PARTICLE TRACKING AND IDENTIFICATION ON MULTI- AND MANY-CORE HARDWARE PLATFORMS

by

PLÁCIDO FERNÁNDEZ DECLARA

A dissertation submitted by in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

Computer Science and Technology

Universidad Carlos III de Madrid

Advisors:

José Daniel García Sánchez
Niko Neufeld

September 2020

To my parents.
A mis padres.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [87]

ACKNOWLEDGMENTS

Since I finished my master's degree and started working in the aerospace industry with GMV I have been interested in pursuing a PhD. But I did not find the right challenge or topic I was looking for. One day I received a call from my professor from Computer Science and Engineering degree and director of the master's degree I would later study in Leganés. He let me know about an opening position to do a PhD in High Performance Computing at CERN, in Switzerland. It was just what I was looking for; an incredible computing challenge working in the Large Hadron Collider. I applied for it and some interviews later I got the job.

I have to thank first and foremost my parents. It is thanks to them that I was able to purse any dream and challenge I ever imagined and I am ever thankful for their love and support. To my love and friend Afri, who helps me overcome any situation and makes any adventure an amazing journey to enjoy life.

I would like to thank my friends in Madrid for endless hours of explorations, discussions and joy in the good and bad times. To my childhood friends Javi and Josemari for every minute in the holm oak woods and the surrounding neighbourhoods. To Miguel, Guille, Centeno and all the group for adventure, climbing and nights in the mountains. To all the people that supported me in GMV... To Jordi, Albano, Patricio, Puma, Aven, Romo, Oso and Abel for many, many hours of rehearsal, all the concerts around Spain and experiences. To Cristian who took care of me in the river and plains. To Jorge for we supported each other to improve and finish our studies. To Auro and Álex who showed and accompany us on discovering the Alps, the lakes and the area. I am eternally grateful to my family who always have been supportive and encouraged me to achieve the career and life I wanted.

I am grateful to Niko and Omar for showing me the way in my doctoral studies. To Jose Daniel for the advice, guiding and conversations all along my career. To Daniel who made the experience at CERN so much more enjoyable. To Luismi for the discussions at CERN and exploring the mountains together. To Dorothea, Flavio, Rainer and Roel for all the help to make Allen an amazing thing. Thank you to all the people from the LHCb experiment and CERN that I worked with,

for showing me the way of physics, for the BBQs at Point 8, endless Factorio games and all the great discussions. Thanks to my colleagues at Universidad Carlos III de Madrid in the ARCOS research group. Firstly, I would like to thank to David del Río, Manuel Dolz and Javier Fernández for their help with the integration with GrPPI. Secondly, I would like to also thank Javier Garcia-Blas who helped me with the experiments in GPU architectures.

Thanks to the Spanish MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD though project grant TIN2016-79637-P TOWARDS UNIFICATION OF HPC AND BIG DATA PARADIGMS, the EU Project ICT 644235 "REPHRASE: REfactoring Parallel Heterogeneous Resource-Aware Applications", Spanish "Ministerio de Economía y Competitividad" under the project grant TIN2016-79637-P "Towards Unification of HPC and Big Data Paradigms", and Madrid Regional Government, CABAHLA-CM (ConvergenciA Big dAta-Hpc: de Los sensores a las Aplicaciones) grant number S2018/TCS-4423.

PUBLISHED AND SUBMITTED CONTENT

Publications and research contributions from the author included as part of this thesis:

- Plácido Fernández et al. «Parallelizing and Optimizing LHCb-Kalman for Intel Xeon Phi KNL Processors.» In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2018, pp. 741–750. DOI: 10.1109/PDP2018.2018.00121
  - This publication is wholly included in this thesis in Chapter 6.
  - The material from this source included in this thesis is not singled out with typographic means and references.
- Placido Fernandez Declara et al. «A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures.» In: *IEEE Access* 7 (2019), pp. 91612–91626. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927261
  - This publication is wholly included in this thesis in Chapter 7.
  - The material from this source included in this thesis is not singled out with typographic means and references.
- Placido Fernandez Declara and J. Daniel Garcia. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2020. Accepted for publication at CHEP 2019 Proceedings, Adelaide, Australia
  - This publication is partly included in this thesis in Chapter 8.
  - The material from this source included in this thesis is not singled out with typographic means and references.

OTHER RESEARCH MERITS

A list of publications and research contributions produced during the work of this thesis is presented here:

Publications:

- Plácido Fernández et al. «Parallelizing and Optimizing LHCb-Kalman for Intel Xeon Phi KNL Processors.» In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2018, pp. 741–750. DOI: 10.1109/PDP2018.2018.00121
- Placido Fernandez Declara et al. «A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures.» In: *IEEE Access* 7 (2019), pp. 91612–91626. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927261
- Placido Fernandez Declara and J. Daniel Garcia. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2020. Accepted for publication at CHEP 2019 Proceedings, Adelaide, Australia
- Roel Aaij et al. «Allen: A High-Level Trigger on GPUs for LHCb.» In: *Computing and Software for Big Science* 4.7 (2020). DOI: 10.1007/s41781-020-00039-7

Posters:

- Placido Fernandez Declara. «CompassUT: study of a GPU track reconstruction for LHCb upgrades.» 2019. URL: https://cds.cern.ch/record/2665033. Poster presented at Winter LHCC sessions, CERN, Switzerland
- Placido Fernandez Declara. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2019. URL: https://cds.cern.ch/record/2699802. Poster presented at CHEP 2019, Adelaide, Australia

Talks at international conferences:

- Placido Fernandez Declara. *Fast Kalman Filtering: new approaches for the LHCb upgrade*. Tech. rep. 2018. URL: http://cds.cern.ch/record/2631784

Technical reports:

- LHCb Collaboration. *LHCb Upgrade GPU High Level Trigger Technical Design Report*. Tech. rep. CERN-LHCC-2020-006. LHCB-TDR-021. Geneva: CERN, 2020. URL: https://cds.cern.ch/record/2717938

I am also a co-author in a number of LHCb related papers. The full list can be found at:

- Placido Fernandez Declara. *Google Scholar profile*. https://scholar.google.com/citations?user=Ygkq_7YAAAAJ. 2020

ABSTRACT

Particle track reconstruction in high-energy physics is used for purposes that include simulation, particle analysis, and particle collision filtering and recording. The LHCb experiment at CERN is undergoing an upgrade where improvements are being applied to the detectors, hardware and software. One of the main changes of this upgrade includes the removal of the "hardware filter" which used custom electronics to filter particle collisions, and its substitute: the "software filter". This new filter is composed of commodity hardware which must be able to process a data rate of 40 Tb per second in real-time. Different architectures are being considered to achieve this goal, and the software used to compute it must be optimized and improved to achieve the target data throughput. This software filter is used to reconstruct particle collisions, also known as *events*, including the trajectories of the resulting particles, which later are analyzed and used to help explain problems like the matter-antimatter asymmetry.

This thesis explores different opportunities with multi and many-core architectures, to improve the throughput processing of particle collisions, and the maintainability and improvement of the source code used for it.

The Kalman filter algorithm is widely used in high-energy physics for particle reconstruction, and the Intel Xeon Phi KNL processor offers a many-core x86 architecture that is well suited for parallel workloads. Performance and maintainability improvements are discussed, where optimization are targeted towards the Intel Xeon Phi processor.

GPU architectures are a good fit for high-energy physics workloads, where its highly parallel architecture can benefit the throughput processing of it. A GPU framework for event filtering is discussed, in particular the optimizations and changes implemented to a tracking algorithm to deliver high-throughput.

Finally vectorization opportunities for CPUs are explored by using data-oriented algorithms and constructs that benefit the vector units found in x86 processors. A SPMD compiler that resembles programming for GPUs is used to improve the readability and performance of these algorithms.

RESUMEN

La reconstrucción de trayectorias en física de partículas se usa con distintos fines entre los que se incluyen la simulación, el análisis y el filtrado y recogida de las colisiones entre protones. El experimento LHCb del CERN se encuentra en pleno proceso de actualización en el que cambios y mejoras serán aplicadas a los detectores, los procesadores y el software. Uno de los principales cambios incluye la eliminación del "filtro hardware" basado en circuitos integrados específicos para estas aplicaciones, por un "filtro software". Este nuevo filtro está compuesto por procesadores de distintas arquitecturas que deben ser capaces de procesar un ratio de datos de 40 Tb por segundo, en tiempo real. Distintas arquitecturas están siendo consideradas para alcanzar este objetivo, y el software utilizado para procesarlo debe ser optimizado y mejorado para conseguir alacanzar el objetivo de rendimiento de procesamiento de datos. Este filtro basado en software es usado para reconstruir las colisiones entre partículas, también conocidas como *eventos*, lo que incluye las trayectorias que se producen tras la colisión entre protones. Estas son procesadas y analizadas posteriormente, lo que ayuda a entender y explicar problemas como la asimetría entre materia y antimateria.

En esta tesis se exploran las potenciales oportunidades que ofrecen las arquitecturas con múltiples núcleos de procesamiento para mejorar el rendimiento al procesar las colisiones entre partículas y el mantenimiento y mejora del código fuente usado para ello. El algoritmo filtro de Kalman es ampliamente utilizado en física de partículas para la reconstrucción de partículas, y el procesador Intel Xeon Phi KNL ofrece una arquitectura x86 con múltiples núcleos que está bien adaptada a cargas de trabajo paralelas.

Las arquitecturas GPU se adaptan bien a los problemas encontrados en física de partículas, donde su arquitectura masivamente paralela puede beneficiar el rendimiento de procesado. En esta tesis se discute un framework software basado en GPUs para filtrado de eventos, en particular se discuten las optimizaciones y cambios implementados para un algoritmo de reconstrucción para conseguir un alto rendimiento.

Finalmente se exploran las oportunidades que presenta la vectorización en CPUs utilizando algoritmos orientados a datos y estructuras que mejoran las unidades de vectorización en los procesadores x86. Un compilador de modelo SPMD que utiliza un modelo similar al utilizado con GPUs, se utiliza para mejorar la legibilidad y rendimiento de los algoritmos.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ALICE A Large Ion Collider Experiment

ATLAS A Toroidal LHC ApparatuS

ASIC Application Specific Integrated Circuit

AVX Advanced Vector Extensions

ccNUMA Cache Coherent Non Unified Memory Access

CERN Conseil Européen pour la Recherche Nucléaire

CMS Compact Muon Solenoid

CUDA Compute Unified Device Architecture

CP Charge Parity

CPU Central Processing Unit

CK Cross Kalman

DNA Deoxyribonucleic acid

DAQ Data Acquisition

DLP Data Level Parallelism

DRAM Direct RAM

DSP Digital Signal Processor

DST Data Summary Tape

EB Event Builder

ECAL Electromagnetic Calorimeter

ECC Error Correcting Code

EFF Event Filter Farm

FLOPS Floating Point operations per Second

FPGA Field Programmable Gate Array

GNSS Global Navigation Satellite System

GPC GPU Processing Clusters

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HCAL Hadronic Calorimeter

HEP High Energy Physics

HLT High Level Trigger

ILP Instruction Level Parallelism

ISPC Implicit SPMD Program Compiler

KNL Knights Landing

MCDRAM Multi Channel DRAM

MISD Multiple Instruction Single Data

MIMD Multiple Instruction Multiple Data

NUMA Non Unified Memory Access

LHC Large Hadron Collider

LHCb Large Hadron Collider beauty

LHCOPN LHC Optical Private Network

LHCONE LHC Open Network Environment

LHCG Large Hadron Collider Grid

LLC Last Level Cache

PS Proton Synchroton

PV Primary Vertex

RAM Random Access Memory

RICH Ring Imaging Cherenkov

SiPM Silicon Photomultipliers

SISD Single Instruction Single Data

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SciFi Scintillating Fibre

SM Streaming Multiprocessor

SM Standard Model

SMT Simultaneous Multi Threading

SNC Sub NUMA Cluster

SPMD Single Program Multiple Data

SPS Super Proton Synchrotron

TBB Threading Building Blocks

TDP Thermal Design Power

TLP Thread Level Parallelism

TPC Texture Processing Clusters

UMA Unified Memory Access

UT Upstream Tracker

VELO Vertex Locator

VPU Vector Processing Units

WLCG Worldwide LHC Computing Grid

Part I

THE LHCB EXPERIMENT

1 INTRODUCTION

The Standard Model of particle physics (SM) is a very successful theory that accurately describes how elementary particles and their forces behave. Yet it struggles to explain some phenomena such as the nature of dark matter and dark energy, neutrino oscillations or the asymmetry between matter and antimatter in the Universe, where the amount of matter exceeds that of the antimatter. Particles described in the SM are grouped into two categories: fermions and bosons. Fermions are further categorized into quarks and leptons each containing 6 particles that are grouped in three families. Bosons are divided into gauge bosons and the Higgs boson, as shown in Figure 1.1. This figure represents each elementary particle. Six quarks are depicted in purple, six leptops in green, four bosons in red and the Higgs boson in yellow. The mass, charge and spin of each one is noted at the top left corner of each particle.

A revision of the SM is described in [65]

Known as the CP Violation

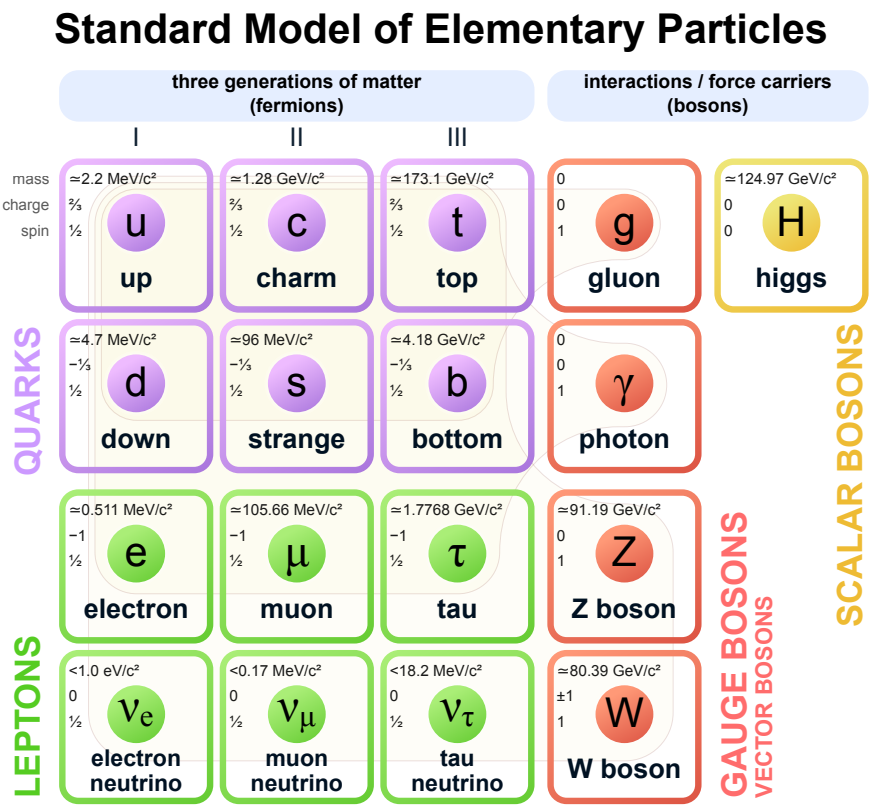


Figure 1.1: The Standard Model or particle physics. Image from [105]

From the name Conseil européen pour la recherche nucléaire

The European Organization for Nuclear Research (CERN) is the biggest particle physics laboratory in the world. CERN produces and uses cutting-edge technologies to do research of particle interactions at every stage of its experiments. CERN studies particle collisions using the largest and most powerful particle collider, the Large Hadron Collider (LHC). The LHC is a 27 Km ring buried up to 175 meters underground where particles collide at nearly the speed of light. Four big experiments -ATLAS, CMS, ALICE and LHCb- use the LHC to complete our understanding of the Universe. These experiments use state of the art technologies to measure the particle collisions in real-time with high precision, recording tens of millions of collisions per second. The data from these collisions is then distributed all over the world through the Grid [92].

