hits are processed. Each triplet is the *seed* of a forming track, and in the forwarding phase we will try to extend them by looking for hits on the following modules.

Track forwarding operates on forming tracks and *forwards* them to a specified module. Threads are assigned to forming tracks. For every track, the segment defined by its last two hits is extrapolated to the working module. Then, a binary search is performed in φ in the module. The extrapolated segment is checked against the hits as a function of their distance in the module (dx, dy) and the distance along the beam axis from the last hit to the current one (dz): $\frac{dx^2 + dy^2}{dz^2}$. The hit that minimizes the extrapolation function and is under a certain threshold is then appended to the forming track, which is kept for a posterior track forwarding step. A configurable number of modules with no compatible hits are allowed when forming a track. If this number is exceeded, three-hit tracks are stored in a *weak tracks* container for posterior consideration, and tracks with four or more hits are stored in the final tracks container.

When a compatible hit is found, track forwarding *flags* all hits of the forming track. The flag can then be used in the track seeding algorithm, imposing the condition that all hits in a track seed be unflagged. Flags are populated in track forwarding, and are read in track seeding. Therefore, this imposes a Read-After-Write dependency between forwarding and seeding, and the requirement of inter-algorithm synchronization.



(a)

(b)

(c)

Figure 3: Three processing stages of *Search by triplet* are depicted. (a) *Track forwarding* operates on the second module from the left. For each track t_i, the segment given from its last two hits is extrapolated to the processing module, and φ_i is calculated. A search window is opened and the hit within this window that minimizes the extrapolation function is chosen. If a compatible hit is found, all hits in the track are flagged. (b) *Track seeding* operates in the middle module. The highlighted hits c_i are considered for creating new seeds. Flagged hits are ignored. (c) *Track forwarding* in the leftmost module. All forming tracks are considered for the search. Since tracks t_3 and t_4 present overlapping search windows, they may be extended with the same hit.

Track seeding and track forwarding are the building blocks of our tracking algorithm. Figure 3 depicts five consecutive modules being processed. A track seeding stage (b) is interleaved between track forwarding stages (a) and (c). This mechanism benefits from temporal and spatial locality, since the data-flow is such that module hits are revisited after every forwarding stage. The module processed in the track forwarding stage in Figure 3a is revisited in the subsequent track seeding stage in Figure 3b. This control-flow is compatible with our flagging mechanism, and guarantees flags be populated prior to seeding stages.

Both seeding and forwarding exploit intra-event parallelism, and several independent events are assigned to independent blocks, for inter-event parallelism. The worst-case complexity of track seeding is $O(m \cdot n^3)$. Track forwarding performs a binary search on every module, and the maximum number of tracks created is bound by $m \cdot n$. Therefore, its worst-case complexity is $O(m^2 \cdot n \cdot log(n))$.

Weak track filter

The *weak track filter* algorithm operates on three-hit tracks, and appends them to the final tracks container given that two conditions are met: (1) all three hits must not be flagged, and (2) the least-squares fit χ^2 of the track must be under a certain threshold.

Additionally, a least-squares fit is calculated for every accepted track, required for subsequent reconstruction algorithms, and stored in an SOA container. The complexity of the weak track filter is $O(m^2 \cdot n)$.

IV. GPU SEQUENCE FRAMEWORK

We have developed an extensible GPU sequence framework[5] in order to perform parallel event reconstruction on many-core architectures. Our framework utilizes CUDA to offload computation to a GPU accelerator. We present here the results of the Velo reconstruction, although an evolving codebase is under development in order to accommodate the entire first stage of the software trigger High Level Trigger 1.

Figure 4 depicts an architectural view of the framework. Our framework reads simulated Monte Carlo events from input binary files, which have been generated in the LHCb reconstruction framework. Geometry descriptions of the detector are also read in this fashion, and are kept constant throughout the execution of the reconstruction sequence.

Control flow

Our framework is multi-threaded. Each of the CPU threads employs one GPU stream to guarantee asynchronous execution of events. A configurable number of events is executed in parallel on every GPU stream. Since every event is physically independent, no communication is required between CPU threads or GPU streams.

The reconstruction of physics events is performed in a sequence of algorithms executed on one CPU thread - GPU stream pair. This sequence is configurable, and consists in device-to-host and host-to-device data transmissions, as well as data decoding and reconstruction algorithms. In order to prevent execution stalls, all data transmissions are invoked through their GPU stream. A pipeline is effectively created when three or more *thread-stream* pairs are created, allowing for concurrent two-way transmission and execution.

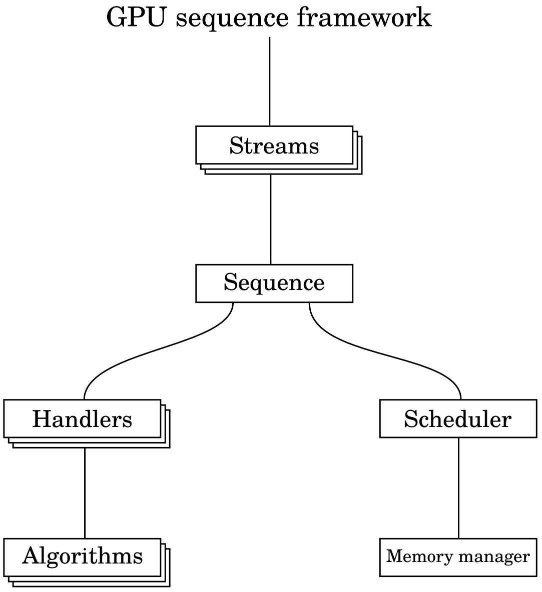


Figure 4: A schematic of the GPU sequence framework. Our software reads binary input files containing simulated Monte Carlo events. Several GPU *streams* can be executed in parallel, each of them with their own *sequence* of algorithms. The memory required by every algorithm in the sequence is managed by the *scheduler*, which employs a *memory manager* with a predetermined memory availability.

The sequence operates through *handlers* that encapsulate algorithms. A handler provides a common façade to an arbitrarily complex control-flow. Code repetition is avoided by encapsulating common tools behind handlers for algorithms such as *prefix sum* or *sorting*, that would otherwise require explicitly instantiating various algorithms.

Data flow

In CUDA, dynamic memory allocation operations such as *cudaMalloc* or *cudaFree* cannot be executed asynchronously, and require all streams to synchronize. The data flow has been developed to solve this central issue. We configure the amount of data to be reserved for every thread-stream pair and allocate it prior to launching the thread. An upper bound for the entire algorithm sequence is therefore necessary, and is currently obtained experimentally.

We have developed a *memory manager* that operates with the allocated memory of the thread-stream. It keeps a view of the memory in segments, and provides non-blocking *malloc* and *free* implementations.

[5] The GPU sequence framework and Search by triplet are available under https://gitlab.cern.ch/dcampora/search_by_triplet, tag v1.0.

Data dependencies are known a priori for each algorithm. Upon configuring the sequence, the *scheduler* iterates the dependencies and determines when arguments need to be allocated or freed. Prior to the execution of every algorithm in the sequence, the scheduler is invoked in order to prepare the required arguments. The scheduler employs an instance of the memory manager to achieve asynchronous memory management.

V. CPU IMPLEMENTATION

We have translated our code to the *Single Program Multiple Data* (SPMD) format employed by the *Intel SPMD Program Compiler* [18] [6]. This method allows our algorithm to be executed on any available ISPC target CPU [7], while preserving our algorithm design with minimal modifications, yielding the exact same result as the GPU counterpart.

The resulting SPMD code is vectorized by the ISPC compiler. The execution model of ISPC executes a *gang of program instances* in parallel, using the vector units available in a processor. The execution of every instruction is masked, similarly to how a *warp* executes threads on a GPU. ISPC allows compilation with a configurable execution mask size and gang size. Additionally, the desired set of *vector extensions*[8] can be configured.

Our CPU implementation is compatible with the Monte Carlo events and geometry descriptions of the GPU sequence framework. We have predefined the Velo sequence, with the same set of algorithms as the many-core model. Events are executed in parallel across different CPU threads via a minimal multi-threading wrapper, while intra-event parallelism is handled by ISPC assigning work to vector units. In order to be able to rigorously compare both implementations, we have avoided any usage of the C++ standard library for common algorithms.

VI. PERFORMANCE ANALYSIS

We have carried out a performance analysis over a variety of hardware, described in tables I and II. The CPUs under analysis are from two different vendors, Intel and AMD. The Skylake processor *Silver 4114* supports the AVX512 instruction set, whereas the Broadwell and EPYC processor only support AVX2. A dual-socket configuration for each server has been tested, with two identical processors of each kind.

The GPUs have different memory types, gaming cards have GDDR5 whereas the scientific card Tesla V100 is equipped with High Bandwidth Memory (HBM2). The *10-series* gaming cards implement the NVIDIA Pascal architecture, the scientific card implements the Volta architecture, and the *RTX 2080 Ti* implements the more recent Turing architecture. The CUDA compute capability of either of the cards is enough to support our implementation of Search by triplet. The memory of the cards impacts the amount of streams and events that can be executed concurrently.

Feature	Intel Xeon Broadwell E5-2630	Intel Xeon Silver 4114	AMD EPYC 7301
# cores	20	20	16
Max freq.	3.1 GHz	3.0 GHz	2.7 GHz
Cache (L3)	25 MB	13.75 MiB	64 MiB
DRAM	64 GiB	64 GiB	64 GiB
SIMD capability	AVX2	AVX512	AVX2
MSRP	667 $	694 $	948 $

Table I: CPU hardware used for our tests. We compare a Broadwell processor (Intel Xeon E5-2630), a Skylake processor (Intel Xeon Silver 4114) and an AMD processor.

Feature	Geforce GTX 1060	Geforce GTX 1080 Ti	Geforce RTX 2080 Ti	Tesla V100
# cores	1280 (CUDA)	3584 (CUDA)	4352 (CUDA)	5120 (CUDA)
Max freq.	1.81 GHz	1.67 GHz	1.545 GHz	1.37 GHz
Cache (L2)	1.5 MiB	2.75 MiB	6 MiB	6 MiB
DRAM	5.94 GiB GDDR5	10.92 GiB GDDR5	10.92 GiB GDDR5	32 GiB HBM2
CUDA capability	6.1	6.1	7.5	7.0
MSRP	249 $	699 $	1199 $	8899 $

Table II: GPU hardware used for our tests. We compare a mid-class gaming graphics card (Geforce GTX 1060), two high-end gaming graphics card (Geforce GTX 1080 Ti and Geforce RTX 2080 Ti) and a scientific card (Tesla V100).

We employ a validation method based on well-established metrics for our algorithm (cref. section II). We obtain a deterministic result across all devices. The use of Monte Carlo data for validation is the standard for validating reconstruction algorithms. The presented results have been validated to produce acceptable physics performance.

We run a configurable number of events s for a number of repetitions r. In each repetition, event data submission and retrieval are performed. The amount of streams t is also configurable. We measure the performance of our sequence by using external counters. We obtain the *wall clock* execution time, and factor in the number of events that have been processed. Our framework presents the performance of a run as the number of events executed per unit of time, measured as frequency (Hz).

Search by triplet presents several free parameters that alter the computing performance. Each of the discussed algorithms are encapsulated in one CUDA kernel, and can be tweaked with respect to the number of blocks and number of threads on each invocation. We have identified, by using local search, the parameter values that provide best performance for the entire sequence, and the resulting configuration is shown in table III. Even though individual kernels may be faster under other configurations, these values empirically showed

[6] Search by triplet SPMD is available under https://gitlab.cern.ch/dcampora/search_by_triplet_spmd, tag v1.0.

[7] At the time of writing, ISPC supports as targets: x86 with SSE2, x86-64, ARM and NVIDIA PTX.

[8] The following vector extensions were tested: SSE2, SSE4, AVX, AVX2, AVX512 (Skylake).

the best performance-to-resource-usage ratio, resulting in a more efficient CUDA scheduler resource assignment. We have found this configuration to provide best performance across all tested devices.

Kernel	# blocks	# threads
sort by phi	# events in execution	64
find candidate windows	{# events in execution, # Velo middle modules}	128
track seeding and track forwarding	# events in execution	32
weak track filter	# events in execution	256

Table III: Best configuration found in local search for each CUDA kernel. We have optimized our configurations minimizing the overall wall clock execution time. Individual algorithms may get faster with different configurations, but the effect on the overall performance is also impacted by resource usage, since other concurrent streams may be blocked.

The configuration of *Search by triplet SPMD* has also been tweaked for each of the CPUs under consideration. A mask of 32 bits was found to yield the best performance for all processors. This is to be expected, as the ISPC guidelines state the mask should have a length of the most used datatypes, which are 32-bit types in our algorithm. The gang size and vector extension has also been tested, and table IV depicts the optimal configurations found for each processor. In the AMD processor, the preferred vector extension and gang size were AVX1 and four, in contrast with the Intel Broadwell processor, which could be due to the differing number of ports and functional units available on both processors.

Processor	Vector extension	Mask size	Gang size
Intel Xeon Broadwell E5-2630	AVX2	32	8
Intel Xeon Silver 4114	AVX512	32	16
AMD EPYC 7301	AVX1	32	4

Table IV: Best configuration for Search by triplet SPMD for each processor. Both Intel processors benefit from their latest available instruction set. The AMD processor benefits from an AVX1 configuration with a gang size of 4, despite supporting AVX2. This could be due to particularities involving the number of ports and functional units of the processor.