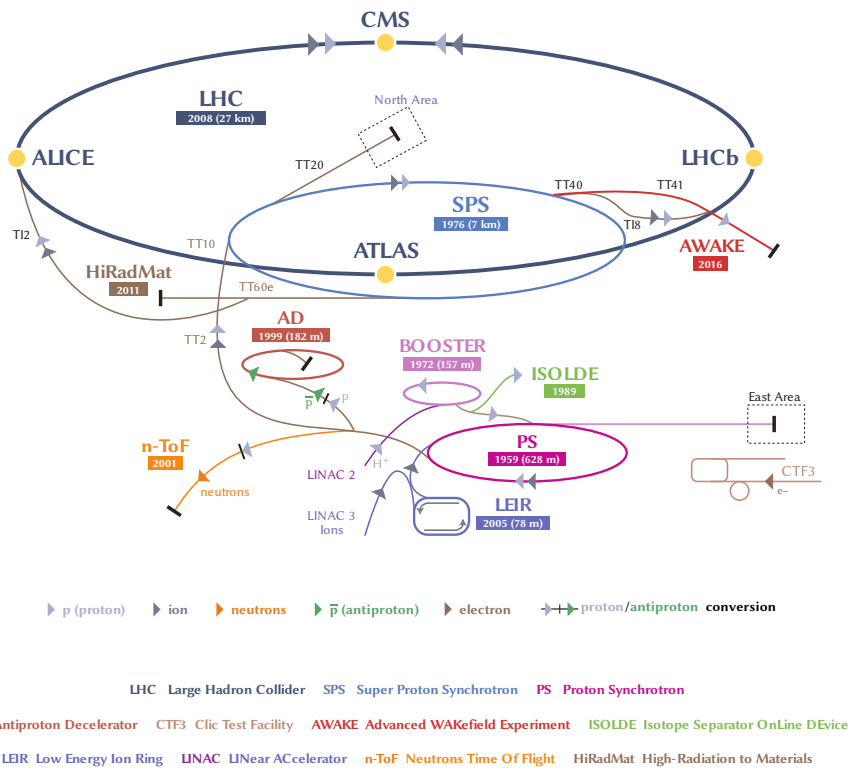


Figure 1.2: CERN accelerator complex, showing all the involved accelerators that push the particles speed and energy to be injected in the LHC. Image from [106]

For particles to collide at a speed close to that of light, they need to be accelerated through different accelerators before they reach the LHC. An overview of CERN accelerators can be seen in Figure1.2. Bunches of protons are accelerated to about a third the speed of light through the LINAC2, these are then passed to the Booster that exit them at over 90% the speed of light. The third stage corresponds to the Proton Synchrotron (PS) which accelerates the bunches over 99.9% the speed of light. The last step is the Super Proton Synchrotron

(SPS) which energizes the bunches before sending them to the LHC in opposite directions so these can cross at four points where the big experiments are located. When bunches of particles cross, only a number of these collide, producing what we call an *event*. The LHC was designed to produce events every 25 ns at a rate of 40 MHz; on average the LHC operates at 30 MHz or 30 million events per second. Each event produces new particles that are measured and recorded as tracks by computers connected to the detectors.

1.1 MOTIVATION

The LHC and its associated experiments are undergoing an upgrade of its components and technologies until 2021. Different detection technologies will be upgraded or substituted completely which will increase the event rate about 5×. The upgraded technologies and the increased event rate will produce an increase in high-energy physics data taking by the experiments. Computing technologies are being redesigned and optimized by the experiments to cope with the expected increases in data throughput.

The LHCb experiment operates at the LHC and it is also being upgraded for the next data-taking period. As in previous data-taking periods, LHCb utilizes a filter which selects and discards particle collisions. This filters is divided into various steps, each of them designed to process different amounts of data. A combination of custom ASICs and x86 processors' filter was used, with the custom electronics being used in the first stage of the filter where the data rate is higher, and software for the x86 stage was used in subsequent stages where the data rate was already reduced. One of the main changes involving the upgrade of LHCb is the removal of the ASICs filter to use a full software trigger based on commodity hardware. The expected data rate this software filter will need to process is 40Tb/s which poses a tremendous challenge to compute in real-time. The needed computing resources were projected for the upcoming upgrade, but the cost and performance of the software did not deliver the expected results. In 2016 the computing needs were estimated between 6 and 10 times below the expected processing data rates.

A high-degree of software optimization is needed to reach the desired throughput, and alternative hardware architectures must be considered to explore the improvements in performance that can be achieved by switching to alternative processors or coprocessors. New particle tracking algorithms that take hardware architectures characteristics can bring performance improvements without sacrificing physics results and redesigning the existing algorithms or creating new ones should be considered for the upgrade.

The *hypothesis* of this thesis can be summarized as follows:

New algorithms, optimizations and hardware architectures can bring throughput improvements to the real-time processing of particle tracking algorithms in the context of high-energy physics. The LHCb experiment can reach the target 40Tb/s data throughput by implementing these techniques and hardware before the next data-taking period commences.

1.2 OBJECTIVES

The main objective of this thesis is the development and optimization of particle tracking algorithms that are efficient in multi- and many-core architectures, and bring throughput performance and source code improvements for the LHCb experiment in the context of the next data-taking period.

This general goal can be divided into the following specific goals:

- **O1:** Explore multi-threading possibilities to parallelize the processing of particle tracks.
- **O2:** Design and optimize algorithms for the Intel Xeon Phi KNL as a many-core target architecture for LHCb.
- **O3:** Explore vectorization opportunities in the tracking algorithms of LHCb.
- **O4:** Implement algorithms for alternative hardware architectures, such as GPUs and study their performance opportunities.

1.3 STRUCTURE OF THE DOCUMENT

The thesis is organized into three main parts, each with a number of chapters to make a total of 9. A brief description of the chapters is presented here:

Part I: The LHCb experiment

Chapter 1: This chapter which briefly introduces CERN and its context, motivates the thesis and states its objective.

Chapter 2: High-energy physics, CERN, the Large Hadron Collider (LHC) and the LHCb experiment in particular are introduced. The main components of the LHCb experiment related to this thesis are described; including a description of the detector and the subdetectors used to reconstruct particle tracks. How particle track reconstruction works at LHCb, its Data Acquisition System, its High Level Trigger and how these will be upgraded are described as the central work of this thesis.

Chapter 3: The computing infrastructure and software used at LHCb to filter, process and analyze particle collisions is described. The two main software processing frameworks named Allen and Gaudi. The Worldwide LHC Computing Grid is explained briefly as one of the main pieces for physics analysis.

Chapter 4: Particle reconstruction or tracking is presented with more detail in this chapter, including the specifics of the LHCb experiment. Different particle tracking methods are discussed, the types of tracks involved in the reconstruction are listed and standard physics efficiency metrics are introduced which are used later in the thesis. The Kalman filter algorithm is introduced using a particle as in the LHCb experiment would occur.

Part II: Parallel Computing

Chapter 5: Introduces parallel computing and its implications. Parallel hardware and software is described, laws and concepts to express speedup and scalability are explained. Hardware concepts affecting computing performance are introduced including memory and alternative hardware architectures. The concept and usage of generic parallel patterns is described.

Part III: Particle tracking in high-energy physics

Chapter 6: Kalman filtering on many-core architectures is introduced. The Intel Xeon Phi is used as the main x86 processor during the whole analysis as one of the candidate hardware architectures for the upgrade. Different parallelization opportunities are explored for the Kalman filter, exploring different intra-track implementation and their effect in the computing performance. The GrPPI library functionality is expanded and the Kalman filter is reimplemented to use generic parallel patterns achieving a comparable performance to the baseline hand-tuned algorithm.

Chapter 7: The Allen GPU framework is introduced and the *Compass* algorithm is described. The decoding and tracking kernels for the GPU are described and implemented, and its throughput performance is discussed. A throughput analysis is presented for different consumer and scientific GPUs alongside a CPU implementation of the algorithm.

Chapter 8: Vectorization opportunities are explored in this chapter. The Allen framework is adapted to compile ISPC algorithms, the VELO kernels are adapted and implemented to compile for ISPC. The UT kernels are re-implemented for the ISPC compiler, removing GPU specific optimizations and focusing on the vectorization capabilities. The throughput results are discussed.

Chapter 9: The conclusions for this thesis are discussed in this chapter. The presented work analyzes the usage and impact of multi- and many-core hardware architectures in the context of high-energy physics. Different high-performance options are explored in the context of the LHCb upgrade, where a high level of software optimization is required to meet the target throughput. Multi-threading, GPU accelerators and vectorization are applied to different tracking algorithms and its impact and performance is discussed.

2

LHCB EXPERIMENT

The Large Hadron Collider beauty (LHCb) [4, 5] experiment is one of the four big physics detector experiments collecting data at the LHC in the border between Switzerland and France. The LHCb detector is a single arm forward spectrometer designed for high-precision measurements and aims to explore the matter-antimatter asymmetry problem known as *CP violation*. The Big Bang should have produced same quantities of matter and antimatter, which then would have annihilated leaving nothing. What we observe differs, as the observable Universe is made out of matter. CP violation partly explains this phenomena, where some other forces not yet known must be discovered to explain this asymmetry [30]. LHCb is composed of various parts which are used to reconstruct particle collisions in detail. The data collected from these particle collisions allows to study B mesons, which allows to better understand this particle (and others) and its different decays to other particles. These decays include measurements of the *CP violation* and other parts of LHCb physics programme. The main parts are:

- *Tracking system*: it reconstructs the particle trajectories caused by a particle collision. It measures the vertices where the trajectories are formed and measures particle momentum.
- *Particle identification system*: it distinguishes between different types of particles obtaining the velocity and energy of the produced particles.
- *Trigger system*: it filters the events by selecting the interesting ones for physics analysis.

The LHCb experiment is depicted in Figure 2.1. It shows the already upgraded subdetectors that will function in the next data-taking period from 2021. Details on the changes and differences between the previous and upgraded subdetectors can be found at [37, 38, 88] The tracking and particle identification subdetectors are presented alongside the magnet. The tracking system is composed of the *VErtex LOcator (VELO)*, the *Upstream Tracker (UT)* and the *Scintillating Fibre (SciFi)*, the particle identification system is composed of the *Ring Imaging Cherenkov (RICH)* subdetectors, the *Electromagnetic (ECAL)* and *Hadron Calorimeters (HCAL)*, and finally the *Muon Stations (M)*. The *trigger system* is composed of the electronics, hardware and software used to collect and process the data coming from these subdetectors.

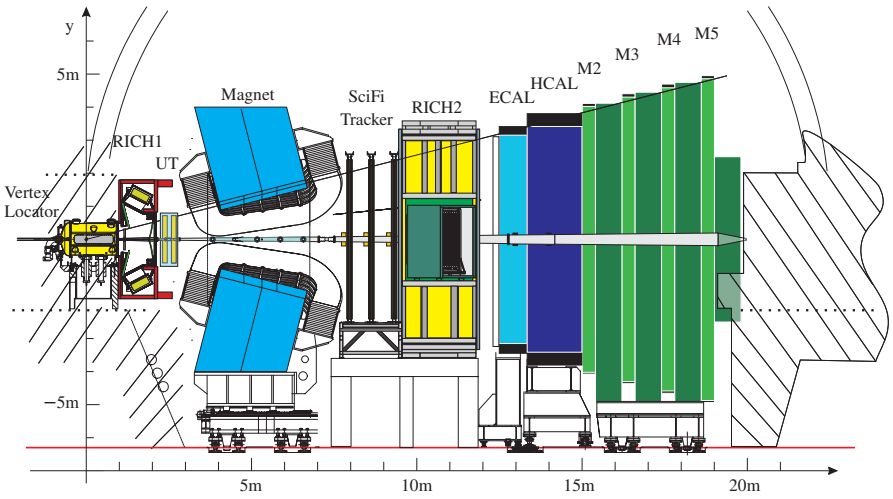


Figure 2.1: Schematic view of LHCb upgrade experiment. Image from [88]

This chapter is structured as follows: In Section 2.1 the tracking system of LHCb is presented, Section 2.2 introduces the Data Acquisition System of LHCb, Section 2.3 presents the High Level Trigger, and Section 2.4 gives a summary of the chapter.

2.1 TRACKING SYSTEM

The process used to reconstruct particle trajectories is known as tracking. The tracking system at LHCb is composed of various subdetectors located at different position in the Z axis. These provide position information for the generated particle trajectories that cross the magnetic field region. The measured values allow to get the vertex of the interaction, the trajectories of charged particles and to obtain a measurement of their momentum. Sorted by forward direction, this is from the interaction point onwards, first is the VELO followed by the UT, the magnet is located in between the UT and the SciFi [103].

2.1.1 *VELO*

VELO is the subdetector enclosing the interaction point where the proton-proton collisions occur. It is composed of two halves each counting 26 silicon pixel detector modules. Each module is composed by four silicon sensors with a "L" shape around the LHCb beam pipe. A silicon sensor is composed of three chips with 256 x 256 square pixels of 55 x 55 µm each, giving a total of 41 M pixels as shown in Figure 2.2. These pixels provide x and y coordinates for the hits of the particles, where the z coordinate is known by association to the module that gives the hit measurement [38].

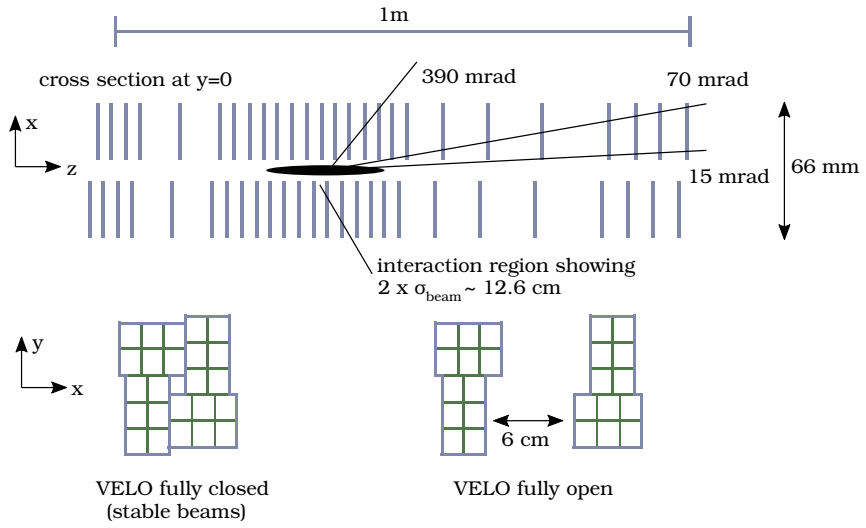


Figure 2.2: Velo subdetector for the upgrade in schematic view. Image from [29]

VELO provides measurements for the charged particles that are generated after a collision and cross the subdetector. The main purposes of VELO are:

- It precisely measures where the primary and secondary vertices originate.
- Provides hits to create initial tracks.
- The VELO is not in the region of influence of the dipole magnet, so the initial tracks measured by the VELO are not bent and form only straight lines.
- The upgraded VELO substitutes microstrips for pixels, which leads to better track reconstruction.

A hit is a signal activation of a particle crossing a detector.

2.1.2 *UT*

The UT subdetector [88] is composed of four planes, where each plane is a single sided silicon strip detector. We refer to the four consecutive planes as UTaX, UTaU, UTbV, UTbX respectively, as can be seen in Figure 2.3. These are sorted into two layers containing 2 planes each, the a and b layers. The X planes are composed of vertical strips whereas the U and V planes are tilted around the Z axis at $+5°$ and $-5°$ respectively. By combining the measurements from the tilted U and V planes, the Y coordinate can also be determined. Each UT plane is composed of micro-strip sensors arranged in vertical staves. A UT plane can be divided into 3 regions with different geometry, where the inner-most region has a finer granularity, and the outer regions

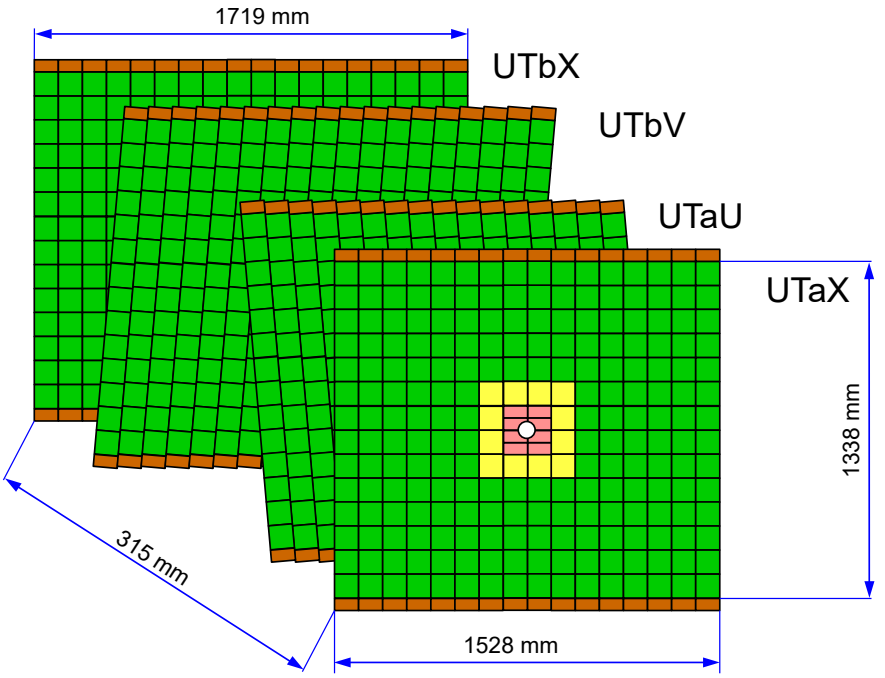


Figure 2.3: The four UT planes are presented in this figure. Image from [88]

have coarser granularity. Each stave measures 160 cm high and 10 cm wide, where various sensors are placed alongside each stave. The sensors in a stave overlap with their neighbour sensors, to avoid gaps, and the vertical staves also overlap for the same reason. The X planes are composed of 16 staves while the U and V are composed of 18 staves. The acceptance of the UT sub-detector is defined by its volume in space, the UT planes for the UT sub-detector. Only particles that traverse this volume can leave signals and are measured.

The UT detector serves various purposes in the LHCb experiment:

- Reconstructs charged particle trajectories that decay after the VELO sub-detector.
- Reconstructs low momentum particles that are bent by the magnet, and go out of acceptance before reaching the SciFi Tracker.
- Gives additional information in the form of hits, that can be used in conjunction with the VELO and SciFi Tracker information to reject tracks.
- As the UT is influenced by the magnet, it can provide momentum resolution for charged particles.
- It can reject low momentum tracks.
- Decreases time to extrapolate VELO tracks to SciFi Tracker by at least a factor of 3.

Finally, UT plays an important role by marking tracks that won't be used by the next tracking detector, the SciFi Tracker. This allows for a faster processing of the whole track reconstruction in the LHCb detector.

2.1.3 *SciFi*

The SciFi subdetector [88] is the tracking system located after the magnet. It is composed of three stations named T1, T2 and T3. These stations are under the influence of the magnet region giving a momentum estimate. Each station consists of four detection planes arranged in a similar fashion to that in the UT. Each plane provides coordinates measurements, where the two planes in the middle are tilted $+5°$ and $-5°$ with respect to the vertical axis. Each detector plane contains 12 modules for a total of 144 modules. The plane arrangement is shown in Figure 2.4.

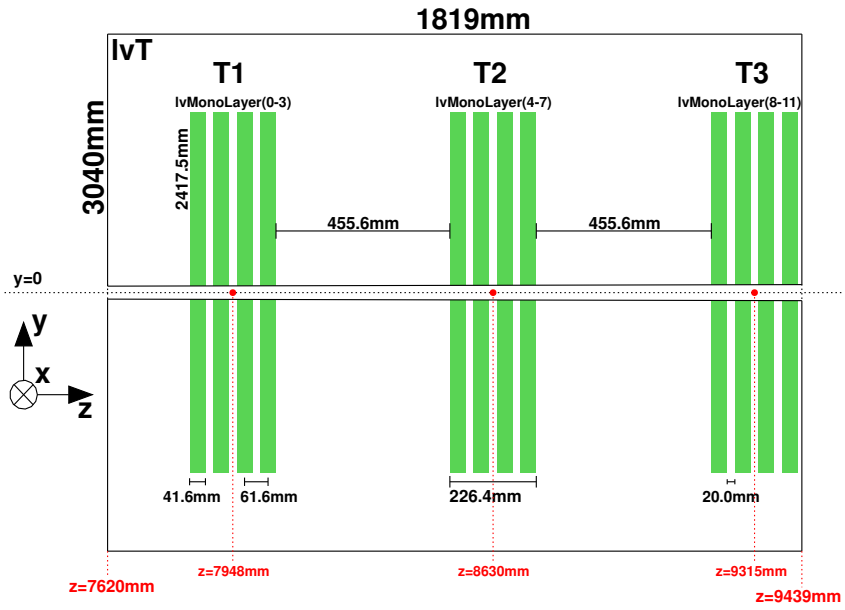


Figure 2.4: Arrangement of the SciFi layers. Image from [88]

Detector modules are composed of 2.5 m long scintillating fibres with a diameter of 250 µm which guide photons that are detected by Silicon Photomultipliers (SiPM). Each detected photon can be associated to a coordinate for a crossing charged particle or hit, although more than one photon detection can occur per hit. This process is represented in Figure 2.5. These fibres are arranged in groups of six in layers, where each groups is a *fibre mat*. Fibre mats are the main component of the SciFi subdetector.

The SciFi tracking system covers a large surface compared to the other tracking subdetectors, covering 340 m^2. It is designed to have a

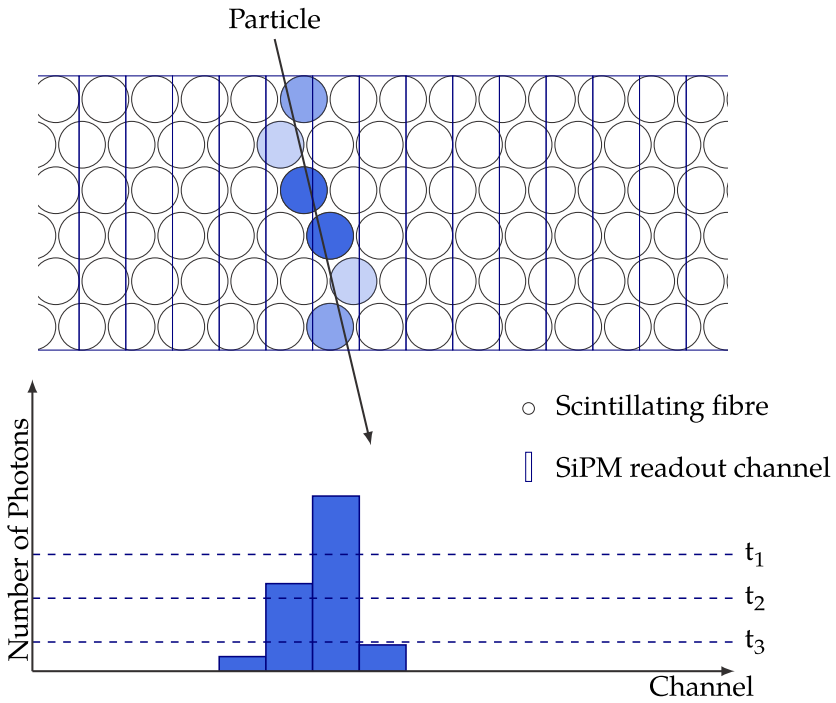


Figure 2.5: A particle leaves an energy deposit. Image from [108]

very high hit efficiency over 99% and a resolution of at least 100µm. The SciFi subdetector serves various purposes for LHCb:

- Reconstructs particles bent by the magnet giving momentum estimation for charged particles.
- It allows to determine precise mass and lifetime resolutions of particles.
- Serves as input for the RICH detector for particle identification.
- Measure tracks in the acceptance region not covered by the UT subdetector.
- Reconstruct T, Long and Downstream tracks.

2.2 DATA ACQUISITION SYSTEM

The data acquisition system (DAQ) at LHCb is designed to filter events in real-time, coping with an input 30 Mhz bunch crossing rate. The average size of the events is estimated to be 150kB, giving an expected data rate of 40 Tb/s. This deluge of data is acquired and filtered by the DAQ, selecting the interesting event for further analysis. Selecting the interesting events requires a two stage process divided into the High Level Trigger 1 and 2 (HLT1 and HLT2), each filtering to different data rates to finally write a data rate of 10GB/s, as seen in

Figure 2.6. Events are fully reconstructed to decide whether to keep the event or not, every 13ms [41, 91].

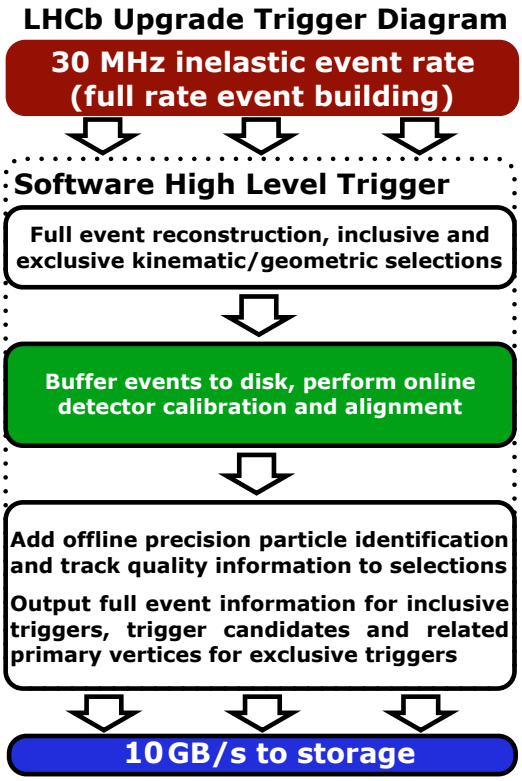


Figure 2.6: Run 3 trigger. Image from [118]

The upgraded high level trigger (HLT) will consist of two main components: the Event-Builder (EB) and the Event-Filter-Farm (EFF). Data from the subdetectors described in 2.1 is sent to the front-end electronics in the form of data fragments containing the event information. These fragments are combined to build the events in real-time, in the EB. The EB is composed of 170 servers connected through a dedicated high-performance network where each uses two 200 Gbit/s Infiniband HDR network, made possible by choosing AMD Rome CPUs. The EB servers compute three main components: a read-out unit reading the information from the detectors through FPGAs connected to the subdetectors, a builder unit that gathers the data fragments from the read-out units to create complete events, and a event manager to schedule the other two components.

After building the events, these are sent to the EFF for filtering. These components are presented in Figure 2.7.

The EFF processes the HLT1 and HLT2 filters to reduce the data rate. The HLT1 selects events by doing a *fast* reconstruction of all the tracks contained in them, and selects based on a PV displacement, momentum of the tracks and muon identification. The rate of events

The muon step is optional.

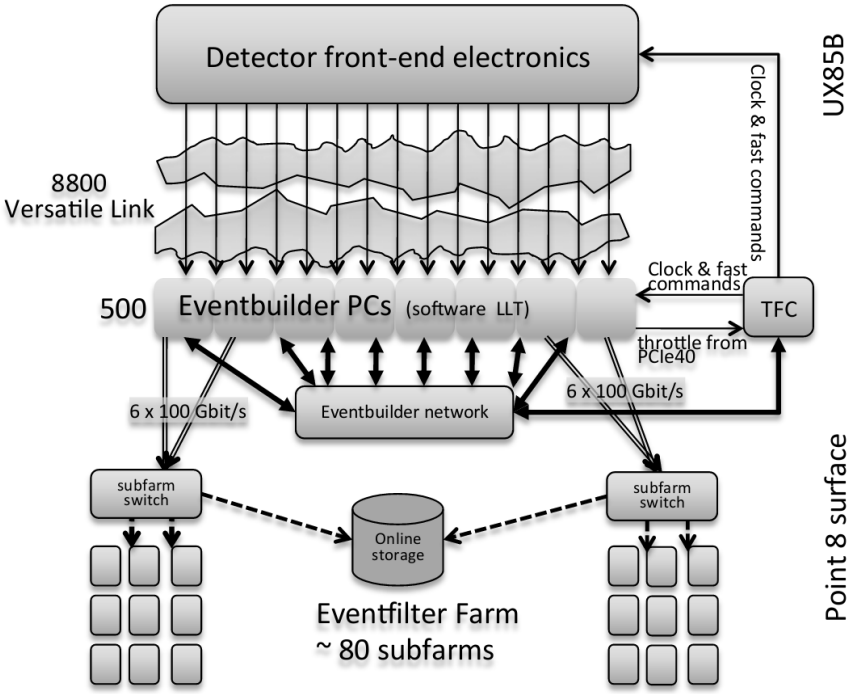


Figure 2.7: Layout of LHCb DAQ. Image from [42]

is expected to be reduced from 30 MHz to 1 MHz. The reduced set of events is stored in a disk buffer to save the output of the HLT1. The HLT2 reads the stored events to perform the *full* reconstruction of the events asynchronously, to reduce the event rate to 100 kHz. While the HLT1 filters the events in real-time during the proton-proton collisions, the HLT2 is processed while there are no collisions. A collision period can last up to a day, leaving a few hours between collision periods used by the HLT2 to make further filtering [42].