The peak performance configuration achieved with every processor is compared in Figure 5. The AMD EPYC processor underperforms when compared to its other CPU competitors. The AVX512 vector extensions in the Skylake processor show a discrete 6% performance speedup over the AVX2 Broadwell processor. Even though our CPU solution is vectorized and utilizes all available threads, all of the tested high-end and scientific GPUs outperform the CPUs in consideration.

The mid-class Geforce GTX 1060 yields a similar performance to the Intel processors under analysis. The projected speedup between the Geforce GTX 1060 and the Geforce GTX 1080 Ti according to their number of cores and maximum frequency is 2.58×, even though this does not take into consideration cache size or base frequency. We observe a speedup of 2.41×, showing our algorithm scales to higher-end architectures. We attribute the difference in performance across the two high-end gaming cards Geforce GTX 1080 Ti and Geforce RTX 2080 Ti to be a combined effect of both the increase in CUDA cores and in L2 cache, since we observe a 1.93× speedup between them. The scientific card tops our speedup chart showing only a 9% speedup over the Geforce RTX 2080 Ti, despite being an older architecture. When factoring in the MSRP of the devices under consideration, the mid-class Geforce GTX 1060 becomes the graphics card that delivers the best price-performance ratio. The scientific card Tesla V100 delivers a worse price-performance than the gaming cards, due to its high MSRP.

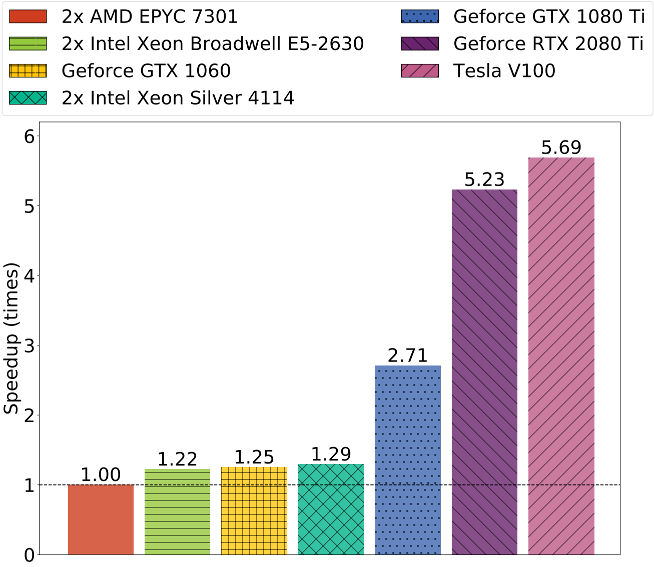


Figure 5: Speedup between the three CPU and the four GPUs under consideration. For each CPU, the performance of a dual-socket server, with the *Search by triplet SPMD* algorithm with their best ISPC configuration is shown. For each GPU, the performance of *Search by triplet* within the GPU sequence framework with their best parameter configuration is shown. CPUs underperform compared to GPUs. The performance scales to higher-end GPU devices.

In order to understand the impact of our work in the field, we can compare the performance obtained with the current *LHCb baseline* implementation [22]. Our SPMD implementation presents a speedup of 1.46× over the LHCb baseline, under the same hardware configuration of a dual-socket Intel Xeon Broadwell E5-2630. The Geforce RTX 2080 Ti presents a speedup of 6.23×, and the Tesla V100 a speedup of 6.77× when compared to the baseline results. We acknowledge the physics quality of the results are not identical between the baseline and our implementation, and that the

LHCb codebase is in active development and its performance has improved since. Nevertheless, we attribute the presented speedup to the combined impact of data structures, locality and vectorization of our algorithm design.

Figure 6 shows a breakdown of the contribution of each algorithm to the overall timing of the Velo track reconstruction. We observe our sequence is dominated by track seeding and track forwarding, as was to be expected from the computational complexity analysis. The weak track filter time fraction is negligible, since it operates in a small subset of leftover 3-hit tracks.

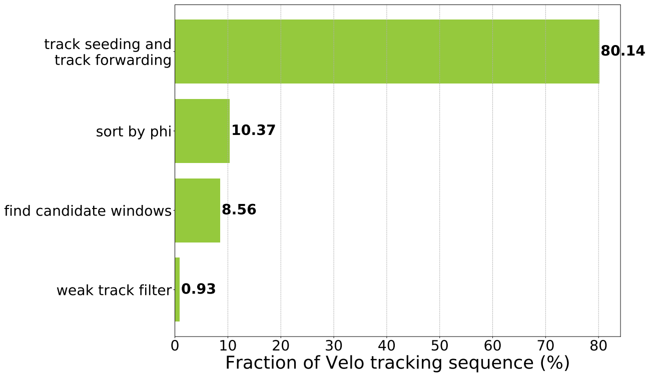


Figure 6: The contribution of each algorithm to the timing of the sequence is shown. Track seeding and track forwarding compose a single *kernel* and are therefore shown together. The track reconstruction sequence is dominated by track seeding and track forwarding, as would be expected from the complexity analysis.

A. GPU sequence results

A percentual comparison of profiled sequence execution and memory transmission data is shown in Figure 7. The sequence execution dominates the time distribution of the GPU. Given that we have created an effective asynchronous pipeline, memory submissions and memory retrievals are hidden behind the execution time of our sequence.

Figure 8 depicts two parameter scans for number of events s and number of streams t, respectively. A configuration of $s = 4096$, $t = 3$ turns out to be effective on all tested hardware. The Geforce GTX 1060 only requires two streams to achieve an effective pipeline. We attribute this to the lower amount of *streaming multiprocessors* on that device, which permits achieving a high occupancy with one stream, hiding the transmissions on the other concurrent stream. A higher number of streams does not increase the throughput. This fact, together with the scaled performance to high-end devices, indicate our software is *compute bound*.

VII. Conclusion

We have presented *Search by triplet*, a new algorithm to efficiently perform track reconstruction on parallel architectures. Our algorithm takes inspiration from track forwarding techniques. We have designed our algorithm removing RAW dependencies and revisiting detector modules in subsequent steps to maximize temporal and spatial locality. We have discussed worst-case complexity for each of its constituent parts. We have developed our algorithm in CUDA and we have optimized the launch parameter configurations. The algorithm has been validated against Monte Carlo simulated data.

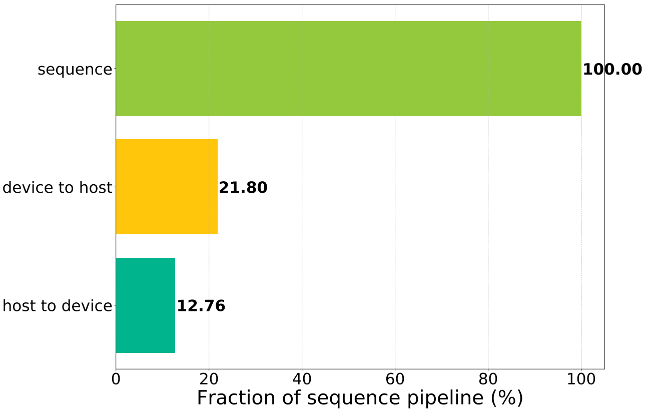


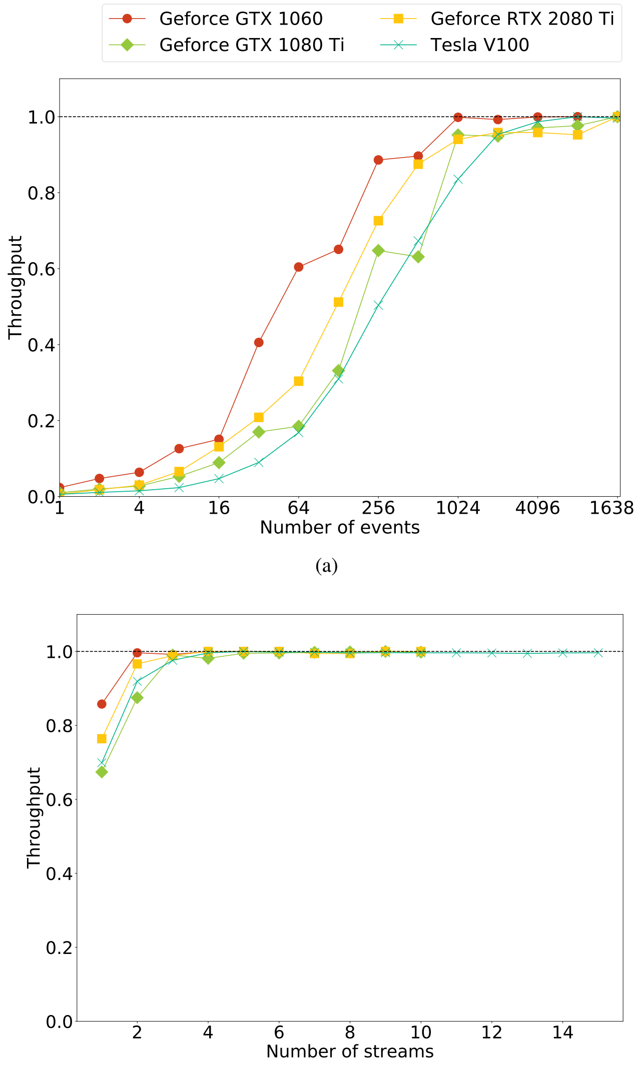
Figure 7: Pipeline of Velo tracking sequence in the GPU sequence framework. The timings of the pipeline were obtained by the *nvprof* command in a full sequence execution. The pipeline is dominated by code execution by a 78.20% margin. The transmissions will be hidden if enough streams are running asynchronously.

We have presented *Search by triplet SPMD*, an SPMD realization of our algorithm geared towards parallel SIMD processors. We have carried over the design of our algorithm to CPUs, and we have optimized our compilation options for each of the processors under consideration.

We have compared the performance of our algorithm across a variety of parallel architectures. Our algorithm benefits from larger vector widths on Intel processors, and scales to high-end GPU architectures. The algorithm performs the Velo track reconstruction with a throughput of 57.36 kHz (AMD EPYC 7301) through 74.17 kHz (Intel Xeon Silver 4114) on CPUs, and 71.75 kHz (Geforce GTX 1060) through 326 kHz (Tesla V100) on the GPUs under consideration.

We have assessed the impact of our algorithm design decisions by comparing the performance of our SPMD implementation with the LHCb baseline implementation. We obtain a 1.46× speedup with respect to the baseline implementation running on the same hardware. We acknowledge this codebase is in active development, and a dedicated study comparing track reconstruction approaches should be pursued.

We have also presented a new framework to perform physics reconstruction on many-core architectures *GPU sequence framework*. We have encapsulated our software into this framework. We have performed a parameter scan over the configurable number of events and streams of our application. An effective pipeline has been created under all studied devices



(a)

(b)

Figure 8: Two parameter scans are shown for the GPU sequence framework application. (a) The number of events parameter is scanned for all devices under consideration. The configuration used throughout all measurements is $t = 3$ and $r = 200$. Performance caps with 1024 events for Geforce cards and 2048 events for the Tesla card. (b) A scan of the number of streams is depicted, with configuration $s = 4096$ and $r = 200$. The Geforce GTX 1060 requires only 2 streams to achieve an effective pipeline, in contrast with the 3 streams required by the other cards. The memory capacity of each device limits the maximum number of concurrent streams under the tested configuration. The peak performances for the Geforce GTX 1060, Geforce GTX 1080 Ti, Geforce RTX 2080 Ti and Tesla V100 are 71.75, 155.33, 299.94 and 326.26 kHz respectively.

that hides transmission times. We have profiled the algorithms that conform our Velo reconstruction implementation, and we have identified the main time consumers. Our framework employs a custom memory manager to allocate and free memory segments as required in an asynchronous manner.

Our track reconstruction algorithm is an indication that other LHCb subdetectors may be amenable to be reconstructed efficiently on many-core architectures. We have shown a translation of our GPU algorithm performs adequately on CPUs, while maintaining the same SIMD-oriented design choices. We will study the applicability of our design to other subdetector-specific geometries and conditions.

Our framework can be extended with additional reconstruction algorithms. We intend to do a detailed cost-analysis of our application for the upcoming LHCb upgrade. The performance of our application will be a determining factor to adopt GPUs in LHCb's trigger server farm.

Acknowledgment

The authors would like to thank D. vom Bruch for fruitful discussions and code reviews. Thanks to V. Gligorov for his guidance and support, and to P. Fernández Declara for framework discussions. Thanks to C. Potterat and M. Rangel for early discussions on the development of the first prototype of Search by triplet. Thanks to D. Rohr for ideas for obtaining better performance, and to R. Quagliani for continuous discussions to improve the physics performance. We would also like to thank the LHCb computing and simulation teams for their support and for producing the simulated LHCb samples used to develop and benchmark our algorithm.