2.3 HIGH LEVEL TRIGGER

The HLT1 and HLT2 process various track reconstruction algorithms to perform the selection of events. The HLT1 processes the events in a synchronous manner to the LHC bunch crossing rate, and thus does the processing in real-time. The HLT2 performs the processing asynchronously, when there are no collisions, but still tied to a time constrain of a few hours. Events are selected and discarded based on various criteria obtained after the reconsruction of the particles, such as particles that have a high transverse momentum larger that 2 or 3 GeV. The computing resources planned for the next data taking period did not increase as expected in terms of GFlops; as a consequence the LHCb software stack will not be able to cope with the expected data rate, thus needs to be optimized for the selected hardware [134]. The LHCb software stack uses the Gaudi framework [31] which is

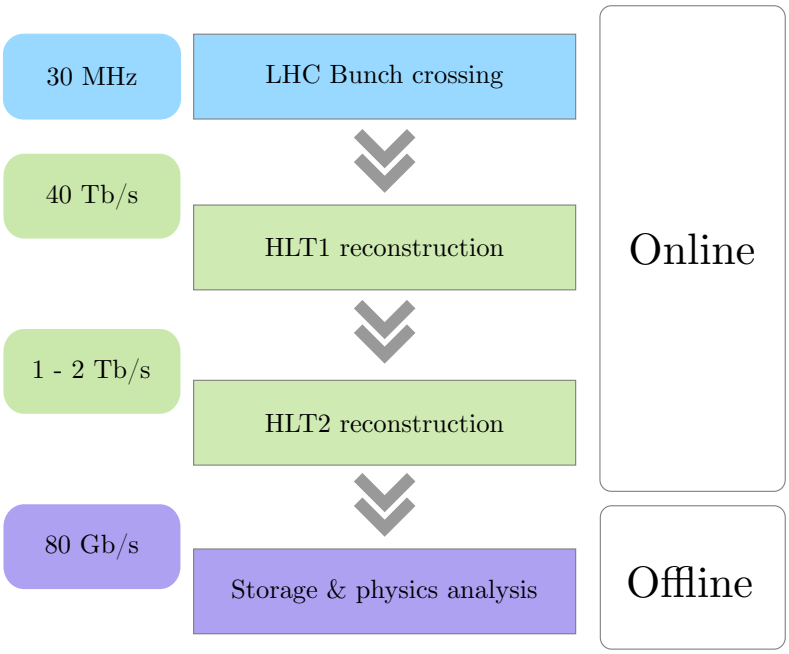


Figure 2.8: From LHC bunch crossing rate to analysis

described in Chapter 3. HLT1 reduces the 30 Tb/s data rate, or 30 MHz event rate to 1 MHz event rate to be used by HLT2. HLT2 computes a similar but different set of algorithms that perform a more precise reconstruction using all the subdetectors in LHCb. Events are then stored for physics analysis and distributed through the Worldwide LHC Computing Grid [90]. Figure 2.8 shows the steps followed from the LHC bunch crossing rate to the final storage for analysis, including the data rates processed at each step.

HLT2 also includes measurements from RICH, Calorimeters and Muons

LHCb software upgrade becomes of critical importance, as compared to the previous data-taking period the data rate is expected to increase by 85×. The increased data rate comes after the decission of removing the previous *hardware level trigger* and moving to a *software level trigger* that performs a full event tracking. The predictions made for the upgrade in terms of computing power and software optimization were underestimated. As a result new algorithms are being developed, and new different EB or EFF architecture designs are being considered [1].

This thesis work focuses on software optimization of reconstruction algorithms for multi- and many-core architectures that are being considered for the EB or EFF. A high level of optimization is needed to be able to process the 40 Tb/s estimated data rate for the given hardware computing constrains.

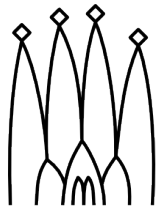
2.4 SUMMARY

This chapter presented the main components of LHCb that are involved in the particle track reconstruction performed at LHCb. These components are central to the work of this thesis; the input data used in the different algorithms comes from these subdetectors and software is optimized based on the design of them for this thesis. The tracking subdetectors are undergoing and upgrade for the next data-taking period (Run3) and these are presented as their upgraded versions. The DAQ and HLT are also being upgraded and have significant changes, being the main one the elimination of the *hardware trigger* that was used in the two previous data-taking periods. The *hardware trigger* will be replaced by a *software trigger* that will use more complete information from the detector, allowing for algorithms to make better filtering decisions to keep or discard events.

3

LHCB SOFTWARE AND COMPUTING

Particle collisions at LHCb are processed, filtered and analyzed at different stages through the various software components. LHCb software is present from the data collection at the detector read-out electronics, to the particle trajectory reconstruction and identification in the HLT1 and HLT2 filters. LHCb software is also used for Monte Carlo Simulation and for further analysis and distribution through the Worldwide LHC Computing Grid. Software at the experiment is developed with the *Gaudi* framework to write data processing applications for High-Energy Physics (HEP) experiments [10]. *Gaudi* is a High Energy Physics software framework that allows to write algorithms to perform the tracking and particle identification. It was used during the the previous data-taking periods at LHCb, it is also used by other HEP experiments, and it is set to be used by the Future Circular Collider [35]. Other pieces of software and frameworks are developed as R&D programs at LHCb to find the most efficient solution for the next data-taking period. In this section, the building blocks of LHCb software framework are outlined, as well as the GPU Allen framework as the novel framework used to process the HLT1.



The Gaudi framework logo representing "La Sagrada Familia"

This chapter is structured as follows: In Section 3.1 the Gaudi software framework is presented, Section 3.2 gives an overview of the Allen GPU framework, Section 3.3 briefly explains the Worldwide LHC Computing Grid, and Section 3.4 gives a summary of this chapter.

3.1 LHCB SOFTWARE FRAMEWORK. GAUDI

The software framework *Gaudi* provides the architecture to build different HEP applications, which is used and developed by LHCb among other experiments. LHCb applications constructed with *Gaudi* include *Moore* and *Brunel* for online and offline reconstruction respectively. These applications enclose the relevant algorithms and topics discussed in this thesis. Other applications of *Gaudi* include the digitalization (*Boole*), analysis (*DaVinci*) or simulation (*Gauss*) [14, 43]. The relationship between these applications is depicted in Figure 3.1.

Names are given after the famous scientists or engineers.

Gaudi provides algorithms and tools with defined input and output data to develop the applications. It separates between data objects and algorithms, where the data objects contain the different inputs from the detector in the form of matrices and vectors; the algorithms are used for reconstruction, identification and simulation tasks. By separating both entities in the framework, different algorithms can be

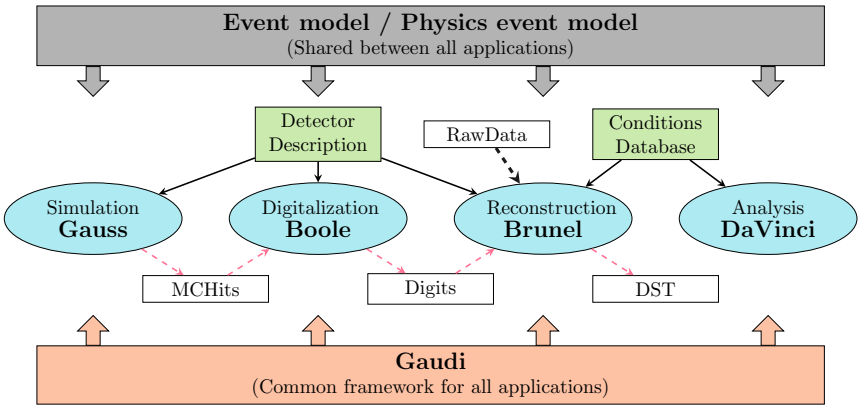


Figure 3.1: LHCb software applications based on Gaudi framework

developed independently while maintaining a common structure for data input and output across the framework.

A common *Event and Physics Model* is shared between all applications for consistency; it describes the event and physics data classes and their relationships. *Gaudi* is also used by all applications as the common software framework to build them. The *detector description* is used as input for *Gauss*, *Boole* and *Brunel*; it describes the materials, structure, geometry and calibration of the detector. The *conditions database* is used during analysis and particle reconstruction: it contains the status at some point in time of the detector, which is only valid for a period of time. Simulations produce *hit* data or Monte Carlo Hits that are digitalized before using them in the reconstruction when executing simulations [124]. The reconstruction of events through *Brunel* produces DST (Data Summary "Tape"). Simulation takes most of the LHCb computing time resources, and it is mainly used outside of data-taking periods [36].

Gaudi itself is open-source and written in C++ [67]

Algorithms for *Gaudi* are written in the C++ language; Python is used to configure the application and define how and which algorithms should run. The reconstruction software at LHCb computes several algorithms that are organized as an acyclic graph. Groups of algorithms correspond to different subdetectors and these are computed following the order in which particles traverse the subdetector: first the VELO, then the UT, then the SciFi. The output of algorithms in that chain serves as input for the ones that come after.

3.2 HLT1 GPU ALLEN FRAMEWORK

The Allen framework allows to run a sequence of GPU parallel algorithms efficiently through a modular and extensible framework. It accommodates the algorithms for the entire HLT1 as depicted in Figure 3.2. It shows the dependencies between the algorithms to reconstruct events in real-time. This sequence includes the tracking

detectors and the optional Muon stations, as it is done in *Moore*. Allen is a multi-threaded framework, where one CUDA *Stream* runs per CPU thread, guaranteeing asynchronous execution of events. Different events are processed independently from each other, allowing for zero communication between CPU threads or GPU streams [29].

Allen source code is available at [145]

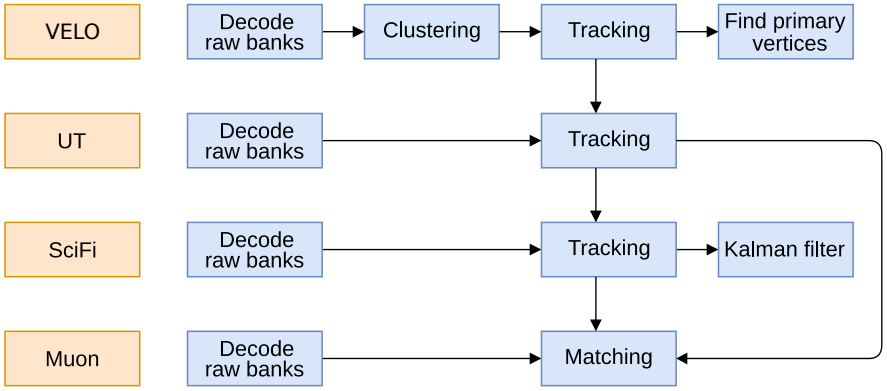


Figure 3.2: Algorithms run in HLT1

Allen HLT1 algorithms are categorized to their corresponding subdetector: VELO, UT, SciFi and Muon. Raw banks data from every subdetector is decoded in software before being able to process it. Tracking algorithms are processed for every tracking subdetector, each with tracking input information from the previous subdetectors. The elements presented in Figure 3.2 are described:

- **VELO**: Raw banks are decoded into fired pixels from the subdetector. Only fired pixels are received to keep data transmission to a minimum. For the VELO decoding, each received raw bank corresponds to a different sensor. With the fired pixels decoded, a *clustering* step is performed. VELO clustering groups neighbour pixels that may be activated from a crossing charged particle. The result is a set of *hits* indicating the coordinates where the particle crossed the subdetector. These *hits* are used to reconstruct the particle trajectory, which for the VELO are straight lines as these tracks are not under the influence of the magnetic field.

- **Find primary vertices**: One of the main goals of the VELO reconstruction is to find the vertices where particle originate from the proton-proton collisions -the *primary vertices*-. Other vertices may originate from particle decays -secondary vertices- which are not processed by this algorithm.

- **UT**: Raw UT banks are decoded into *hits*. UT reconstruction uses the resulting tracks from VELO reconstruction as input. Using the VELO track, it searches for UT hits that match the extrapolation of a VELO track to construct UT tracks. These

tracks present a slightly bent trajectory as UT regions are already under the influence of the magnet, where various combinations of hits may result in possible tracks.

- **SciFi**: as with the other subdetector, the raw banks are decoded into SciFi hits. For this subdetector, tracks from VELO and UT are used as input to construct *long tracks*. As these tracks fully traverse the magnet region, the problem of matching hits to VELO or UT tracks becomes an exponential problem due to the bent trajectory of these particles.
- **Kalman filter**: with a *long track* reconstructed a Kalman filter is applied to the track to better estimate the trajectory and reduce the error associated with it. At this stage a simplified version of the Kalman filter is used.
- **Muon**: raw banks are decoded into Muon hits. Tracks from VELO and UT are extrapolated to the Muon stations to identify these tracks as muons.

3.3 WORLDWIDE LHC COMPUTING GRID

The Worldwide LHC Computing Grid (WLCG) provides the LHC experiments like the LHCb with computing resources for storage, distribution and analysis data from the data-taking periods. These resources are distributed across a collaboration of computer centers comprising institutions, universities and research centres [16, 113]. The WLCG is hierarchically structured by three layers:

- *Tier-0*: is a combination of CERN Data Center in Geneva, Switzerland and the Wigner Research Center for Physics in Budapest, Hungary.
- *Tier-1*: is composed of various large data centers.
- *Tier-2*: are different universities and scientific institutes.