References

[1] The LHCb Collaboration, "LHCb detector performance," *International Journal of Modern Physics A*, vol. 30, no. 07, p. 1530022, mar 2015. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0217751X15300227

[2] ——, "Framework TDR for the LHCb Upgrade: Technical Design Report," Tech. Rep. CERN-LHCC-2012-007. LHCb-TDR-12, Apr. 2012. [Online]. Available: https://cds.cern.ch/record/1443882

[3] ——, "LHCb Trigger and Online Upgrade Technical Design Report," Tech. Rep. CERN-LHCC-2014-016. LHCB-TDR-016, May 2014. [Online]. Available: https://cds.cern.ch/record/1701361

[4] D. Rohr, S. Gorbunov, and V. Lindenstruth, "GPU-accelerated track reconstruction in the ALICE High Level Trigger," *J. Phys. Conf. Ser.*, vol. 898, no. 3, p. 032030, 2017.

[5] P. Sen and V. Singhal, "Event selection for much of cbm experiment using gpu computing," in *2015 Annual IEEE India Conference (INDICON)*, Dec 2015, pp. 1–5.

[6] D. vom Bruch, "Online Data Reduction using Track and Vertex Reconstruction on GPUs for the Mu3e Experiment," *EPJ Web of Confernces*, vol. 150, no. 00013, 2017. [Online]. Available: https://www.epj-conferences.org/articles/epjconf/pdf/2017/19/epjconf_ctdw2017_00013.pdf

[7] The LHCb Collaboration, "LHCb VELO Upgrade Technical Design Report," Tech. Rep. CERN-LHCC-2013-021. LHCB-TDR-013, Nov 2013. [Online]. Available: http://cds.cern.ch/record/1624070

[8] O. Callot, "FastVelo, a fast and efficient pattern recognition package for the Velo," CERN, Geneva, Tech. Rep. LHCb-PUB-2011-001. CERN-LHCb-PUB-2011-001, Jan 2011, lHCb. [Online]. Available: http://cds.cern.ch/record/1322644

[9] D. Funke, T. Hauth, V. Innocente, G. Quast, P. Sanders, and D. Schieferdecker, "Parallel track reconstruction in CMS using the cellular automaton approach," *Journal of Physics: Conference Series*, vol. 513, no. 5, p. 052010, jun 2014. [Online]. Available: http://stacks.iop.org/1742-6596/513/i=5/a=052010?key=crossref.85cff4ebb76ffe912b706a3d23b5f608

[10] R. E. Kálmán, "A New Approach to Linear Filtering and Prediction Problems," *Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, Mar. 1960. [Online]. Available: http://dx.doi.org/10.1115/1.3662552

[11] D. H. Cámpora Pérez and O. Awile, "An efficient low rank kalman filter for modern simd architectures," *Concurrency and Computation: Practice and Experience*, vol. e4483, 2018.

[12] R. H. C. Lopes, I. D. Reid, and P. R. Hobson, "A well-separated pairs decomposition algorithm for k-d trees implemented on multi-core architectures," *Journal of Physics: Conference Series*, vol. 513, no. 5, p. 052011, jun 2014. [Online]. Available: https://doi.org/10.1088%2F1742-6596%2F513%2F5%2F052011

[13] C. Cheshkov, "Fast hough-transform track reconstruction for the alice tpc," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 566, no. 1, pp. 35 – 39, 2006, tIME 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900206008059

[14] L.-B. Niu, Y.-L. Li, M. Huang, B. He, and Y.-J. Li, "Track reconstruction based on Hough-transform for nTPC," *Chinese Physics C*, vol. 38, no. 12, p. 126201, dec 2014. [Online]. Available: http://stacks.iop.org/1674-1137/38/i=12/a=126201?key=crossref.04ab3117f629b4f1118b49f222f1d94c

[15] A. Abba, F. Bedeschi, F. Caponio, R. Cenci, M. Citterio, A. Cusimano, J. Fu, A. Geraci, M. Grizzuti, N. Lusardi, P. Marino, M. Morello, N. Neri, D. Ninci, M. Petruzzo, A. Piucci, G. Punzi, L. Ristori, F. Spinella, S. Stracka, D. Tonelli, and J. Walsh, "An "artificial retina" processor for track reconstruction at the full lhc crossing rate," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 824, pp. 260 – 262, 2016, frontier Detectors for Frontier Physics: Proceedings of the 13th Pisa Meeting on Advanced Detectors. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168900215012607

[16] A. Glazov, I. Kisel, E. Konotopskaya, and G. Ososkov, "Filtering tracks in discrete detectors using a cellular automaton," *Nucl. Instr. and Meth.*, vol. A329, pp. 262–268, 1993.

[17] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[18] Intel, "Intel spmd program compiler." [Online]. Available: https://ispc.github.io/

[19] R. Aaij *et al.*, "Performance of the LHCb Vertex Locator. Performance of the LHCb Vertex Locator," *JINST*, vol. 9, no. CERN-LHCB-DP-2014-001. CERN-LHCB-DP-2014-001. LHCB-DP-2014-001, p. P09007. 61 p, May 2014, comments: 61 pages, 33 figures. [Online]. Available: http://cds.cern.ch/record/1707015

[20] M. Schiller, "Track reconstruction and prompt k_S^0 production at the LHCb experiment," Dissertation,, University of Heidelberg,, 2011.

[21] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485. [Online]. Available: http://doi.acm.org/10.1145/1465482.1465560

[22] M. De Cian, A. Dziurda, V. Gligorov, C. Hasse, W. Hulsbergen, T. E. Latham, S. Ponce, R. Quagliani, H. F. Schreiner, S. B. Stemmle, J. Van Tilburg, M. J. Zdybal, and J. M. Williams, "Status of HLT1 sequence and path towards 30 MHz," CERN, Geneva, Tech. Rep. LHCb-PUB-2018-003. CERN-LHCb-PUB-2018-003, Mar 2018. [Online]. Available: http://cds.cern.ch/record/2309972

B

AN EFFICIENT LOW-RANK KALMAN FILTER FOR MODERN SIMD ARCHITECTURES

An Efficient Low-Rank Kalman Filter for Modern SIMD Architectures

Daniel Hugo Cámpora Pérez[1,2] | **Omar Awile**[1]

[1]CERN, CH-1211 Geneva 23, Geneva, Switzerland
[2]Universidad de Sevilla, C/San Fernando, 4, C.P. 41004, Sevilla, Spain

Correspondence
Daniel Hugo Cámpora Pérez,
CERN, CH-1211 Geneva 23,
Geneva, Switzerland.
Email: dcampora@cern.ch

Present Address
Daniel Hugo Cámpora Pérez,
CERN, CH-1211 Geneva 23,
Geneva, Switzerland.

Summary

The Kalman filter is a fundamental process in the reconstruction of particle collisions in high-energy physics detectors. At the LHCb detector in the Large Hadron Collider this reconstruction happens at an average rate of 30 million times per second. Due to iterative enhancements in the detector's technology, together with the projected removal of the hardware filter, the rate of particles that will need to be processed in software in real-time is expected to increase in the coming years by a factor 40. In order to cope with the projected data rate, processing and filtering software must be adapted to take into account cutting-edge hardware technologies.

We present Cross Kalman, a cross-architecture Kalman filter optimized for low-rank problems and SIMD architectures. We explore multi and many-core architectures, and compare their performance on single and double precision configurations. We show that under the constraints of our mathematical formulation, we saturate the architectures under study. We validate our results and integrate our filter in the LHCb framework.

Our work will allow to better use the available resources at the LHCb experiment and enables us to evaluate other computing platforms for future hardware upgrades. Finally, we expect that the presented algorithm and data structures can be easily adapted to other applications of low-rank Kalman filters.

KEYWORDS:
Kalman filter, data-intensive parallel algorithms, numerical methods

1 | INTRODUCTION

The LHCb detector at CERN will be upgraded in 2020[1] to acquire data at an estimated rate of 30 MHz, requiring to process a data throughput of 40 Tbit/s. At the same time the first stage of filtering in the Data Acquisition process, also known as hardware level trigger, will be discontinued in favor of a full software trigger[2]. Consequently the throughput that the software level trigger will need to sustain in order to maintain a steady triggering rate will dramatically increase, due to both the increase in rate of events processed in software, and the influx of larger events.

To be able to cope with the increased data rate, several hardware architectures are currently under consideration. While the current LHCb software trigger farm is composed solely of Intel Xeon processors, in the last few years many High Performance Computing sites are adopting other alternative hardware architectures, such as ARM 64, IBM Power X, FPGAs, or many-core architectures such as GPGPUs or Intel Xeon Phi. This has raised the question within the High Energy Physics community whether these architectures are also suitable for performing the software trigger in a sustainable way. To answer this question, performance, economical, power consumption and software maintainability aspects need to be taken into account.

In this work we will consider the Kalman filter component used in the LHCb software framework. The Kalman filter is a linear quadratic estimator, first introduced by Rudolf E. Kálmán in 1960[3], that has been extensively used to estimate trajectories in various systems[4 5]. In its discrete implementation[6], it consists in a *predict* stage where the state of the system is projected according to a given model, and an *update* stage where the state is adjusted taking into account a measurement. In particular we consider here a filter that is low-rank.

In LHCb the Kalman filter is applied to estimate particle trajectories (*tracks*) as they travel through the particle detector[7]. Tens of millions of collisions per second occur in the detector, each requiring tens of thousands of filter computations. The Kalman filter is therefore the single largest time contributor in the LHCb software chain, taking about 60% of the first stage software trigger reconstruction time.

In the LHCb experiment, many particles travel through the detector simultaneously and independently. Taking into account the scale of the problem, our Kalman filter can be considered a petascale embarrasingly parallel problem.

In contrast to the work by G. Cerati et al.[8], we do not use our Kalman filter for track finding, but instead, we filter fully built tracks. That allows us to take into account the number of tracks and nodes when envisioning a scheduling strategy, resulting in an effective use of the SIMD capabilities of the processors under study. The execution conditions of our use-case diverge significantly from those in the works by Lin et al.[9] and Lu et al.[10] in that we are presented with a known workload and we do not require to schedule in real time.

Huang et al.[11] show results on a GPU implementation of Kalman filters with an increasing observation dimension. Nevertheless, we are interested in a slightly different implementation in some cases, and our observation dimension is very small compared to theirs, making their solution unsuitable for our use-case.

Here we extend over previously presented results[12][13]. We describe in detail our methodology and the integration of our proto-application into the LHCb framework. We have updated our results, explored new architectures and evaluated their cost-effectiveness. Additionally, we have analyzed the performance impact of using single precision floating-point numbers on a variety of SIMD architectures, including multi and many-core architectures.

We explore performance gains over the current LHCb particle reconstruction software[14], and compare the speedup obtained over a variety of architectures. Additionally, we validate our implementation and integrate it back in the LHCb reconstruction framework, observing a performance gain on existing hardware.

Our work has a direct impact on the current reconstruction process of LHCb, as it improves the throughput of the Kalman filter sequences. It also enables us to evaluate the existing architectures in regards to the upcoming upgrade of the servers, without the requirement of porting a framework composed of millions of lines of code.

2 | CROSS KALMAN

In LHCb track reconstruction a particle trajectory consists of *signal nodes* originating from detector signals. Additionally, virtual *reference nodes* are placed in concrete positions to determine the track parameters up-front for later reconstruction stages. As opposed to signal nodes, reference nodes trigger a prediction with no update in the Kalman filter.

For a given particle trajectory, the Kalman filter is applied twice: First, a fit in the forward direction, positive in the Z axis, is followed by a fit in the backward direction, processing the nodes in reverse order. Afterwards, a smoothed state is calculated averaging both states. The quality of the smoothed states of a track will determine its acceptance in the track reconstruction process.

The calculation of smoothed states requires having processed the forward and backward fit before. For a given particle track, in order to calculate the leftmost smoothed state we would require the first forward state and the last backward state. Furthermore, the rightmost smoothed state would require the last forward state and the first backward state. Thus, this introduces a dependency between the stages with little room for parallelization. However, a particle collision generates many independent particles that can be reconstructed at the same time, allowing us to envision a horizontally parallel scheme.

For either direction, the first encountered signal node does not have any preceding signal data. *Reference parameters* according to their position are generated and fed onto those nodes, and the prediction is applied to these parameters. Figure 1 shows two particles traversing the LHCb detector with various nodes. When performing the forward fit, the top particle carries out three predictions from reference parameters before doing the first update. From that point on, all states are predicted from previous states, however only signal nodes trigger an update. The particle at the bottom performs a single prediction from reference parameters, given the first node is a signal node.

Furthermore, given a node, the resulting state is calculated as the average between its forward updated state and its backward predicted state. However, if the node has no preceding signal node in one of the directions, the smoother copies the updated state of the other direction.

Given this problem formulation, we describe the design of our algorithm in the following parts: the control flow, the data structures and an efficient implementation for performing the math computations.