WLCG centers are connected by a dedicated network. The two *Tier-0* centers are connected by three 100Gb/s lines. CERN is then connected to every *Tier-1* center using both a high-bandwidth network called LHC Optical Private Network (LHCOPN) and the LHC Open Network Environment (LHCONE). This structure is represented in Figure 3.3. The *Tier-1* centers expected to be used by LHCb for the next data-taking period are:

- IN2P3-CC (Lyon / France)
- FZK-T1 (Karlsruhe / Germany)
- CNAF-T1 (Bologna / Italy)

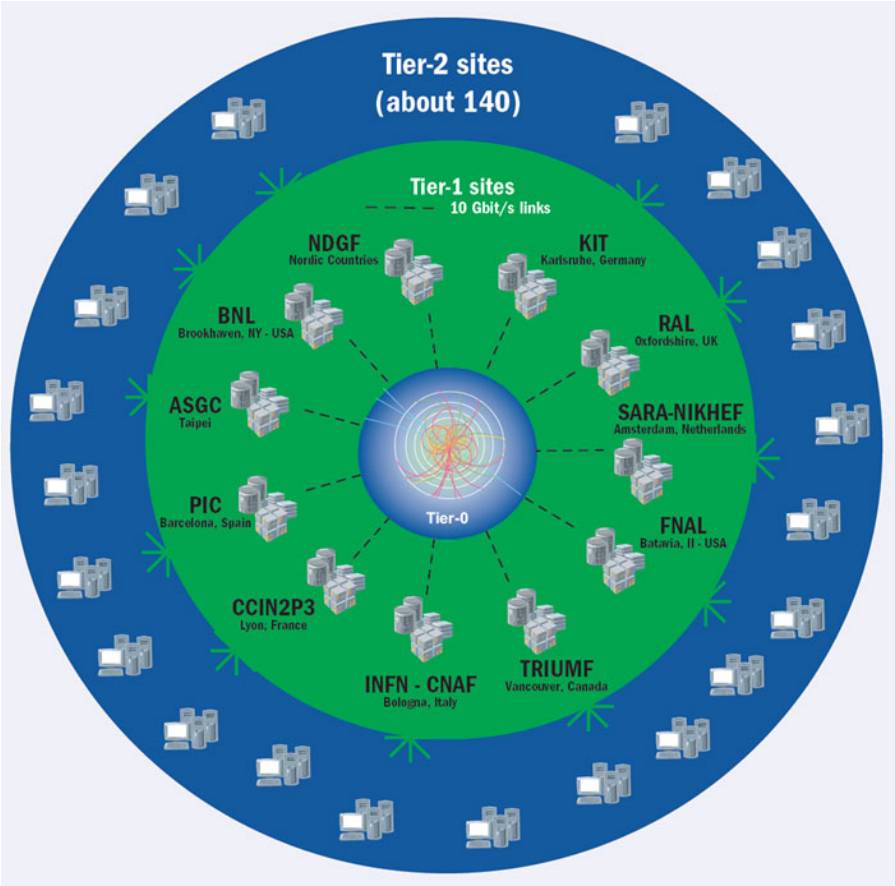


Figure 3.3: WLCG Tiers. Image from [98]

- NL-T1 (Amsterdam / The Netherlands)
- PIC (Barcelona / Spain)
- RAL-T1 (Rutherford / UK)
- RRCKI-T1 (Moscow / Russia)

Computing resources for the LHCb experiments come from LHCb own distributed computing infrastructure, from the WLCG and includes computing resources given by other providers in a voluntary basis. From the WLCG, the tape storage resources is provided by *Tier-0* and *Tier-1* sites only, disk usage is used in *Tier-2* sites, and CPU resources is provided in all levels [90].

The HLT1 at LHCb runs synchronously with the LHC bunch crossing rate; it stores the selected events into a 10 PB buffer. The HLT2 takes as input the selected events from the HLT1 and further reduces the selection, but in an asynchronous manner. The output from HLT2 is then stored at a rate of 10 GB/s. The final stored output will be in the order of tens of PB per year and it is stored and distributed by the WLCG, first by the *Tier-0* and one *Tier-1*, then data is reconstructed at CERN and the *Tier-1* [81].

3.4 SUMMARY

This chapter introduced the main software components used in LHCb related to this thesis. The two main software frameworks used for the HLT are introduced: Gaudi and Allen. The Gaudi framework contains the algorithms for both the HLT1 and HLT2 and is focused on CPU processing, whereas the Allen framework is focused on GPUs and the HLT1 algorithms. Gaudi and Allen differ on the events that can be processed in parallel: while Gaudi can process one event at a time, Allen is designed to process multiple events in parallel. These frameworks contain the algorithms to reconstruct the particle collisions with different levels of detail, and are optimized to deliver high throughput for different architectures. Finally the WLCG is shown as it is the software used after filtering the events in the two steps of the HLT.

4

LHCB TRACK RECONSTRUCTION

Proton-proton collisions at LHCb produce new particles. HEP detector technologies are used at LHCb to reconstruct the results from these collisions, where three main steps can be identified:

- Tracking or particle trajectory reconstruction.
- Vertexing or grouping particles into vertices.
- Particle identification or classifying each particle trajectory to particle (i.e. electron, muon, pion, etc.).

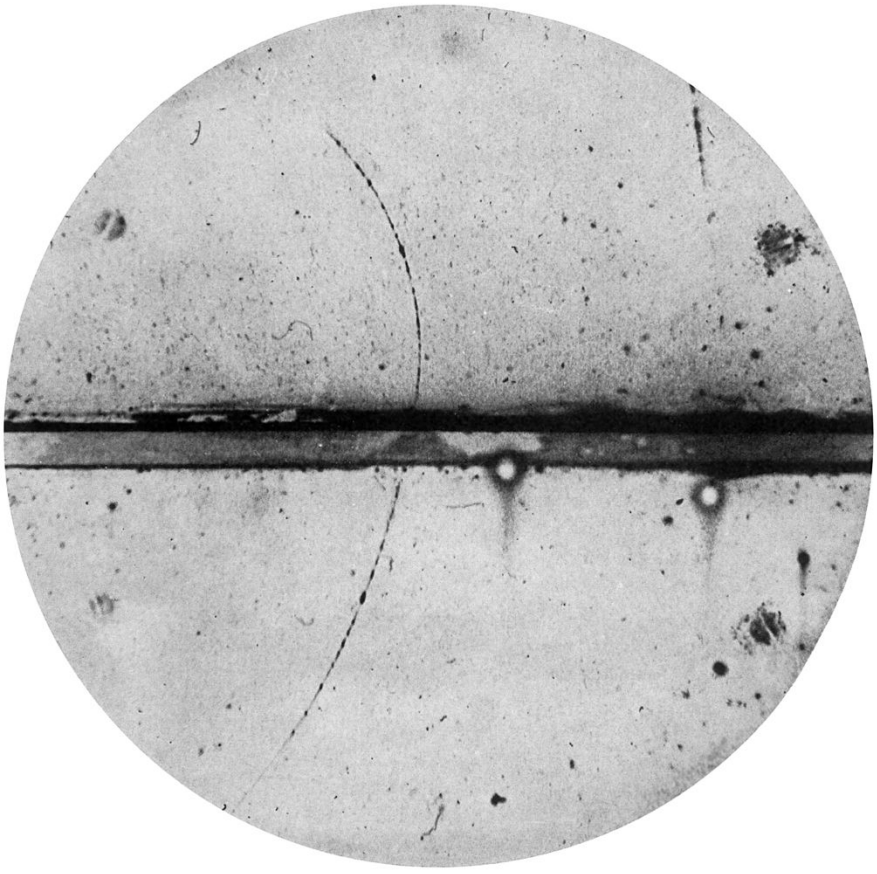


Figure 4.1: Track in a cloud chamber left by the first identified positron. Image from [7]

Tracking is discussed in this chapter, specifically for the LHCb experiment. Paths described by charged particles could be bent by the influence of the magnetic field produced by the magnet, where the amount of bending a particle experiments is a function of its charge

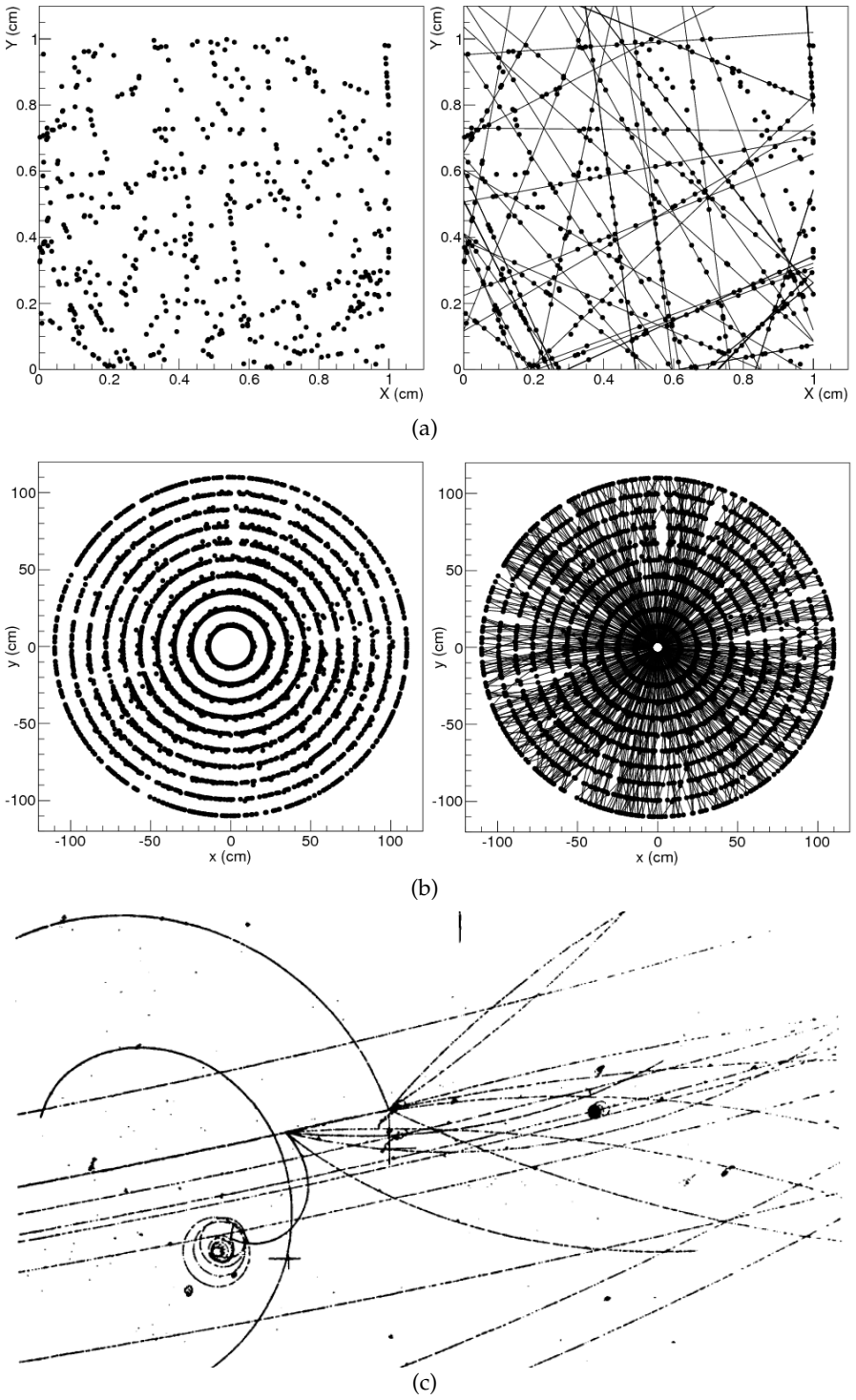


Figure 4.2: Various track reconstruction problems. (a) Crossing tracks in straight lines. (b) Curved tracks reconstruction. (c) Bubble chamber reconstruction. Images from [73] and [12]

and momentum. Early track reconstruction was performed by manually observing the behaviour of particles, or taking photos of the resulting tracks for later analysis. Figure 4.1 shows the first picture ever of a positron in a cloud chamber, taken in 1932. This was possible by using bubble chambers, which allowed to observe such particle interactions without the need of modern particle detection technology. A bubble chamber allows charged particles to produce bubbles in a superheated liquid, where their sizes are proportional to the energy loss of the particle. These bubbles can be photographed to highlight the particle path [68]. Various examples of track reconstruction problems are depicted in Figure 4.2. Figure (a) shows straight-line reconstructed tracks applying a Hough transform algorithm [51]. Figure (b) depicts the reconstruction of tracks bent by the influence of a magnet. Both figures present the hits extracted from the subdetector on the left and the result of applying a reconstruction algorithm on the right as tracks. Figure (c) shows an example of early track reconstruction using a bubble chamber [12].

The Hough transfrom was initially used for machine analysis of bubble chamber photographs.

Tracking at LHCb uses hit information from the subdetectors to compute all the positions and trajectories that are produced in each event, producing the tracks of the event. LHCb subdetectors provide x and y coordinates at various points in each subdetector. The z coordinate is given by the position of each subdetector; the direction in which particles travel is known as they travel from the interaction point to the rest of subdetectors. At LHCb there is no *time* coordinate as the events are considered to happen instantaneously, each event separated by 13 ms.

This chapter is structured as follows: In Section 4.1 track reconstruction and pattern recognition methods are discussed, Section 4.2 presents the track types and the subdetectors involved in tracking, Section 4.3 introduces the parameters to measure physics efficiency of algorithms, Section 4.3 presents the Kalman filter algorithm, and Section 4.5 gives a summary of this chapter.

4.1 TRACK RECONSTRUCTION AND PATTERN RECOGNITION

Track reconstruction in high-energy physics can be performed with diverse techniques [18], such as the Kalman filter or the Hough transform [144]. Finding tracks in events usually takes most of the time in this process, and its quality and performance directly impacts the computing resources needed to perform track reconstruction. Particle collisions produce new particles that are detected by the electronic detectors, and these are measured as hit coordinates. The inverse procedure, to go from hits to tracks, is done with pattern recognition methods; in general global and local methods are used [97].

Global methods take all hits of the detectors simultaneously and apply a similar procedure to all of them. This implies the resulting

solution of the method is independent from the order used to process the hits, which differs from local methods. Some examples of global methods are briefly described here.

Template matching are simple mathematical algorithms that can be applied when the number of possible patterns is finite. Each pattern can be defined with a template. The number of templates to check is always the same which makes the computing time needed for the algorithm independent from the complexity. On the other hand, when handling big complexity or high granularity problems, template matching cannot scale well. A tree-search algorithm can be applied for high granularity problems: a first search is applied with a coarse granularity, then successive higher granularity searches are used in the found patterns as seen in Figure 4.3.