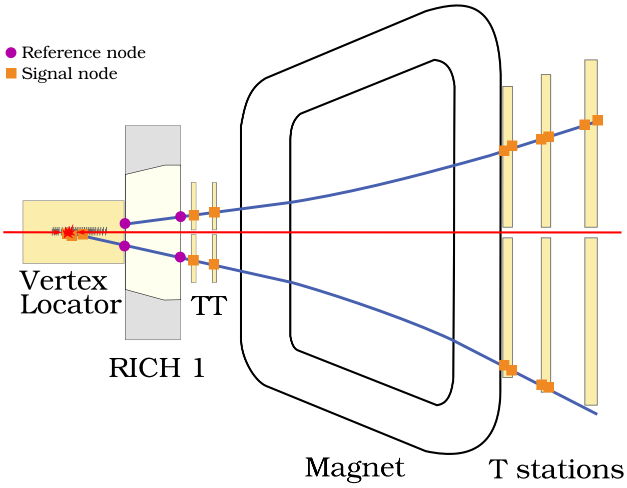


FIGURE 1 Schematic of two particles (blue) traversing LHCb subdetectors. A particle collision is indicated by the two red arrows meeting in the center of the *Vertex Locator* subdetector. Particles produced from the collision traverse tracking subdetectors; here the Vertex Locator, TT and T1, T2 and T3 stations are depicted. A magnet bends the trajectory of produced particles according to their momentum and charge.

2.1 | Control Flow

Since the control path of processing a particle trajectory diverges depending on the nature of its nodes, we have divided each particle trajectory in three stages: `pre`, `main` and `post` (see Figure 2 for an overview). `pre` is the forward trajectory from the first node until a signal node is encountered, inclusive. Similarly, `post` is the backward trajectory from the last node until a signal node is encountered, inclusive. Finally, `main` includes the remaining nodes. The forward fit processing logic differs between `pre` and `main`, while for the backward fit processing logic differs between `post` and `main`.

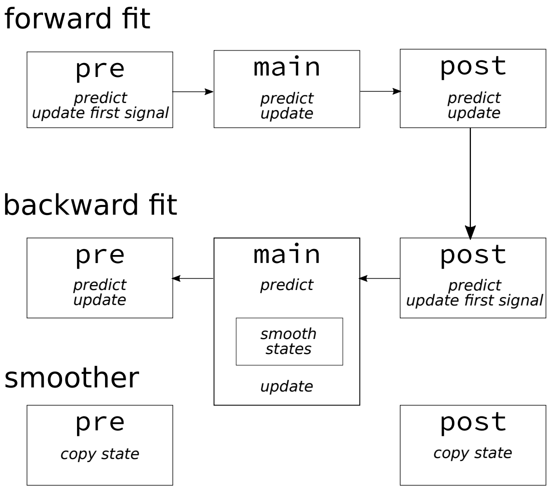


FIGURE 2 Flowchart of our application. Tracks are divided in three stages, according to the location of reference and signal hits. In order to process all tracks, three static schedules are generated, `pre`, `main` and `post`. The dependencies between stages are shown, alongside the subprocesses on each stage.

A single particle trajectory would be processed as follows: First, pre, main and post stages would be identified. Then, all nodes in the pre stage would be processed in order, followed by main and post. The first node in the backward direction requires a starting state that can optionally be fed from the last forward state, hence the optional requirement that backward processing start after forward processing. The nodes would then be processed in reverse order, starting from the post stage, followed by main and pre. Finally, the smoothed states can be calculated in no particular order.

In order to fully exploit the capabilities of SIMD architectures, we employ a static scheduler that assigns node calculations to SIMD lanes. Since the execution of nodes from different particles is independent, we execute them in a horizontally parallel (data parallel) scheme. In order to minimize branches and guarantee instruction locality, we generate three such schedules, one for each stage.

The amount of nodes processable at a time depends directly on the SIMD width of the processor. Hence our scheduler accepts a configurable vector width. It is also able to detect at compile time the supported vector width of the platform. There are no restrictions on the width of the lane, allowing this design to also target many-core architectures, where wider vector units are available.

More formally, given m particle trajectories with n_i nodes each and k processors, we want to assign nodes to processors minimizing the number of compute iterations. This problem is a variant of the number partitioning problem Npp[15], which is known to be NP-complete. Our scheduling algorithm orders the trajectories in descending order of nodes, and assigns nodes to processors following a Decreasing-Time Algorithm (DTA).

The same schedule can be used for the forward fit, the backward fit, and the smoother. The forward and backward dependencies between node calculations are naturally resolved by traversing the schedule in the respective direction (cf. Figure 2). All tracks are processed on each stage prior to processing the next one. The smoother `pre` and `post` stages are processed after completion of the backward fit.

In our implementation we place particular emphasis on avoiding as much as possible memory copy operations and exploiting memory locality. We reuse data structures throughout the schedule iterations replacing only necessary data portions when required to do so.

Additionally, the data structures must be aligned and refer relatively to the same nodes in order for the smoother to be able to produce an average state from the previously calculated forward and backward states. Using our scheduler this requirement is trivially met.

Figure 3 depicts 10 iterations computed with the described scheduler, with vector width set to four. The last column denotes the particle-node being processed at the moment. This schedule can be iterated forwards or backwards, maintaining the sequentiality enforced by the Kalman filter process. Due to its flexible design, execution can be optimized on multi and many-core SIMD architectures, with varying vector widths.

```
it    in   out  act  vector (#particle-#node)
#540: 0000 0001 1111 { 112-9 80-11 81-11 113-10 }
#541: 0001 1110 1111 { 112-10 80-12 81-12 79-3 }
#542: 1110 0000 1111 { 107-2 109-1 108-2 79-4 }
#543: 0000 0000 1111 { 107-3 109-2 108-3 79-5 }
#544: 0000 0000 1111 { 107-4 109-3 108-4 79-6 }
#545: 0000 0000 1111 { 107-5 109-4 108-5 79-7 }
#546: 0000 0000 1111 { 107-6 109-5 108-6 79-8 }
#547: 0000 0000 1111 { 107-7 109-6 108-7 79-9 }
#548: 0000 0000 1111 { 107-8 109-7 108-8 79-10 }
#549: 0000 0000 1111 { 107-9 109-8 108-9 79-11 }
```

FIGURE 3 Main scheduler iterations. The first column shows the iteration number. The second and third show the input and output mask, used to notify a change of particle. The fourth column is the action mask. The last column shows the nodes being processed in parallel. The scheduler was run with a vector width of four.

2.2 | Data Structures

The algorithm's main data structure is composed of three parts. A hardware-specific data backend stores data contiguously and aligned to the required SIMD width, and provides chunks of requested data agnostic to their contents. In order to avoid a performance impact of memory allocations of big chunks of contiguous space, data backends are created on demand and can store a configurable number of elements. Iterators point to the data backends and are configured with a structure size. We provide forward and reverse iterators in order to traverse the data as required.

We use Arrays of Structures of Arrays (AOSOA) as data views over the data backends. This kind of data structures benefit from locality when accessing any of their elements, and have been shown to work well with SIMD processors[16]. Further locality is preserved by storing these structures next to each other contiguously. Figure 4 depicts one such data view used for calculating the fits. For each node, it consists of a state composed of a parametric description of the trajectory (x, y, tx, ty) and its charge over its momentum $\frac{q}{p}$, a covariance matrix and a χ^2 goodness of fit value.

For a single particle position, the state is a five-element vector $\begin{pmatrix} x & y & tx & ty & \frac{q}{p} \end{pmatrix}$, while the covariance σ is a 15-element matrix (5×5 symmetric matrix), and the χ^2 is a single scalar value. This structure is used to calculate the predicted and updated state. We store the resulting updated state of each node, and therefore such an AOSOA structure is generated for each node and fit direction. In order to perform the smoother we require the forward updated and backward predicted states, and hence this is computed inline prior to overwriting it with the updated state.

$$\begin{vmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ tx_0 & tx_1 & tx_2 & tx_3 \\ ty_0 & ty_1 & ty_2 & ty_3 \\ \frac{q}{p}_0 & \frac{q}{p}_1 & \frac{q}{p}_2 & \frac{q}{p}_3 \\ \sigma_{0,0} & \sigma_{1,0} & \sigma_{2,0} & \sigma_{3,0} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma_{0,14} & \sigma_{1,14} & \sigma_{2,14} & \sigma_{3,14} \\ {\chi^2}_0 & {\chi^2}_1 & {\chi^2}_2 & {\chi^2}_3 \end{vmatrix}$$

FIGURE 4 AOSOA data structure for scheduler with width four.

2.3 | Efficient Vector Implementation

In order to test the proposed algorithm and data structures in section 2 we have implemented the core routines of the fit and smoother algorithms in a proto-application, *Cross Kalman*. By implementing the algorithm outside LHCb's analysis framework, Gaudi [17], we were free to test a number of choices both in programming model and SIMD framework. We focus particularly on manual vectorization with the help of vector intrinsics libraries, namely VCL [18], UMESIMD [19]. For parallelizing across CPU cores we use the *Thread Building Blocks* (TBB) library. Based on our initial implementation we have performed several iterations of fine-grained code optimizations . We tested several formulations, unrolling loops, inlining functions, changing compiler options and reordering of the code. Furthermore, we implemented the arithmetic backend of our application in an even more compact synthetic benchmark, *Cross Kalman Mathtest*. This allowed us to port our implementation to the language extensions OpenCL and CUDA allowing us to compare performance and scalability across a large number of hardware platforms (cf. section 3). Finally, we provide a scalar implementation as fall back, which can run on architectures not supporting vectorization.

3 | RESULTS

Using the *Cross Kalman*[1] and *Cross Kalman Mathtest*[2] applications we are able to benchmark our proposed algorithm's performance and parallel scalability on a variaty of hardware platforms including traditional multi-core platforms, many-core processors as well as GPGPUs. In the following we outline the conditions under which we have performed the various computer experiments and benchmarks presented below:

- All our benchmark applications except for the CUDA and OpenCL kernels were compiled with the `gcc 6.x` compiler and `-O2 -march=native` compiler options.
- All the systems under test were equipped with DDR4 RAM. The Intel Xeon Phi platform is also equipped with MCDRAM.
- Version v1.3.1 of Cross Kalman was used for obtaining the results shown here.
- For Cross Kalman, we generated synthetic events out of an unbiased sample of 72 events generated using LHCb's Monte Carlo simulator. Each event is scheduled to be processed by a TBB thread, emulating the behavior of the full application.
- For Cross Kalman Mathtest, we generated 2^{23} synthetic events, and cross-checked the result across architectures with a checksum.

[1] https://gitlab.cern.ch/dcampora/cross_kalman

[2] https://gitlab.cern.ch/dcampora/cross_kalman_mathtest

- On Non-Uniform Memory Access (NUMA) platforms we spawn one process per domain with as many TBB threads as logical cores pinned to its memory.
- Using the simulated data we were able to validate our results against the results from the original algorithm as implemented in the LHCb production code.
- We performed all benchmarks in both double and single precision.
- All hardware platforms were configured to be in *performance* power mode and Turbo Boost available when possible.
- The Intel Xeon Phi Knights Landing platform was run in quadrant and flat memory mode, pinning all threads to MCDRAM.
- The figure of merit is the average throughput, calculated as $\#\mathrm{fits}/\mathrm{time}$.

3.1 | Cross Kalman results

Figure 5 shows a comparison between the architectures under study for the Cross Kalman use case. The leftmost bar in (a) shows the performance of the scalar implementation of the fit, obtained from the timings reported by the framework on a representative node of LHCb's computing farm. Our Cross Kalman implementation outperforms the scalar implementation on the same hardware platform by a factor of 3.03x. ThunderX shows the poorest performance of the architectures under study. Even though a speedup of 1.75x over the scalar implementation on E5-2630 v3 is observed, this is only due to optimizations in the software. When both architectures run Cross Kalman, the E5-2630 v3 outperforms ThunderX by 1.73x. This is likely due to a comparatively lower peak DRAM bandwidth and peak floating point performance on ThunderX.

The transition from Haswell (v3) to Broadwell (v4) makes the E5-2630 processor 1.20x faster. We believe this is the combined effect of an increase in the core count from 8 to 10, a proportional increase in cache size and an increase in streaming memory bandwidth, that we measured in 52.0 to 55.3 GB/s. As we will see in the roofline model in Figure 9 our application is memory bound, and therefore these changes have a direct impact in the performance we observe. Accordingly, the Xeon Phi processor outperforms all the other processors when configured with the high bandwidth MCDRAM.

A price performance plot is shown in (b). Intel Xeon Phi outperforms the rest of the competitors, rendering it the most competitive from a price / throughput standpoint. It is 33% cheaper than our Broadwell system, and 27% cheaper than AMD's EPYC 7351 for the Cross Kalman use case.

A throughput scaling plot for all architectures is shown in Figure 6 a. The processor that shows less performance degradation up to using all of its cores is ThunderX. On the IBM Power8 architecture, depicted in yellow, we are able to scale linearly while no Simultaneous MultiThreads (SMTs) are being used. Using 2 SMTs per processor, a performance improvement of 32% is observed. Moving from 2 to 4, a further 15% is gained, while moving from 4 to 8 no performance benefit is observed.

On the Intel architectures we observe an almost linear scaling until we reach the limit of physical cores. The Intel Xeon Phi processor shows a 27% gain from using 2 HyperThreads, and a further 9% from using 4. We do not obtain any gain from HyperThreads on other Intel processors, which we attribute to the higher bandwidth of MCDRAM on Intel Xeon Phi.