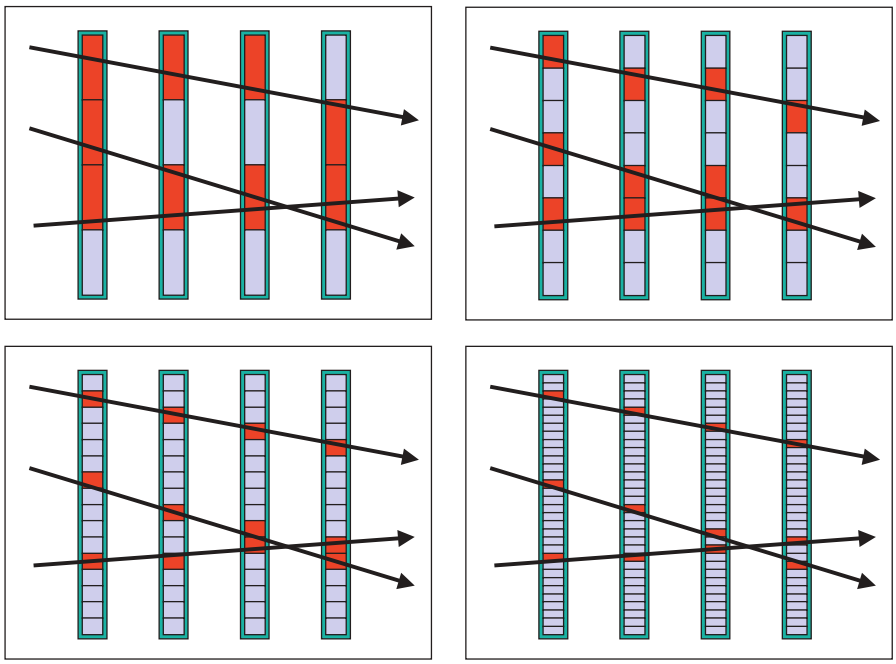


Figure 4.3: Global method, tree search. Image from [97]

Histogramming is used as a special case of the Hough transform [50]. To extract the features of various hits as the track that connects them, each hit is represented in parameter space as lines. These lines will intersect at some point in the parameter space which would reveal the parameters that correspond to the track the hits belong to, as depicted in Figure 4.4.

Lines would usually get very close in a reduced area.

Neural network techniques often look for global patterns [125] which makes them a good fit for these kind of global methods. A *Denby-Peterson method* [48] is briefly described here as an example of a neural network technique. Hits are connected through the neural network where a neuron is activated when two hits belong to the same track. The neuron activation is then defined by an energy function that gathers information such as the angle between the connected neurons

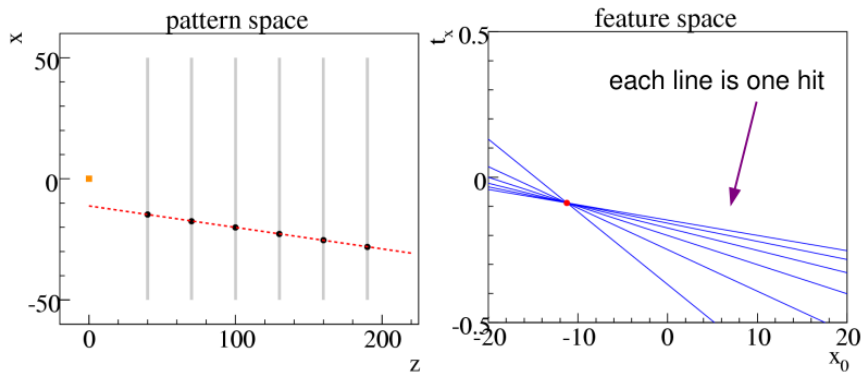


Figure 4.4: Histogramming with hits in a track. Image from [97]

or the number of neurons. The Denby-Peterson method works without the need of the track model; hits are connected as a straight line but curved lines are also detected.

Local methods differ from global methods in treating all hits the same way. These methods usually provide a way to extrapolate a trajectory and use a small set of hits to start the algorithm. Because local methods do not consider all hits equally these need to differ hits that can be *ghost* from real ones.

A small set of hits known as seed.

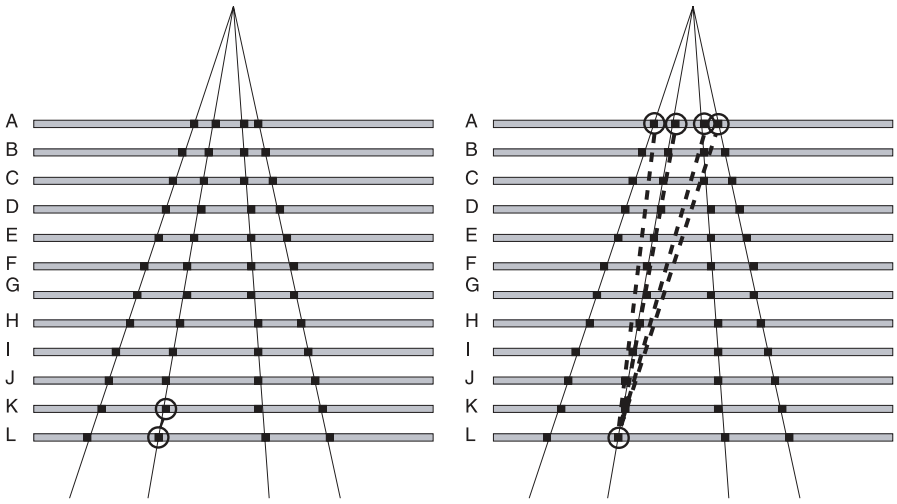


Figure 4.5: Two seeding techniques compared side by side. Image from [97]

Seeds algorithms create a small tracklet or segment with a small number of hits to start the tracking. Seeds can be started with different approaches as shown in Figure 4.5: the example in the left starts in the layer most distant from the vertex, where hits from different tracks have more distance between them, and then if uses the closest hit from the previous layer to form the seed; the right part of the figure uses hits from distant layers to create the seed which can give better precision but also increases the multiplicity. This kind of algorithm is commonly used with a *Kalman filter* as it follows a similar approach [15]. A

Seeds are also known as track following

Kalman filter provides a way to select good hits by reducing the uncertainty to select the next hit.

Naïve track following [97] approaches are used as a very simple method where a seed is taken as a starting point. The seed is extrapolated to the next subdetector where a hit is expected to be, and if a close hit is found, it is selected. This method presents problems in situations with high density of hits, as wrong hits can be selected and many ambiguities arise to select the next hit.

Combinatorial track following [97] uses a tolerance window at each step, generating a tree of candidates. This local method can give good results in selecting a true track, at the cost of high computational complexity.

Track propagation in a magnetic field presents the problem of tracks that are curved by a magnetic field; some magnetic fields are non-homogeneous or it is difficult to propagate a track analytically. For these cases various methods can be used to propagate a track, such as a parametrized extrapolation given the measurements of the magnetic field, or a Runge-Kutta method [19, 130].

4.2 TRACK TYPES AND SUBDETECTOR TRACKING

Different types of tracks can be identified in the LHCb experiment. These are classified according to the subdetectors they traverse. Figure 4.6 represents a top view of the LHCb experiment, where particles travel from left to right. The upper part of the figure shows the influence of the magnetic field across the z axis of the detector. The lower part of the figure matches the z coordinates of the tracking subdetectors, the magnetic field and shows a representation of the different tracks and how these are bent by the magnetic field. Tracks are classified as follows:

The acceptance region means the physical part of the subdetector where the particle can be detected.

- *VELO track*: It traverses the VELO subdetector only. If not matched to hits in other subdetectors, these tracks go out of the acceptance of these other subdetectors and thus are not detected by them.
- *Upstream track*: It traverses both the VELO and UT subdetectors, going out the acceptance of the SciFi subdetector. These tracks are also referred to as VeloUT tracks.
- *Downstream track*: It traverses the UT and SciFi subdetectors.
- *T track*: It traverses the SciFi subdetector but not the others. These tracks are not matched to other tracks.
- *Long track*: It traverses at least the VELO and SciFi subdetectors, and may have traversed the UT. These tracks give the most information being the longest, and are the most useful ones.

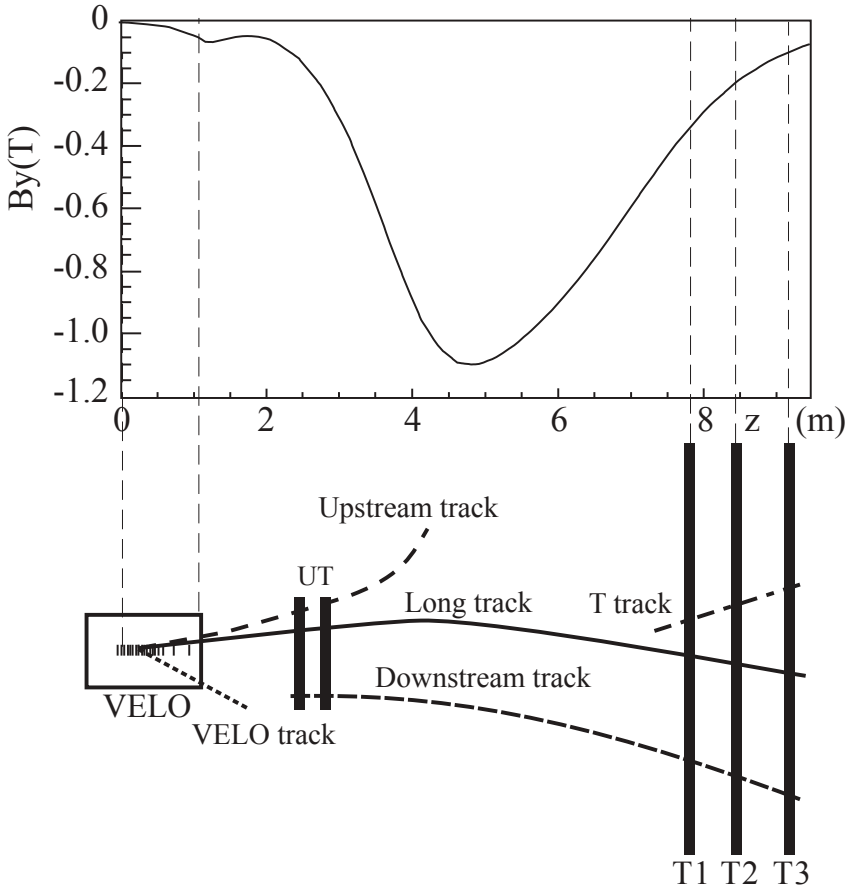


Figure 4.6: LHCb magnetic field influence and track types. Image from [39]

In the context of the LHCb experiment, long tracks play an important role as they traverse the full magnetic field and therefore have the most precise momentum information [136].

The interaction point where proton-proton collisions occur is surrounded by the VELO subdetector. Particles that result from these collisions originate within it, and the tracks are first reconstructed by the VELO. A percentage of these particles could go out of the acceptance range of the next tracking subdetector -the UT-, and the rest of them, in acceptance, produce hits in the tracking subdetectors. Right after the UT subdetector particles traverse the magnet, where its influence will bend charged particles proportionally to their charge and momentum. Its trajectory is then tracked by the SciFi subdetector to get a complete picture of the behaviour of the track from the collision point to the region after the magnet. With the hit signals from the subdetectors, the geometry information that defines them and the influence of the magnetic field at each point, particle tracking is performed.

A hit is produced with high probability, but a small fraction may not produce an activation signal.

Tracking is performed by both the HLT1 and HLT2, where both use the information from the same subdetectors to reconstruct the

particle trajectories. Differences in the algorithms used to perform tracking are defined by a trade-off between computing speed and physics accuracy. The HLT1 favours a fast reconstruction due to its synchronous mode of operation; HLT2 computes a more precise reconstruction with information from more subdetectors which is possible due to its asynchronous mode of operation. The general principles to reconstruct tracks at different subdetectors apply for both HLT1 and HLT2:

- *VELO tracking*: Uses the hits recorded by the pixel detector modules to reconstruct VELO tracks. These tracks are straight lines as the influence of the magnet is paltry in the VELO region. The vertices where the particles originate can be obtained through the VELO tracks. VELO tracking is performed first as the initial reconstructed tracks are used as input for the other tracking algorithms.
- *UT tracking*: Hits recorded by the UT planes are used to create tracklets. These tracklets are matched to the VELO tracks resulting in VeloUT tracks which can be bent paths as there is some influence from the magnetic filed in the UT region.
- *SciFi tracking*: Also called *Forward tracking*, the SciFi stations provide hits that are matched to both VELO and VeloUT tracks to create long tracks. Tracks reconstructed during the forward tracking are bent by the magnetic field as these fully traverse the magnet.

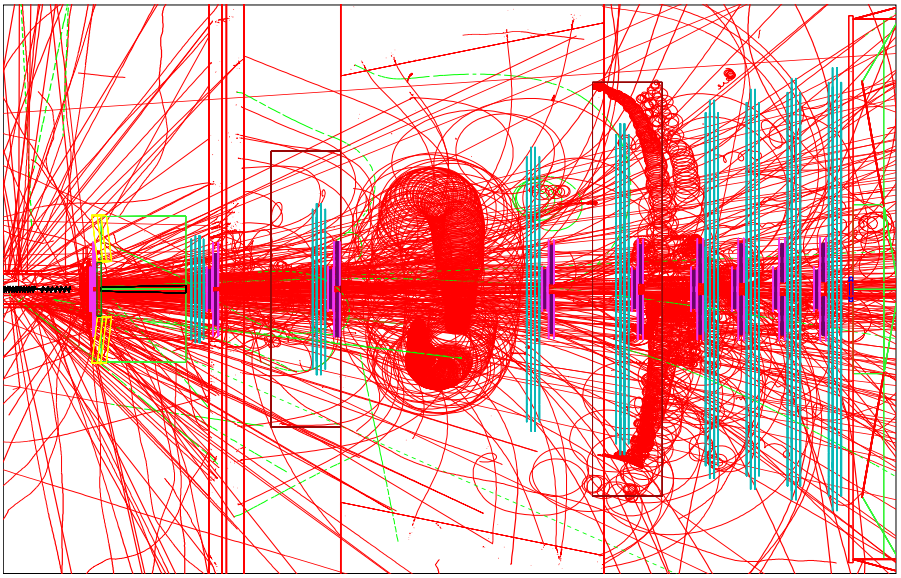


Figure 4.7: Track multiplicity. Image from [53]

Tracking under the influence of the magnetic field results in an explosion of combinatoric options for the UT and SciFi sub-detectors,

whereas the VELO reconstructs straight line tracks. Multiple matching hits need to be considered for each track. This situation is represented in Figure 4.7, where a real situation would involve hundreds of tracks per event, making the problem of finding matching hits an exponential combinatorics problem.

Tracking algorithms are expected to achieve high reconstruction efficiency, purity and hit efficiency with a low fake and clone rates for different types of tracks. The computing performance of the algorithms is determined by how many events per second can be processed for a given hardware configuration. This is a key aspect of event filtering in high-energy physics, specially for the LHCb experiment which will rely only on software for its event filter system. The combination of hardware and optimized software for it, will need to process the 30 MHz rate of events in real-time.