Figures 6 b and 6 c show speedup and parallel efficiency graphs, respectively. All Xeon processors diverge from perfect scaling before the other processors under study. Xeon Phi and ThunderX show performance gains using all of their available processors, with a speedup of 74.98x and 64.88x respectively. For PowerNV, its optimal configuration is reached when configured with 96 processors (24.44x), where the performance flattens out. Enabling HTs on the Xeon processors generates overhead and degrades performance. As expected on all tested hardware platforms, parallel efficiency is significantly degraded when using SMT. PowerNV shows a parallel efficiency of 1.0 until it starts using additional SMTs. We observe a similarly abrupt decrease in parallel efficiency in Xeon Phi when using additional HTs. The Xeon processors efficiency drop even without HTs. With all their physical cores active, we see 40-45% efficiency, which could be due to the memory requirements of the application.

3.2 | Synthetic benchmark results

Figure 7 shows the throughput of the fit and smoother as the vector width is increased. In order to obtain the results of these figures, we used our synthetic benchmark, that allows us to execute the bulk of the computation of the application in a portable and generic way. The tests were compiled against the UMESIMD library. This library conveniently allows for scalar emulated vectorized execution, although with a performance penalty.

In the double precision fit, we observe a 1.44x speedup when increasing the vector width from 128 to 256 bits, and 1.43x from 256 to 512 bits. On the other hand, the double precision smoother experiences 1.46x and 1.61x respectively. We attribute the better scaling of the smoother to its

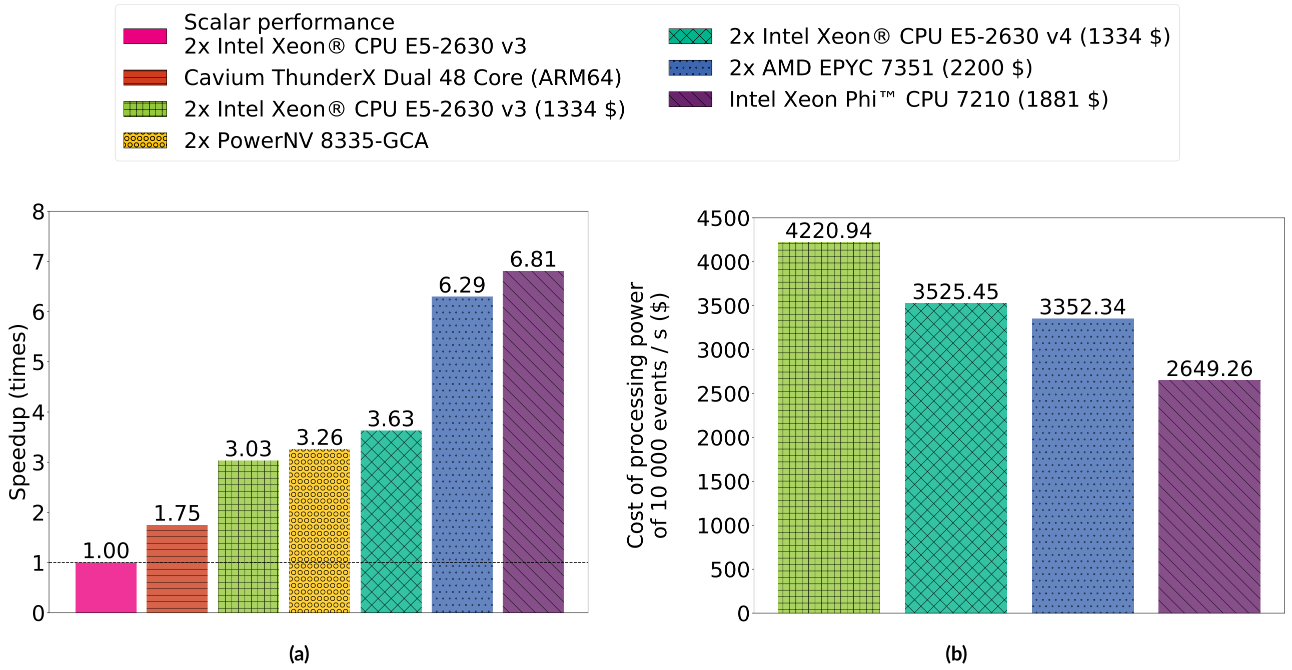


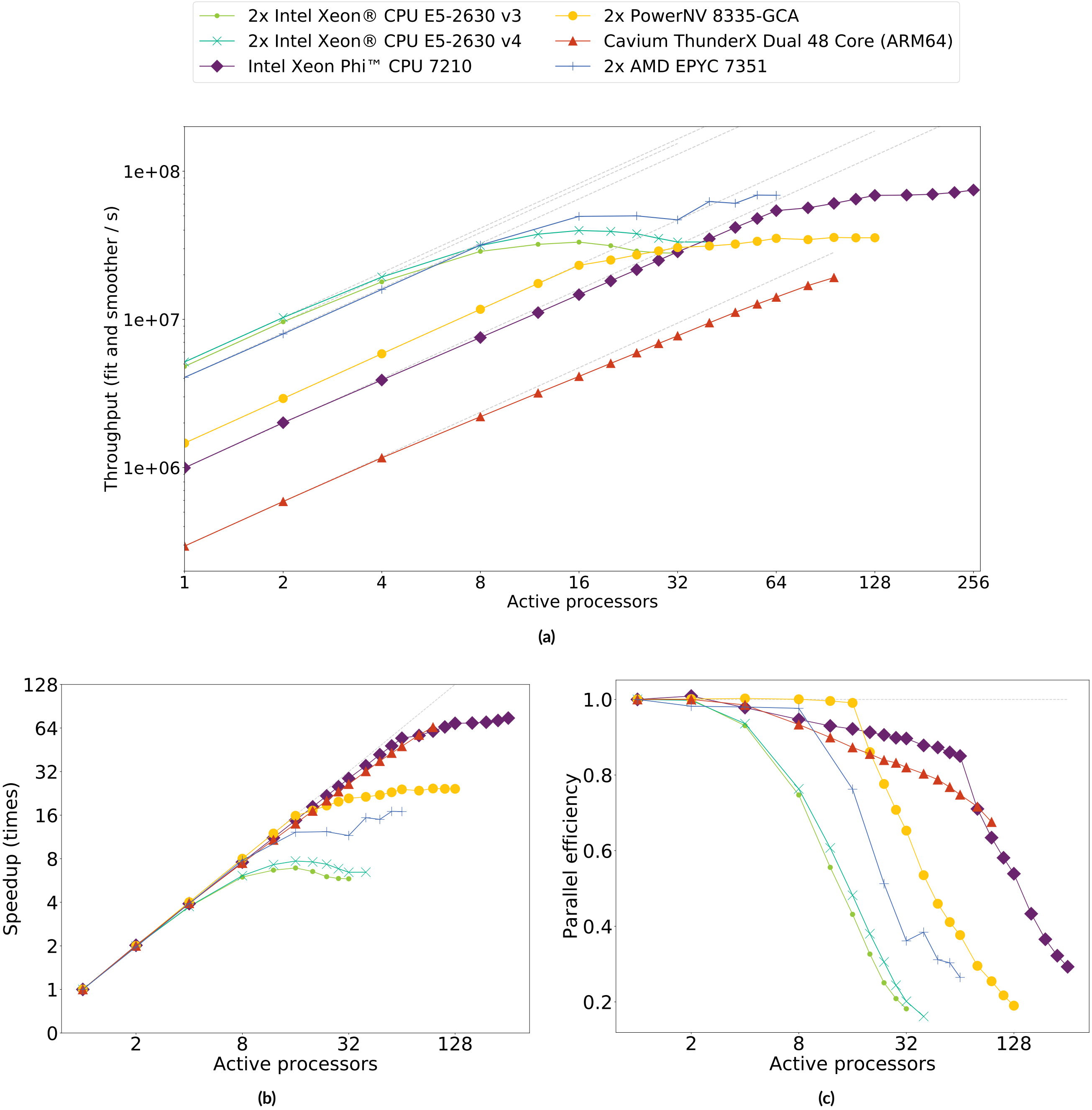
FIGURE 5 (a) Performance of Cross Kalman against the scalar implementation of the fit across several architectures (higher is better). The best configuration for each architecture in terms of active threads is used, shown in Figure 6 . The leftmost bar represents the scalar implementation, running on a Xeon E5-2630 v3 processor. When running Cross Kalman on the same processor, we observe a 3.03x speedup. (b) Price required to obtain a throughput of 10 000 events per second (lower is better). The Intel processors and AMD EPYC 7351 pricing were obtained from https://ark.intel.com and https://www.theregister.co.uk/2017/06/20/amd_epyc_launch/ respectively, in September, 2017. The Xeon Phi processor shows a better price for throughput ratio than its competitors.

higher arithmetic intensity. We observe the same scaling for single and double precision. Single precision produces a deviation from the expected results in 1% of the experiments, deeming it unacceptable to current physics standards.

A cross-architecture performance comparison of Mathtest is shown in Figure 8 . Panel (a) shows the speedup of the double precision tests. The improvement we observe from Haswell to Broadwell is 11%, instead of the 20% we observed in Cross Kalman. We attribute this to the fact tests do not use cache, as opposed to Cross Kalman. Many-core architectures outperform all the other architectures under study, and the two best performing architectures have access to high bandwidth memory MCDRAM and HBM2 respectively.

Panel (b) factors in the price of the processors. Out of the multi-core architectures under study, AMD EPYC obtains the best price per throughput ratio, at 87% of the price of our Broadwell processor. Consumer-grade graphics cards are significantly cheaper than other architectures; Radeon R9 Nano obtains the same throughput as the Broadwell processor for 12% of the price, or 37% of the price of a Xeon Phi.

A single precision Mathtest speedup plot is shown in (c). The speedup across Intel architectures is the same as in double precision. The comparative performance of Tesla is lower than in double precision, whereas for the consumer-grade graphics cards it is higher. The ratio of double precision floating point units in Tesla is higher than in the other graphics cards, which explains this divergence. The performance speedup across Tesla and Radeon RX Vega is 1.43x, similar to the ratio between the memory bandwidths of both architectures: $720.9/483.8 = 1.49x$. We factor in the price of the architectures in the single precision case in (d). The performance benefit from moving to single precision was 2.15x on the Intel architectures, 2.29x on TITAN X, 2.40x on Radeon R9 Nano, 2.30x on Radeon RX Vega and 1.88x on Tesla. Therefore, the most economical architectures remain the consumer-grade graphics cards. Nevertheless, we do not consider the transfer time in this comparisons, in the same way we do not take into account the time to transpose the data into Structure of Arrays (SOA) for all architectures. In order to do such measurements, one would have to implement the Cross Kalman filter in Gaudi for all architectures, and explore different pipelining methods for transferring the data to the graphics cards, which is out of the scope of the Mathtest software. We discuss the integration of Cross Kalman in Gaudi for x86 architectures in section 4.



(a)

(b)

(c)

FIGURE 6 (a) Throughput of Cross Kalman across various architectures. For each architecture, an increasing number of processors is enabled. Additional SMTs are only used on high core counts, when all physical processors have already been enabled. Scaling on ThunderX, PowerNV and Xeon Phi is close to linear, and we observe little gain from enabling SMT. The remaining Xeon processors have a higher performance for lower core counts, but show no performance gain when activating HyperThreads. (b) Speedup against number of active processors. (c) Parallel efficiency against active processors. The PowerNV processors shows no performance degradation using all its physical cores. In contrast, Xeon Phi shows a parallel efficiency of 85% (64 processors), ThunderX 68% (96 processors), E5-2630 v3 43% (16 processors), E7-8890 v3 40% (72 processors) and E5-2683 v4 45% (32 processors).

Figure 9 shows a Roofline model[20] of the Cross Kalman Mathtest synthetic benchmark. The Roofline model relates the arithmetic intensity of an algorithm (or specific implementation thereof) against its floating point performance. The arithmetic intensity is the ratio of floating point

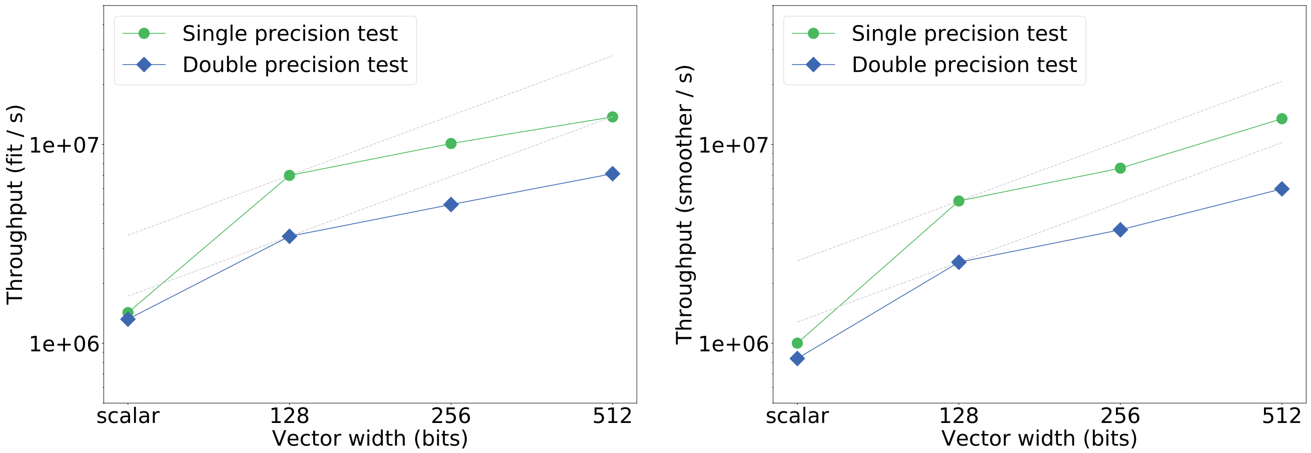


FIGURE 7 Throughput of program as vector width increases, for single and double precision, under Intel Xeon Phi 7210. Left: fit throughput. Right: smoother throughput. The smoother scales better than the fit for wider vector units, due to its higher arithmetic intensity.

operations (FLOPs) executed per bytes transferred from or to main memory. The Roofline plot also shows the peak floating point performance and memory bandwidth of the hardware platform on which the measurements were performed, bounding the maximum attainable performance for a given program. A higher arithmetic intensity means that the program under study is able to take better advantage of the platform's peak performance, ideally avoiding memory bottlenecks.

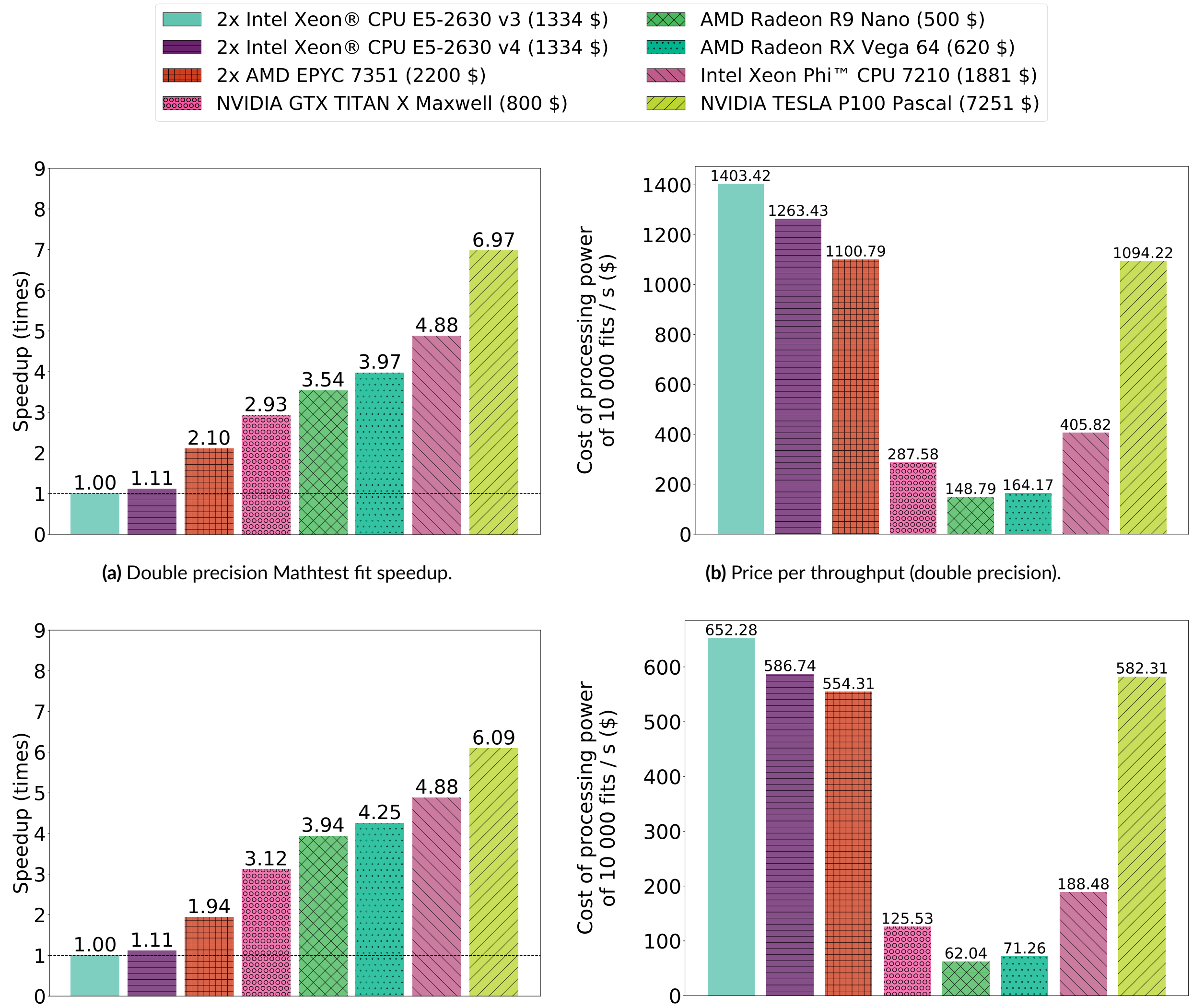
Platform rooflines can either depict theoretical peak bandwidths and floating point performances or actual sustained performance bounds. In Figure 9 the horizontal upper lines depict the *theoretical* peak floating point performances. The diagonal roof lines, however, are Triad streaming bandwidths, offering a tighter and more realistic bound for memory-bound applications. We ran the Roofline benchmarks with 2^{23} synthetic events. A high number of experiments is required in order to avoid data being cached from its generation to its execution, which would affect the effective arithmetic intensity of the application. Nevertheless we notice a small additional performance gain particularly on the Xeon Phi and EPYC platforms, which we attribute to data caching. This can be seen in Figures 9 a and 9 b where the benchmark performance is slightly above the memory bandwidth roofline. We obtain the total FLOPS by counting executed floating point instruction using Intel SDE and NVIDIA `nvprof` for the respective CPU/GPU platforms. Since accurate memory transfer counts are difficult to measure we count the number of bytes read or written by code inspection. These two figures allow us to calculate the arithmetic intensity, and together with accurate time measurements, the performance of the Cross Kalman core algorithm as implemented in the Mathtest benchmark. As can be seen in Figure 9 the algorithm's performance is on all but one of the tested hardware platforms in the arithmetic-intensity regime limited by memory bandwidth and not peak floating point performance. The notable exception is seen in Figure 9 d where the low peak double precision FLOP performance (190 GFLOP/s) and high memory bandwidth (336.5 GB/s) of the NVIDIA GTX TITAN X Maxwell platform make it a particularly easy to achieve peak performance even at low arithmetic intensities.

4 | INTEGRATION

Integrating the Cross Kalman proto-application into the LHCb framework codebase requires a number of code changes and in some cases a redesign of data structures. In addition to the existing Kalman filter implementation *TrackMasterFitter* (TMF), we have created the *TrackVectorFitter* (TVF) package. As the LHCb framework currently only supports the x86 architecture family, our framework implementation targets specifically this architecture. Naturally we still take advantage of the CPUs SIMD extensions when available.

In Cross Kalman, the input and output data are assumed to be in the SOA layout. In the LHCb framework, however, we can not rely on this assumption. In fact, in its current current version data is still mostly stored in Array of Structures (AOS) layout. Hence, TVF includes not only the Kalman filter and smoother processes, but also the generation of the data structures for each Kalman filter step. A detailed performance analysis revealed that data generation and transformation has a significant impact on overall performance. Therefore, data layout choices must be carefully considered.

In order to avoid data copies in TVF, we employ the static schedulers prior to the data generation. This process initializes all the data pointers, permitting the data to be constructed in-place. Nevertheless, the current framework implementation will still create a temporary copy of the



(a) Double precision Mathtest fit speedup.

(b) Price per throughput (double precision).

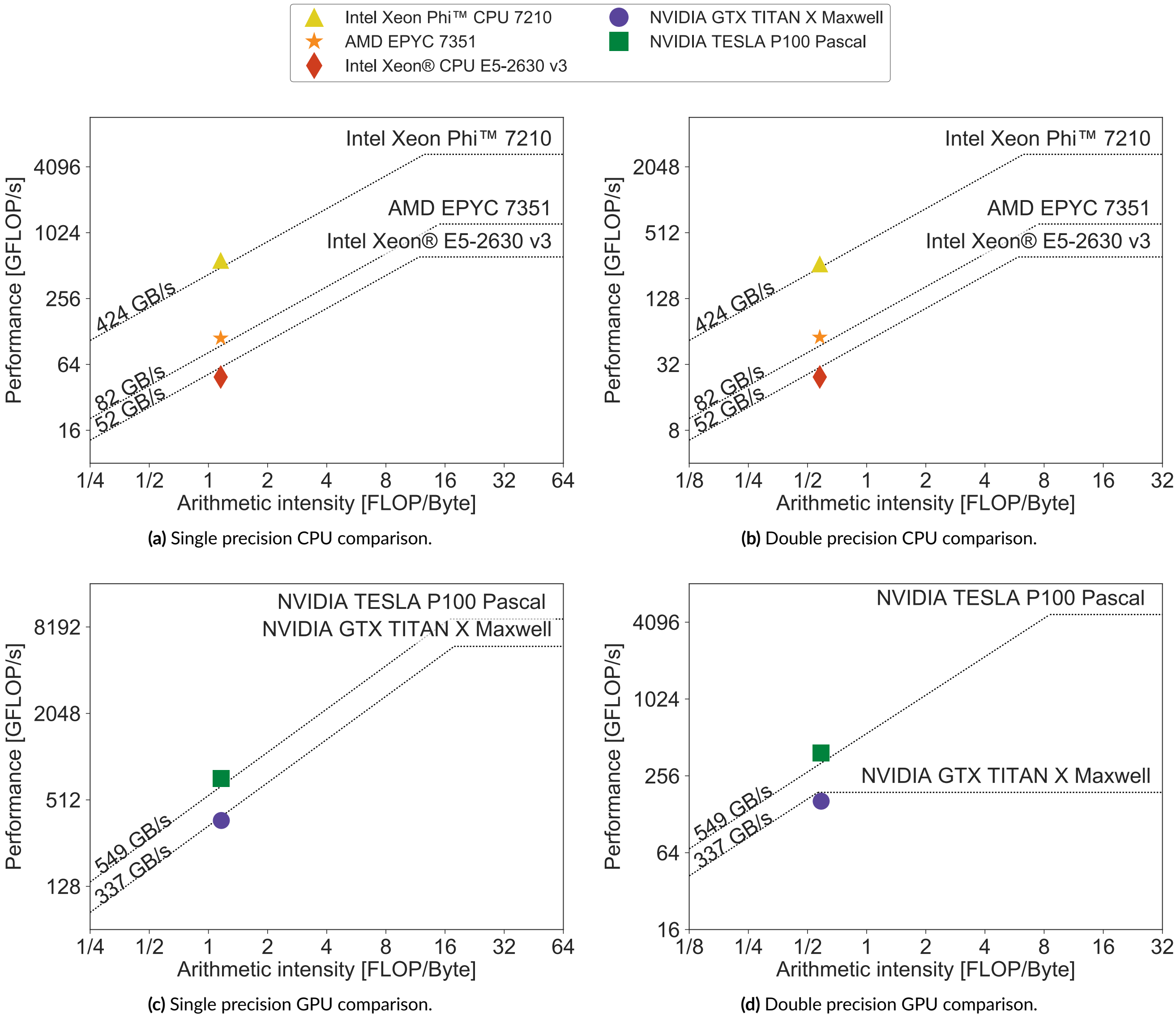
(c) Single precision Mathtest fit speedup.

(d) Price per throughput (single precision).

FIGURE 8 Mathtest synthetic benchmarks across architectures, in single and double precision. All tests are validated, and the configuration saturating each architecture is shown here. (a) Speedup across architectures of double precision Mathtest fit (higher is better). Many-core architectures show an improvement over multi-core architectures. Both Intel Xeon Phi and NVIDIA Pascal benefit from a higher memory bandwidth over other architectures. (b) Price per performance for the double precision tests (lower is better). Consumer-grade accelerators show the best price per performance ratio. (c) (d) Speedup and price per performance for single precision tests. Intel architectures show the same behaviour as in double precision. Consumer-grade graphics cards are comparatively better in single precision than in double precision. We observe 2.15x on the Intel architectures, and up to 2.40x on the AMD Radeon R9 Nano, when moving to single precision. The Intel processors and graphics cards pricing estimates were obtained from https://ark.intel.com and https://amazon.com respectively.

generated data in AOS. Additionally, the framework expects the data to be in an AOS layout at the end of TVF and must be therefore retransposed. More work in the LHCb framework on the Node structure may be required to remove the existing inefficient data layout.

The projection and noise matrices are the only two exceptions to the in-place generation of data structures in TVF. In the forward filter, the calculation of node k requires the projection and noise matrix from the *same* node k. At the same time in the backward filter, the calculation of node k requires the projection and noise matrix from the *previous* node $k + 1$ (recall that indexing in the backward filter is done in reverse). Given that tracks have to be processed in stages, fulfilling these two conditions cannot always be guaranteed. Figure 10 depicts this problem. If projection matrices are generated in-place, pre iteration 1 of the backward filter cannot fetch the previous data without swapping, as it would require an iteration containing nodes *0-2 1-2*.