See Section 4.3 for physics efficiency definitions.

4.3 PHYSICS EFFICIENCY

When an algorithm is used to reconstruct a particle trajectory it will not always reconstruct all particles perfectly: some hits from other particle trajectories that are close may be used, some hits that belong to a particle may not be used or noise may interfere in the reconstruction. During algorithm development, testing and maintenance, simulated tracks are used to test the accuracy and *physics efficiency* of the algorithm. The simulated tracks are generated using Monte Carlo simulations which produce hits and data used to reconstruct the tracks, which later can be verified against them. When performing tracking at LHCb, a particle is considered to be *reconstructible* according to different criteria depending on the subdetector that performs the tracking algorithm:

- *VELO*: A minimum of three distinct hits were recorded in the VELO subdetector.
- *UT*: A minimum of one hit in one of the x layers and at least another hit in the u or v layers.
- *SciFi*: A minimum of six hits is needed. These hits must be distributed as one hit per station x layer, and at least one hit for each u or v layers.

Various parameters are measured to determine physics efficiency. Table 4.1 presents an example of these parameters for different track types: high reconstruction efficiency, purity and hit efficiency, explained below, combined with a low clone and fake rate are expected from a good reconstruction algorithm. These are verified by identifying each hit recorded during the event as follows [129]:

- *Track reconstruction efficiency*: It is measured with simulation data comparing the number of tracks correctly reconstructed against

Table 4.1: Physics efficiency indicators.

Track type	Reconstruction efficiency	Clone rate	Purity	Hit efficiency
Velo	43.16%	0.80%	99.44%	96.66%
Velo+UT	49.45%	0.80%	99.46%	96.65%
Velo+UT, p > 5 GeV	71.67%	0.86%	99.55%	97.46%
Long	60.34%	0.86%	99.51%	96.97%
Long, p > 5 GeV	72.62%	0.83%	99.56%	97.51%
Long from B	81.22%	0.62%	99.47%	97.39%
Long electrons	18.40%	2.76%	97.75%	95.11%

the number of tracks that are reconstructible. To be considered *reconstructed*, 70% of the hits on a track need to be associated to the particle from the Monte Carlo simulation. The reconstruction efficiency is given as:

$$\frac{N_{\text{reconstructed \& reconstructible}}}{N_{\text{reconstructible}}}$$

- *Clone rate*: When two or more tracks are associated to the same Monte Carlo particle, only one is considered to be reconstructed correctly and the others are counted as *clones*. The clone rate is the number of clone tracks relative to all reconstructed tracks. The clone rate is defined as:

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

- *Fake rate*: A track is considered fake when it is reconstructed but it cannot be associated with a Monte Carlo particle. Fake tracks are also referred to as *Ghost tracks*. The fake rate is defined as follows:

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

- *Purity rate*: It is determined by the ratio of number of hits from the reconstructed particle that are found in the real particle, and the number of hits of the reconstruced particle.

$$\frac{N_{\text{hits from real particle}}}{N_{\text{hits in reconstructed particle}}}$$

- *Hit efficiency rate*: It is determined by the ratio of number of hits from the reconstruced particle that are found in the real particle, and the number of hits of the real particle. It is the ratio between the hits that correspond to the real particle and the hits in the real particle:

$$\frac{N_{\text{hits from real particle}}}{N_{\text{hits in reconstructed particles}}}$$

We refer to physics efficiency to describe how good a tracking algorithm performs, analogous to a cost function that uses the described parameters, reconstruction efficiency, clone, fake, purity and hit efficiency rates. There is no analytical form of such cost function, where an algorithm is said to attain good physics efficiency if the reconstruction efficiency, purity and hit efficiencies are high, and the clone and fake rates are low.

4.4 KALMAN FILTERING

Tracking at LHCb experiment performs a last fitting step applying a Kalman filter to the track to reduce the error associated with it. The Kalman filter [86] is a linear quadratic algorithm used at LHCb experiment to estimate the particle trajectories as they travel through the detector. The tracking subdetector provides position with an associated error as inputs for the algorithm. The Kalman filter is one of the main time-contributors of the LHCb baseline in the HLT1.

A Kalman filter receives measurements as input, typically over time, which contain an amount of error and noise. It is widely used in GNSS, guidance and navigation [71, 138], but it can be applied to any system with uncertainty in its measurements which is also dynamic. An educated guess about the system is needed to be able to predict the next step of the system.

GNSS systems such as Galileo, GPS or GLONASS [78]

At LHCb the Kalman filter does not receive measurements over time but over the z axis of the detector, as the events are not measured with a time variable. As particles travel from the interaction point to the subdetectors the measurements can be sorted and thus the Kalman filter can be applied.

The Kalman filter prediction, measurement and output is a *state vector* $\vec{x}$ and a *covariance matrix* P for every iteration. Each iteration in the LHCb Kalman filter is a hit in a tracking subdetector at a given z axis position. The *state vector* $\vec{x}_z$ contains information about the particle for position (x, y), slope (tx, ty) and charge over momentum (q/p):

$$\vec{x}_z = (x\ y\ tx\ ty\ q/p) \tag{4.1}$$

Each measurement of a *state* is assumed to be random and Gaussian distributed, where an error is associated to the measurement in the form of a *covariance matrix* P_z that indicates the relationship between the mean values (the most likely ones) of the *state* μ, and the variance they could have σ^2. The covariance matrix is always symmetric and it is described as follows:

The initial values for the covariance matrix depend on the accuracy of the sensors. Covariance matrix values change during the predict and update stages.

$$P_z = \begin{bmatrix} P_{x,x} & P_{x,y} & P_{x,tx} & P_{x,ty} & P_{x,q/p} \\ P_{y,x} & P_{y,y} & P_{y,tx} & P_{y,ty} & P_{y,q/p} \\ P_{tx,x} & P_{tx,y} & P_{tx,tx} & P_{tx,ty} & P_{tx,q/p} \\ P_{ty,x} & P_{ty,y} & P_{ty,tx} & P_{ty,ty} & P_{ty,q/p} \\ P_{q/p,x} & P_{q/p,y} & P_{q/p,tx} & P_{q/p,ty} & P_{q/p,q/p} \end{bmatrix} \tag{4.2}$$

4.4.1 *Predict stage*

The Kalman filter is composed of two stages, the *Predict* and *Update*. To predict how a *state* $\vec{x}_z$ in the next position of the tracking detectors will be, a *Transport matrix* F_z is used. The *state* $\vec{x}_{z-1}$ is multiplied by the *Transport matrix* F_z to calculate the next *state* $\vec{x}_z$. If every value of the *state* is multiplied by a matrix we apply the following property:

$$\begin{aligned} &Cov(x) = P \\ &Cov(F\,x) = F\,P\,F^T \end{aligned} \tag{4.3}$$

Considering this property, the *state* and *covariance matrix* calculations are predicted like the following:

$$\begin{aligned} &\vec{x}_z = F_z \vec{x}_{z-1} \\ &P_{z+1} = F_z P_{z-1} F_z^T \end{aligned} \tag{4.4}$$

The previous equation does not take into account the external influence that may alter the measurement with elements that are not related to the state. For the LHCb Kalman filter the magnet creates a magnetic field which will affect the charged particles. This known external influence is modelled as a *transport vector* calculated with a *control vector* $\vec{u}_z$ and *control matrix* B_z. This vector is added to our prediction step to correct the known external influence. For the model to take into account the unknown external influence or noise, a *noise matrix* Q_z is appended as a covariance matrix for the system to model the noise influence.

The final *prediction step* with the *noise matrix* lays as follows:

$$\begin{aligned} &\vec{x}_z = F_z \vec{x}_{z-1} + B_z\,\vec{u}_z \\ &P_z = F_z P_{z-1} F_z^T + Q_z \end{aligned} \tag{4.5}$$

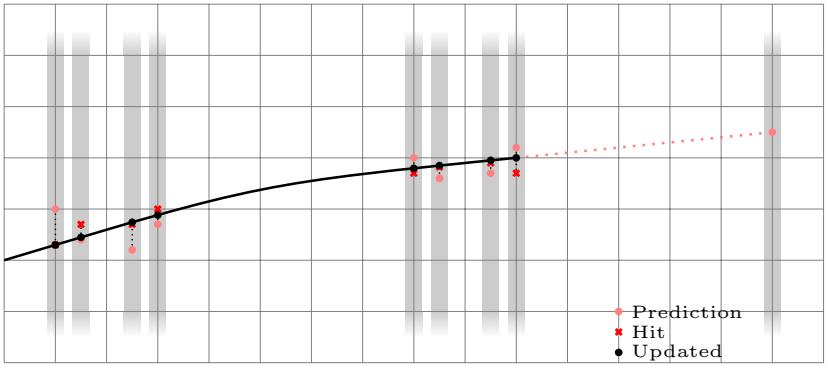


Figure 4.8: Kalman predict stage track representation

The *prediction step* in track reconstruction extrapolates the position where the particle will be at some z position in the direction were the particle is traveling. This is represented in Figure 4.8. Previous predictions are represented by a light red dot, and the hit measurement position of the hit is represented by a red dot. Some measurements will be very close to the actual prediction, and the final real position is determined by Kalman filter in the *update stage*.

4.4.2 *Update stage*

The prediction stage results in a *state* and *covariance matrix* that uses the previously calculated value $(z-1)$. The update stage calculates the final *state* and *covariance matrix* values that will server as input for the next iteration by correcting the predicted ones. It combines the predicted value with the read given by the sensors of the detector, and gives more weight to one part or the other through the *Kalman gain matrix* K_z. This step takes into account the measurements from the sensors, which are modelled with a *Projection matrix* H_z that relates the sensor reads to the *state*, giving the range of value this could take. This matrix is applied to the *state* and *covariance* as follows:

$$\begin{aligned} \vec{x}_{expected} &= H_z \, \vec{x}_z \\ P_{expected} &= H_z \, P_z \, H_z^T \end{aligned} \tag{4.6}$$

For the update stage a measurement *pre-fit residual* $\tilde{y}_z$ is calculated. It uses the measurements, *Projection matrix* and observation noise v_z as inputs to calculate it as follows:

$$\tilde{y}_z = v_z + H_z(x_z - x_{z-1}) \tag{4.7}$$

To give weights to the prediction and the measurements, a *Kalman gain matrix* K_z is calculated. To calculate it, a pre-fit residual covariance

S_z is calculated. It combines the covariance of the *sensor measurement* R_z, the *Projection matrix* H_z and predicted covariance matrix P_z so that:

$$\begin{aligned} CHT &= P_z \, H_z^T \\ S_z &= H_z \, CHT + R_z \\ K_z &= CHT \, S_z^{-1} \end{aligned} \tag{4.8}$$

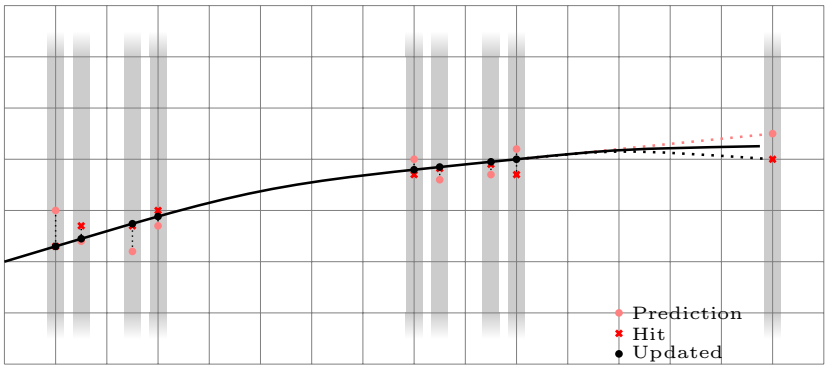


Figure 4.9: Kalman update stage track representation

By putting it all together we get the *update step*:

$$\begin{aligned} \vec{x}'_z &= \vec{x}_z + K_z \, \tilde{y}_z \\ P'_z &= P_z - K_z \, H_z \, P_z \end{aligned} \tag{4.9}$$

The *update step* is represented in Figure 4.9. The prediction previously calculated is represented as an extrapolation with the light red dots. The actual hit measurement, read as an input, and used during the *update step* to be combined with the prediction and pondered by the Kalman gain, is represented as an extrapolation with the red dot. The real position of the particle is given in the end of the *update stage* and usually results in a different position from those predicted and measured but that more accurately represents the real position where the particle crossed in that z position.

The final flow diagram from the equations that involve the *predict* and *update* stages is represented in Figure 4.10. Here the *prediction stage* is represented at the top of the Figure where the input from the previous updated node is depicted in blue. This node is then used to calculate a prediction status and covariance matrix depicted in pink. The input used to calculate is represented by the letters of the matrices, with the *transport* F_z, *control matrix* B_z, *control vector* $\vec{u}_z$ and *noise matrix* Q_z represented by pointing to the predict stages were these are used: for the status or covariance matrix. The *update stage* is depicted in the lower part of the figure, were the just calculated status and covariance

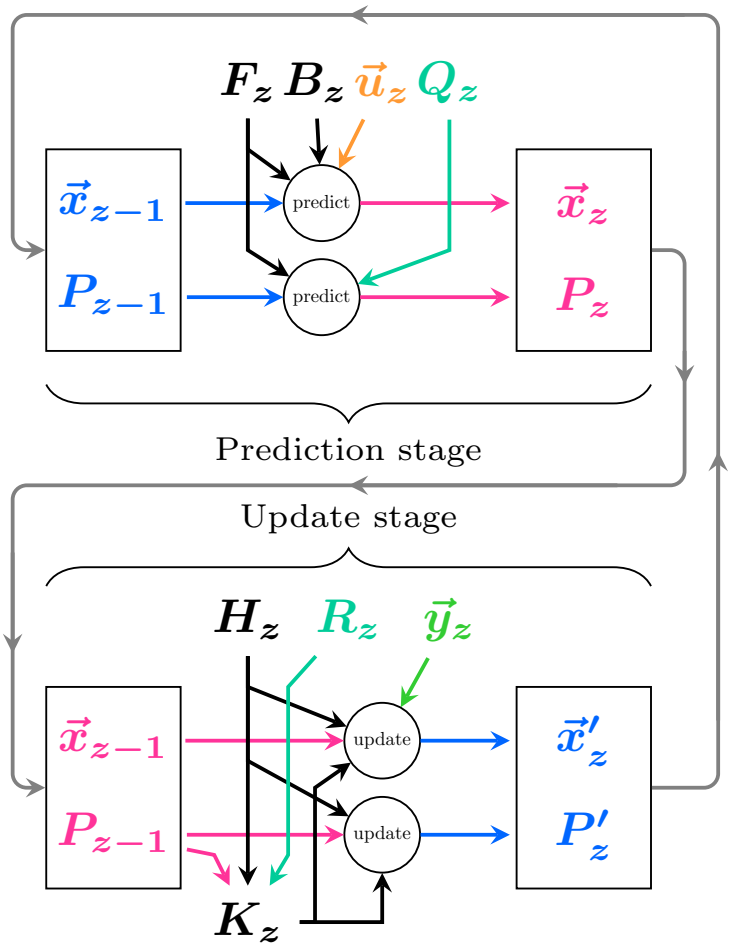


Figure 4.10: Kalman predict and update stages relationship

matrix of the prediction are used as an input to produce the updated status and covariance result that will be used in the following *predict* and *update*. The *Projection matrix* H_z is used in combination with the *sensor measurement* R_z to produce the *Kalman gain matrix* K_z which is used in both the status and covariance matrix update with the *pre-fit residual* $\tilde{y}_z$ also used for the status calculation.