(a) Single precision CPU comparison.

(b) Double precision CPU comparison.

(c) Single precision GPU comparison.

(d) Double precision GPU comparison.

FIGURE 9 Roofline models of various CPU and GPU platforms and performances of the synthetic Kalman filter benchmark on the respective systems. Panels (a) and (b) show arithmetic intensity and performance of the core algorithm on Intel Xeon E5-2630 v3, Intel Xeon-Phi 7210 (KNL) and the AMD EPYC 7351 platform, allowing a performance comparison between LHCb's current production platform and two contemporary high core-count CPUs. Panels (c) and (d) compare an NVIDIA high-end consumer GPU (TITAN X) with one of NVIDIA's HPC oriented GPU models (P100). We note that our Kalman filter implementation exhibits an arithmetic intensity well below the memory bandwidth / peak floating-point performance sweet spot on all tested platforms. This renders its performance memory-bandwidth limited.

In order to deal with these exceptions, projection and noise matrices are kept as AOS and transposed upon request on each iteration. This process was found to be more efficient than generating SOAs and swapping.

5 | VALIDATION

We have employed several techniques in order to validate the results generated by the Cross Kalman algorithms. The synthetic benchmark Cross Kalman Mathtest calculates a checksum of all executed fits, which are validated against a precomputed sequential implementation. Furthermore we have developed a binary input datatype and an extractor for the LHCb software. The extractor can be executed alongside the original LHCb

```
it    Pre scheduler vector     Main scheduler vector
#0:   { 0-0 1-0 }              { 0-4 1-2 }
#1:   { 0-1 1-1 }              { 0-5 1-3 }
#2:   { 0-2 2-0 }              { 0-6 2-2 }
#3:   { 0-3 2-1 }              { 0-7 2-3 }
```

FIGURE 10 Iterations of Pre and Main schedulers illustrating the need for a projection and noise matrix swap. For each iteration in Pre and Main, two nodes are processed at a time, as the schedule would be generated with SSE and double precision. A scheduled vector is written in the notation *particle number - node number*, the same used in Figure 3 . The backward fit in pre iteration 1 requires data from the previous nodes, that is, *0-2 1-2*. However, no such configuration was generated, thus requiring data swapping. A similar situation can be observed in the backward fit pre iteration 3.

Kalman filter program TrackMasterFitter, to extract and store input data and result from the production filter execution. Cross Kalman is able to read this data as input and use it to cross-check the generated result with the expected result.

Our framework implementation of Cross Kalman, TrackVectorFitter, is already available to LHCb users and serves as the foundation for the numerical results described in this section. We have validated the physics performance of TVF against the original implementation under the current LHCb run conditions, and also under the foreseen conditions of the upgrade. In the current scheme, the number of collisions per *bunch crossing* in the LHCb detector is less than 2, whereas in the upgrade it is proposed to increase to more than 7. This will increase the number of tracks to reconstruct and the throughput required in the filter roughly by a factor 5.

The LHCb experiment uses Monte Carlo simulation to generate validation data sets. Particle collisions and their interaction with the detector are simulated. This simulation generates a data set that can be processed by the LHCb reconstruction software. Finally the reconstructed particles are compared to the Monte Carlo generated ground truth.

Track reconstruction validation is done using three metrics [21]. The *reconstruction efficiency* compares the reconstructed tracks to the expected tracks reported by the Monte Carlo truth. The *clone rate* reports how many track equivalent track pairs were found. The *ghost rate* reports how many tracks were reconstructed with nodes belonging to different particles or noise. Finally, tracks are categorized by their physical properties and category statistics are compared to statistics from the ground truth.

Comparing the Cross Kalman implementation TVF to the original track filter TMF we observe an identical reconstruction efficiency, clone rate and ghost rate under all tested scenarios. While the reconstruction of the track itself does not depend on the fit, the final track χ^2 is used in the different categories as a track quality cutoff. Hence, the identical reconstruction efficiency between the two algorithms validates TVF for its physical properties.

A second test on a smaller sample has been performed, consisting in a track-by-track comparison of physical properties and fit results, such as the χ^2 of the final tracks. It shows a perfect match between TMF and TVF.

We have checked the performance of TVF against TMF under various scenarios. Table 1 shows comparative execution times for LHCb nightly tests. These tests are representative of the conditions under which the LHCb reconstruction runs in the production environment.

Test name	TMF (AVX)	TVF SSE2	TVF AVX2+FMA	Overall reconstruction speedup
magup2016	13.518	12.817	11.504	1.09x
baseline-upgrade	93.713	93.839	91.014	1.03x
sim08	8.307	8.134	7.986	1.02x

TABLE 1 LHCb test times in seconds. All tests are run on a single core of an Intel Xeon E5-2650 v3. All timings refer to the algorithm *TrackBestTrackCreator*, configured with different filter settings. TMF (TrackMasterFitter) is the original filter implementation. Internally, it executes a vertically vectorized code optimized for AVX on this setup. TVF (TrackVectorFitter) refers to our implementation, compiled with either the SSE2 extension (default setting for x86_64) or AVX2+FMA. The *overall reconstruction speedup* refers to the entire reconstruction time of the test, compared between TMF and TVF AVX2+FMA.

It is worth noting that the original implementation TMF has specializations of its Kalman filter code targeting concrete vector extensions. In the machine under analysis the program chooses the AVX implementation, as it is supported by the E5-2650 v3 processor.

We observe a varying performance depending on the test under execution. magup2016 shows gains of up to 9% in the overall reconstruction time, whereas baseline-upgrade and sim08 gains in TVF do not seem to impact much the overall performance. In the case of baseline-upgrade, we believe this is due to the configuration of such test. It uses a *full geometry* setting in its current form, which dominates the time distribution of the fit. We expect its performance to improve in the future.

6 | CONCLUSIONS AND OUTLOOK

In this work we have presented Cross Kalman, an algorithm that is able to efficiently perform low-rank Kalman filters. Cross Kalman is particularly optimized for the LHCb particle tracking use case, but the presented algorithms and data structures can be applied to other situations where a large number of low-rank Kalman filters are used. Using this algorithm we were able to obtain up to 3x speedup over the previous scalar solution on the same hardware platform. Our implementation is flexible enough to accommodate for any kind of SIMD architecture and we have tested it a wide array of architectures. The choice of the Decreasing-Time Algorithm as a scheduling algorithm should be revisited, and we intend to explore other heuristics in the future. Our data structures allow us to efficiently perform the Kalman filter and smoother of many independent particles in parallel. Given the specific nature of our problem instances, it may be possible to reuse data structures across different particle trajectories, and further decrease the memory footprint of our application.

In addition, we have showed that single precision performance scales similarly to its double precision counterpart. An in-depth analysis of the precision requirements and numerical stability of the algorithm, taking into account also the possibility of alternative mathematical formulations, should be carried out. We expect that moving to single-precision and thus doubling the arithmetic intensity of our algorithms will significantly improve performance. Our software is validated and has been integrated in the LHCb codebase under the name TrackVectorFitter, making the overall reconstruction up to 9% faster for certain datasets.

We have verified that our implementation is able to scale to full hardware nodes and is able to adapt to the architectures under study. As expected enabling SMT does not yield further performance improvements with the notable exception of Intel Xeon Phi, which could be due to its higher memory throughput. This architecture stands out as a candidate for joining the computing infrastructure of LHCb over the next few years, given its flexibility supporting the x86_64 instruction set, its high memory throughput from the on-package MCDRAM and its competitive price point-throughput ratio. However, other algorithms used in the LHCb software framework need to be adapted to make the most out of many-core architecture before a more definite answer can be given to the suitability of many-core hardware platforms such as Intel Xeon Phi for LHCb's software framework.

We have evaluated accelerators in the context of our synthetic benchmark Cross Kalman Mathtest. Consumer-grade graphics cards show a competitive price per throughput ratio, an effect that is magnified when moving to single precision. In order to evaluate the impact of the required data transmissions in graphics cards, a Gaudi demonstrator would have to be developed. We intend to port Cross Kalman to GPU accelerators and further analyze our software scalability.

Given the low arithmetic intensity of our formulation, our application utilizes efficiently the processors under study, achieving peak floating point performance on all benchmarked platforms. We will continue to track the performance of modern hardware architectures and adapt our software to it, and observe the evolution of the different platforms. At the same time a more compute intensive formulation or improvements in the implementation's cache-locality could offer significant performance gains.

Acknowledgements

The authors would like to thank the High-Throughput Computing Collaboration at CERN openlab for fruitful discussions through the process of designing and writing the presented software, and early access to Intel hardware. Thanks to C. Potterat for his contributions on the validation of the software. Thanks to F. Lemaitre for his contribution of the vectorized transposition code, and to O. Bouizi and S. Harald for the low-level code discussions and for providing early results and insight on the Xeon Phi architecture. Thanks to R. Nobre, currently working at the SPeCS research group at Faculdade de Engenharia da Universidade do Porto (FEUP), for significant improvements to the performance of the CUDA and OpenCL Mathtest modules. In addition, thanks to W. Hulsbergen and R. Aaij for the mathematical discussions and data structure design. Finally, thanks to N. Neufeld and A. Riscos Núñez for their guidance and support.

References

1. The LHCb Collaboration . *Framework TDR for the LHCb Upgrade: Technical Design Report.* CERN-LHCC-2012-007. LHCb-TDR-12: ; 2012.
2. The LHCb Collaboration . *LHCb Trigger and Online Upgrade Technical Design Report.* CERN-LHCC-2014-016. LHCB-TDR-016: ; 2014.

3. Kálmán R. E.. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering.* 1960;82(1):35–45.
4. Mcgee L. A., Schmidt S. F.. *Discovery of the Kalman filter as a practical tool for aerospace and industry.* : ; 1985.
5. Houtekamer P. L., Mitchell H. L.. Data Assimilation Using an Ensemble Kalman Filter Technique. *Monthly Weather Review.* 1998;126(3):796–811.
6. Welch G., Bishop G.. *An Introduction to the Kalman Filter.* : Chapel Hill, NC, USA; 1995.
7. Hulsbergen W.D.. The global covariance matrix of tracks fitted with a Kalman filter and an application in detector alignment. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment.* 2009;600(2):471 - 477.
8. Cerati G., Elmer P., Lantz S., et al. Kalman Filter Tracking on Parallel Architectures. *Journal of Physics: Conference Series.* 2015;664(7):072008.
9. Lin Zhiyun, Wang Chen. Scheduling parallel Kalman filters for multiple processes. *Automatica.* 2013;49(1):9 - 16.
10. Lu Shiyuan, Lin Zhiyun, Zheng Ronghao, Yan Gangfeng. Scheduling parallel Kalman filters with quantized deadlines. *Systems & Control Letters.* 2015;86:9 - 15.
11. Huang M. Y., Wei S. C., Huang B., Chang Y. L.. Accelerating the Kalman Filter on a GPU. In: :1016-1020; 2011.
12. Cámpora Pérez D. H.. LHCb Kalman filter cross-architecture studies. *Journal of Physics: Conference Series.* ;:to appear.
13. Cámpora Pérez D. H., Awile O., Potterat C.. A high-throughput Kalman filter for modern SIMD architectures. *Euro-Par 2017: Parallel Processing Workshops.* ;:to appear.
14. Aaij R., Fontana M., Le Gac R., et al. *Upgrade trigger: Biannual performance update.* : ; 2017.
15. Mertens S.. The Easiest Hard Problem: Number Partitioning. *arXiv:cond-mat/0310317.* 2003;. arXiv: cond-mat/0310317.
16. Gou C., Kuzmanov G., Gaydadjiev G. N.. SAMS Multi-layout Memory: Providing Multiple Views of Data to Boost SIMD Performance. In: ICS '10:179–188ACM; 2010; New York, NY, USA.
17. Barrand G., others . GAUDI - A software architecture and framework for building HEP data processing applications. *Comput. Phys. Commun..* 2001;140:45-55.
18. Fog A.. *VCL C++ vector class library.* 2012.
19. Karpiński P., McDonald J.. A High-performance Portable Abstract Interface for Explicit SIMD Vectorization. In: PMAM'17:21–28ACM; 2017; New York, NY, USA.
20. Williams S., Waterman A., Patterson D.. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM.* 2009;52(4):65.
21. Schiller M.. Track reconstruction and prompt K^0_S production at the LHCb experiment. Dissertation,University of Heidelberg,2011.

□

REFERENCES

[1] D. H. Cámpora Pérez, N. Neufeld, and A. Riscos Núñez. "A Fast Local Algorithm for Track Reconstruction on Parallel Architectures". In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019, pp. 698–707.

[2] D. H. Cámpora Pérez and O. Awile. "An efficient low-rank Kalman filter for modern SIMD architectures". In: *Concurrency and Computation: Practice and Experience* 30.23 (Dec. 2018), e4483.

[3] T. L. Collaboration et al. "The LHCb Detector at the LHC". In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08005–S08005.

[4] E. Mobs. "The CERN accelerator complex - August 2018". In: *The CERN accelerator complex Octobre 2016* (Aug. 2018).

[5] R. Aaij et al. *Expression of Interest for a Phase-II LHCb Upgrade: Opportunities in flavour physics, and beyond, in the HL-LHC era*. Feb. 2017.

[6] LHCb Collaboration. *LHCb Tracker Upgrade Technical Design Report*. Feb. 2014.

[7] LHCb Collaboration. *LHCb VELO Upgrade Technical Design Report*. Nov. 2013.

[8] O. Callot. *FastVelo, a fast and efficient pattern recognition package for the Velo*. Jan. 2011.

[9] R. Quagliani. *Study of Double Charm B Decays with the LHCb Experiment at CERN and Track Reconstruction for the LHCb Upgrade*. Springer Theses. Cham: Springer International Publishing, 2018.

[10] R. J. Ekelhof. "Studies for the LHCb SciFi Tracker". PhD thesis. Technische Universität Dortmund, Germany.

[11] LHCb Collaboration. *LHCb PID Upgrade Technical Design Report*. Nov. 2013.

[12] M. T. Schiller. "Track reconstruction and prompt K_S^0 production at the LHCb experiment". PhD thesis. University of Heidelberg, Germany, 2011.

[13] O. Omelaenko et al. *LHCb calorimeters : Technical Design Report*. CERN, 2000.

[14] LHCb Collaboration. *LHCb Trigger and Online Upgrade Technical Design Report*. May 2014.

[15] T. Colombo et al. "The LHCb Online system in 2020: trigger-free read-out with (almost exclusively) off-the-shelf hardware". In: *Journal of Physics: Conference Series* 1085 (Sept. 2018), p. 032041.

[16] P. Moreira, Kloukinas, and A. Marchioro. "The GBT : A proposed architecure for multi-Gb/s data transmission in high energy physics". In: *Proceedings of the Topical Workshop on Electronics for Particle Physics TWEPP-07*. CERN, 2007, pp. 332–336.

[17] D. H. Cámpora Pérez, R. Schwemmer, and N. Neufeld. "Protocol-Independent Event Building Evaluator for the LHCb DAQ System". In: *IEEE Transactions on Nuclear Science* 62.3 (June 2015), pp. 1110–1114.

[18] B. Voneki et al. "Evaluation of 100 Gb/s LAN networks for the LHCb DAQ upgrade". In: *2016 IEEE-NPSS Real Time Conference (RT)*. IEEE, June 2016, pp. 1–3.

[19] G. Barrand et al. "GAUDI - The software architecture and framework for building LHCb data processing applications". In: *11th International Conference on Computing in High-Energy and Nuclear Physics (CHEP 2000)*. 2000, pp. 92–95.

[20] B. Barney. *Introduction to Parallel Computing*. URL: https://computing.llnl.gov/tutorials/parallel_comp/.

[21] G. Moore. "Cramming More Components Onto Integrated Circuits". In: *Electronics* 38.8 (Jan. 1965).

[22] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Ed. by Pearson. 2011.

[23] R. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974), pp. 256–268.

[24] H. Sutter. *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. URL: http://www.gotw.ca/publications/concurrency-ddj.htm.

[25] G. M. Amdahl and G. M. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*. New York, New York, USA: ACM Press, 1967, p. 483.

[26] J. L. Gustafson and J. L. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (May 1988), pp. 532–533.

[27] M. J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960.

[28] J. L. Hennessy and D. A. Patterson. *Computer architecture : a quantitative approach*. Morgan Kaufmann, 2011.

[29] S. Williams, A. Waterman, and D. Patterson. "Roofline". In: *Communications of the ACM* 52.4 (Apr. 2009), p. 65.

[30] M. Á. Martínez del Amor. "Aceleración de Simuladores de Sistemas de Membranas Mediante Computación de Altas Prestaciones con GPU". PhD thesis. Universidad de Sevilla, Spain, 2013.

[31] J. E. Stone, D. Gohara, and G. Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: *Computing in Science & Engineering* 12.3 (May 2010), pp. 66–73.

[32] *TOP500 Supercomputer Sites*. URL: https://www.top500.org/.

[33] NVIDIA. *CUDA C Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[34] R. Wilton et al. "Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space". In: *PeerJ* 3 (2015), e808.

[35] R. Wilton et al. "Arioc: GPU-accelerated alignment of short bisulfite-treated reads". In: *Bioinformatics* 34.15 (2018).

[36] S. Pawar, A. Stanam, and Y. Zhu. "Evaluating the computing efficiencies (specificity and sensitivity) of graphics processing unit (GPU)-accelerated DNA sequence alignment tools against central processing unit (CPU) alignment tool". In: *Journal of Bioinformatics and Sequence Analysis* 9.2 (2018), pp. 10–14.

[37] M. D. Cranmer et al. “Bifrost: A Python/C++ Framework for High-Throughput Stream Processing in Astronomy”. In: *Journal of Astronomical Instrumentation* 6.04 (2017), p. 1750007.

[38] A. Recnik et al. “An efficient real-time data pipeline for the CHIME Pathfinder radio telescope X-engine”. In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2015, pp. 57–61.

[39] P. Sen and V. Singhal. “Event selection for MUCH of CBM experiment using GPU computing”. In: *2015 Annual IEEE India Conference (INDICON)*. IEEE, Dec. 2015, pp. 1–5.

[40] D. vom Bruch. “Online Data Reduction using Track and Vertex Reconstruction on GPUs for the Mu3e Experiment”. In: *EPJ Web of Conferences* 150 (Aug. 2017). Ed. by C. Germain et al., p. 00013.

[41] A. Glazov et al. “Filtering tracks in discrete detectors using a cellular automaton”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 329.1-2 (May 1993), pp. 262–268.

[42] D. Funke et al. “Parallel track reconstruction in CMS using the cellular automaton approach”. In: *Journal of Physics: Conference Series* 513.5 (June 2014), p. 052010.

[43] D. Rohr et al. “GPU-accelerated track reconstruction in the ALICE High Level Trigger”. In: *Journal of Physics: Conference Series* 898.3 (Oct. 2017), p. 032030.

[44] D. H. Cámpora Pérez. “A Study of a Parallel Implementation for the Pixel VELO Subdetector”. MA thesis. Universidad de Sevilla, Spain, 2013.

[45] LHCb Collaboration. *Upgrade Software and Computing*. Mar. 2018.

[46] T. Bird et al. *VP Simulation and Track Reconstruction*. Oct. 2013.

[47] G. Blelloch. “Scans as Primitive Parallel Operations”. In: *IEEE Transactions on Computers* 38 (1987), pp. 1526–1538.

[48] H. Nguyen and NVIDIA Corp. *GPU gems 3*. Addison-Wesley, 2008, p. 942.

[49] NVIDIA Research. *CUB: Main Page*. URL: https://nvlabs.github.io/cub/.

[50] R. H. C. Lopes, I. D. Reid, and P. R. Hobson. "A well-separated pairs decomposition algorithm for k-d trees implemented on multi-core architectures". In: *Journal of Physics: Conference Series* 513.5 (June 2014), p. 052011.

[51] C. Joram et al. *LHCb Scintillating Fibre Tracker Engineering Design Review Report: Fibres, Mats and Modules*. Mar. 2015.

[52] D. A. Glaser. "Some Effects of Ionizing Radiation on the Formation of Bubbles in Liquids". In: *Physical Review* 87.4 (Aug. 1952), pp. 665–665.

[53] B. Friman. *The CBM physics book : compressed baryonic matter in laboratory experiments*. Springer, 2011, p. 980.

[54] R. Frühwirth and M. Regler. *Data analysis techniques for high-energy physics.* Cambridge University Press, 2000.

[55] P. Billoir. "Progressive track recognition with a Kalman-like fitting procedure". In: *Computer Physics Communications* 57.1-3 (Dec. 1989), pp. 390–394.

[56] R. Frühwirth. "Application of Kalman filtering to track and vertex fitting". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 262.2-3 (Dec. 1987), pp. 444–450.

[57] A. Fröhlich et al. *MARC - track finding in the split field magnet facility*. July 1976.

[58] J. Olsson et al. "Pattern recognition programs for the JADE jet-chambers". In: *Nuclear Instruments and Methods* 176.1-2 (Oct. 1980), pp. 403–407.

[59] A. Pernia et al. "Use of R-trees to improve reconstruction time in pixel detectors". In: *Proceedings of the CTD/WIT*. 2019.

[60] D. Rohr et al. "ALICE HLT TPC Tracking of Pb-Pb Events on GPUs". In: *Journal of Physics: Conference Series* 396.1 (Dec. 2012), p. 012044.

[61] H. Kälviäinen et al. "Probabilistic and non-probabilistic Hough transforms: overview and comparisons". In: *Image and Vision Computing* 13.4 (May 1995), pp. 239–252.

[62] G. W. Milligan and M. C. Cooper. "Methodology Review: Clustering Methods". In: *Applied Psychological Measurement* 11.4 (Dec. 1987), pp. 329–354.

[63] H. Eichinger. "Global methods of pattern recognition". In: *Nuclear Instruments and Methods* 176.1-2 (Oct. 1980), pp. 417–424.

[64] L. McInnes, J. Healy, and S. Astels. "hdbscan: Hierarchical density based clustering". In: *The Journal of Open Source Software* 2.11 (Mar. 2017), p. 205.

[65] D. Funke et al. "Parallel track reconstruction in CMS using the cellular automaton approach". In: *Journal of Physics: Conference Series* 513.5 (June 2014), p. 052010.

[66] D. Cassel and H. Kowalski. "Pattern recognition in layered track chambers using a tree algorithm". In: *Nuclear Instruments and Methods in Physics Research* 185.1-3 (June 1981), pp. 235–251.

[67] *Intel SPMD Program Compiler*. URL: https://ispc.github.io/.

[68] V. Akishina and I. Kisel. "Parallel 4-Dimensional Cellular Automaton Track Finder for the CBM Experiment". In: *Journal of Physics: Conference Series* 762 (Oct. 2016), p. 012047.

[69] X. Li and Z. Fang. "Parallel clustering algorithms". In: *Parallel Computing* 11.3 (Aug. 1989), pp. 275–290.

[70] LHCb Collaboration. "LHCb detector performance". In: *International Journal of Modern Physics A* 30.07 (Mar. 2015), p. 1530022.

[71] Y. Amhis et al. *Description and performance studies of the Forward Tracking for a scintilating fibre detector at LHCb*. Apr. 2014.

[72] S. Markidis et al. "NVIDIA Tensor Core Programmability, Performance & Precision". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Mar. 2018, pp. 522–531. arXiv: 1803.04014.

[73] R. E. Kalman. "A New Approach to Linear Filtering and Prediction Problems". In: *Journal of Basic Engineering* 82.1 (Mar. 1960), p. 35.

[74] S. Gorbunov et al. "Fast SIMDized Kalman filter based track fit". In: *Computer Physics Communications* 178.5 (Mar. 2008), pp. 374–383.

[75] G. Cerati et al. "Kalman Filter Tracking on Parallel Architectures". In: *Journal of Physics: Conference Series* 664.7 (Dec. 2015), p. 072008.

[76] D. Hugo and C. Pérez. “LHCb Kalman Filter cross architecture studies”. In: *Journal of Physics: Conference Series* 898.3 (Oct. 2017), p. 032052.

[77] D. H. Cámpora Pérez et al. “Cross-architecture Kalman filter benchmarks on modern hardware platforms”. In: *Journal of Physics: Conference Series* 1085 (Sept. 2018), p. 032046.

[78] D. H. Cámpora Pérez, O. Awile, and C. Potterat. “A High-Throughput Kalman Filter for Modern SIMD Architectures”. In: *Euro-Par 2017: Parallel Processing Workshops*. Springer International Publishing, 2018, pp. 378–389.

[79] A. P. Badalov. “Coprocessor integration for real-time event processing in particle physics detectors”. PhD thesis. Universitat Ramon Llull, Spain, 2016.

[80] E. Zenker et al. “Alpaka - An Abstraction Library for Parallel Kernel Acceleration”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (Feb. 2016), pp. 631–640. arXiv: 1602.08477.

[81] E. Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995, p. 395.

[82] R. Aaij et al. “Design and performance of the LHCb trigger and full real-time reconstruction in Run 2 of the LHC”. In: *Journal of Instrumentation* 14.04 (Apr. 2019), P04013–P04013.

[83] P. Fernandez Declara et al. “A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures”. In: *IEEE Access* (2019), pp. 91612–91626.

[84] M. Stahl. “Machine learning and parallelism in the reconstruction of LHCb and its upgrade”. In: *J. Phys. : Conf. Ser. 898 (2017) 042042* (Oct. 2017).

[85] R. Aaij et al. “Performance of the LHCb trigger and full real-time reconstruction in Run 2 of the LHC”. In: *JINST* 14.04 (2019), P04013. arXiv: 1812.10790 [hep-ex].

[86] M. De Cian et al. *Status of HLT1 sequence and path towards 30 MHz*. Mar. 2018.

[87] C. IEEE Computer Society. Technical Committee on Microprogramming and Microarchitecture., V. ACM Special Interest Group on Microprogramming., and ACM Special Interest Group on Programming Languages. *International Symposium on Code Generation and Optimization : CGO 2004 : 20-24 March, 2004 : San Jose, California*. IEEE Computer Society, 2004, p. 325.

[88] H. C. Edwards, C. R. Trott, and D. Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3202–3216.