4.5 SUMMARY

This chapter presented an overview of track reconstruction, or tracking, at the LHCb experiment. Different algorithms and methods are introduced, which are used to obtain the resulting tracks from an event; some are better suited for more specific types of reconstruction problems and different results are achieved. The different types of tracks that can be identified at LHCb are shown, and the indicators used to measure the efficiency of different algorithms are explained.

These tracks and indicators vary between the different subdetectors involved in providing hits for the reconstruction algorithm. Finally the Kalman filter is explained with more detail as one the main topics of this thesis: this algorithm is used to produce more accurate tracks at the expense of extra computing time, and it is used in different stages of the reconstruction process.

Part II

PARALLEL COMPUTING

5

PARALLEL COMPUTING

Parallel computing is often defined as the ability of one or more computers to process independent computations *simultaneously*. At a hardware level, parallelism is found in different forms: various independent, network-connected computers can cooperate to compute tasks in parallel; a multi-core processor contains a number of cores capable of processing tasks simultaneously; accelerators can be attached to computers to work in parallel; vector processors; and single cores in a chip use instruction pipelines and superscalar pipelines to compute instructions in parallel.

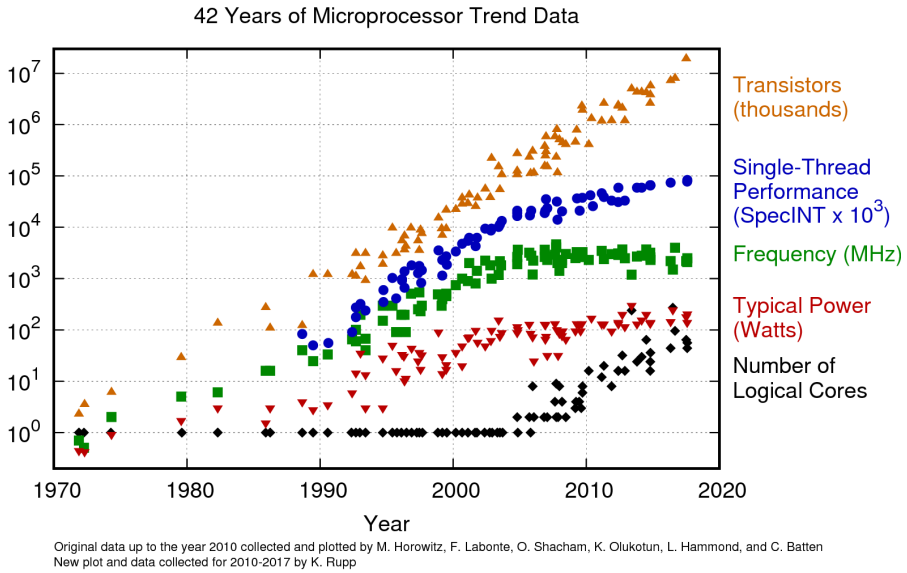


Figure 5.1: Microprocessor trend over the years. Image from [127]

The motivation to implement more parallelism into chips is related to *Moore's Law* [107], which states that processor manufacturers are able to put double the number of components per integrated circuit every 18 months. This prediction still holds true as depicted in Figure 5.1. During the first years processor manufacturers increased the clock frequency to continue gaining performance until a point were the temperature and power required to run at higher frequencies was not worth it, *the free lunch is over*. Processor manufacturers continued doubling the number of transistors every 18 months, but the clock speed and power consumption flattened or even lowered in favour of multi-core architectures. Figure 5.1 also shows how manufactures changed the number of cores put into processors after the frequency and power stagnated. The number of transistors continued to increase due to the improvements in the litography process. With the presented

From the 70s to around 2004

H. Sutter phrased this in 2005 [133]

limitations, processor manufacturers started using the increased number of transistors to put more cores and slightly reduce the processor frequency. This is well represented by Graphics Computing Units (GPUs) which present a high number of processing cores with a lower clock frequency, but are able to deliver higher Floating Point Operations per Second (FLOPS). The same tendency is explored with CPU many-core processors, such as the Intel Xeon Phi that uses the x86 architecture to deliver high core count with reduced clock frequency to offer higher FLOPS, as explained in Section 5.2.2.2.

Multi- is often used for tens of cores. Many- is used for the hundreds.

Multi- and many-core processors opened new possibilities for software programs to exploit this type of parallelism to run programs faster by dividing them into smaller pieces. To predict how much faster a program could run, various methods where developed which are described in Section 5.1.

This thesis focuses on designing and implementing software algorithms for parallel architectures in the context of high-energy physics. Its research focuses on using the resources offered by different hardware architectures efficiently to achieve high throughput and maintainability with parallel technologies. Algorithms can be designed, changed and optimized around the exposed architectural characteristics of the different platforms to exploit their performance. The main goal of the real-time LHCb computing is to deliver a combined event throughput of 40Tb/s for a given hardware configuration. This can be achieved with different programming languages, frameworks and technologies; but the chosen hardware architecture limits the available options and programming models that can be used.

Good physics performance must be achieved, which could improve with new algorithm design.

This chapter is structured as follows: In Section 5.1 speedup and scalability concepts are introduced, Section 5.2 discusses different parallel architectures concepts including memory and accelerators, and Section 5.3 gives a summary of this chapter.

5.1 SPEEDUP AND SCALABILITY

A parallel architecture allows to compute a task in less time by dividing the work, and processing the divisions in parallel. The amount of time that can be saved by parallelizing tasks, and processing them with a parallel architecure can be expressed in various ways and depend on various factors that are explained in this chapter. To measure the performance of a computation usually the time it takes to complete is used, which allows to compare different computations and determine which one is *faster*. To compare how much faster a program is, we compare them in terms of *speedup*:

Throughput, data transmission or latency are other ways of measuring performance.

$$\text{Speedup} = \frac{\text{Time}_n}{\text{Time}_m} \tag{5.1}$$

Equation 5.1 expresses the relative performance between two systems solving the same problem, each with different resources. n and m are the number of parallel processors or resources available to compute the same problem. Typically $m > n$ and $n = 1$ is used to express how much faster a system with m parallel resources is compared to the sequential one, but other speedup options can be used to compare. The *scalability* of a system is determined by its ability to achieve good speedup when using more parallel resources.

Amdahl's law [6] gives an upper bound to the scalability a system could achieve. Under this law, a program's execution time can be divided into two parts: the *non-parallelizable, sequential work* and the *parallelizable work*. Given n number of processors to compute the program, and being p the parallelizable portion of the program, Amdahl's law can be depicted in Equation 5.2:

$$\text{Speedup} \leqslant \frac{1}{1 - p + \frac{p}{n}} \tag{5.2}$$

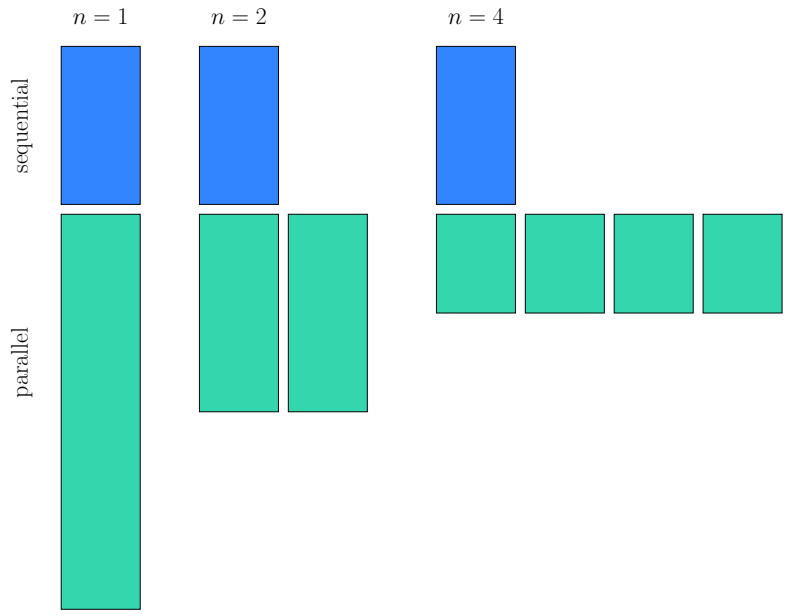


Figure 5.2: Amdahl's law representation

Figure 5.2 represents how Amdahl's law limits how fast a program could run, no mater how many parallel resources are given to the work. Even on an infinite number of processors, the sequential part will dominate the computation.

John Gustafson's view proposes a solution to this problem through *Gustafson-Barsis' law* [72]. Rather than viewing the problem to solve as a fixed-sized problem, Gustafson proposes a law were the parallel part of the work increases as more resources are put into solving it, as shown in Figure 5.3. The time to solve the problem remains

rather constant while the work size increases, which is formalized in Equation 5.3:

$$\text{Speedup} \leqslant 1 - p + np \tag{5.3}$$

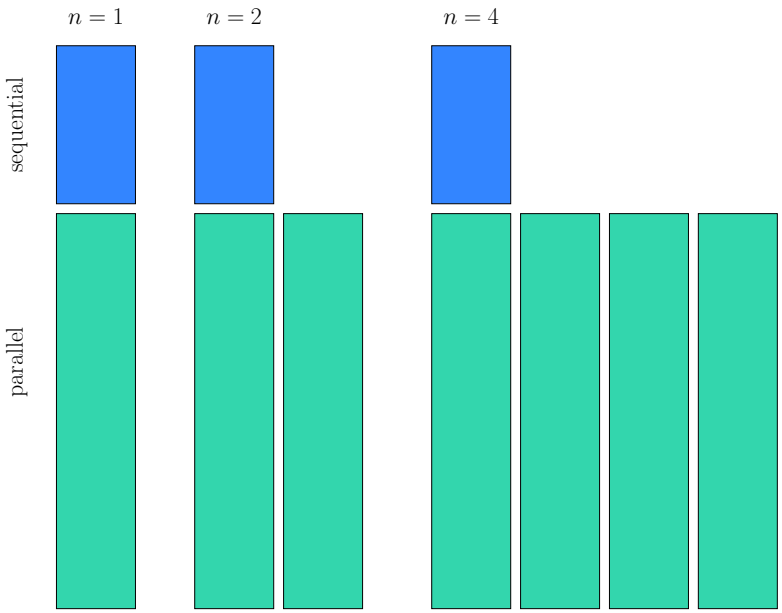


Figure 5.3: Gustafson-Barsis' law representation

As in shared memory programming.

Both laws apply to different problems. To calculate the speedup that a single computer node can achieve, Amdahl's law better predicts the achievable speedup. For systems composed of various nodes or systems where resources can be expanded to distribute the work, Gustafson-Barsis' law should be used.

A different way of predicting speedup for the work-span model [132] offers a more realistic prediction for parallel programs. This model considers that the parallel part of a work cannot be parallelized *perfectly*. It divides the work into tasks and describes the relationship between them as a graph, as can be seen in Figure 5.4. The important part of this model is to reveal the *critical path* of the graph, the shortest path through the nodes to complete the work. The length of this critical path determines the achievable speedup. If *work* is the number of tasks that define the graph, and *span* is the critical path of the graph, the work-span model speedup is given by:

$$\text{Speedup} \leqslant \frac{work}{span} \tag{5.4}$$

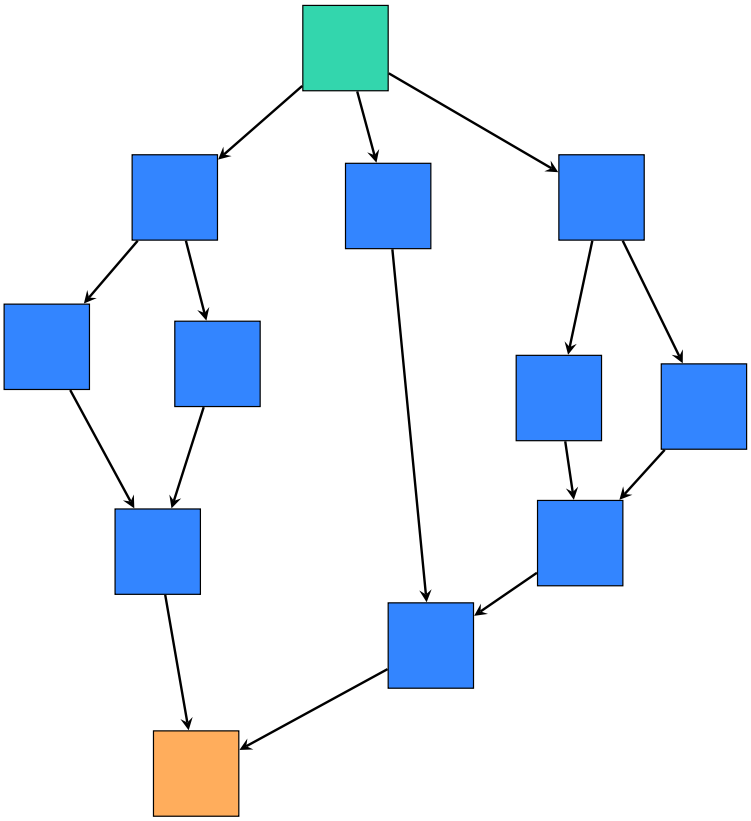


Figure 5.4: Work-span model representation

5.2 PARALLEL ARCHITECTURES

Parallel computing architectures can be classified based on how its control flow and data management are arranged. Flynn's taxonomy [62] characterizes various types of architectures as follows:

- SISD *(Single Instruction, Single Data)*: a processor with no parallel capabilities that computes data sequentially, a scalar processor.
- SIMD *(Single Instruction, Multiple Data)*: a processor that can handle various data streams in parallel, applying the same instruction to different data. Instructions are executed sequentially.
  - SIMT *(Single Instruction, Multiple Threads)*: it is a tiled SIMD type that uses masking to compute in groups of threads. Although these architectures expose hundreds or thousands of threads, these blocks of threads actually share control units and work similarly to other SIMD machines. This is typically found in Graphics Processing Units (GPUs), and it is discussed more in detail in Section 5.2.2.1
  - SPMD *(Single Program, Multiple Data)*: is a higher level abstraction compared to SIMD. Various programs run si-

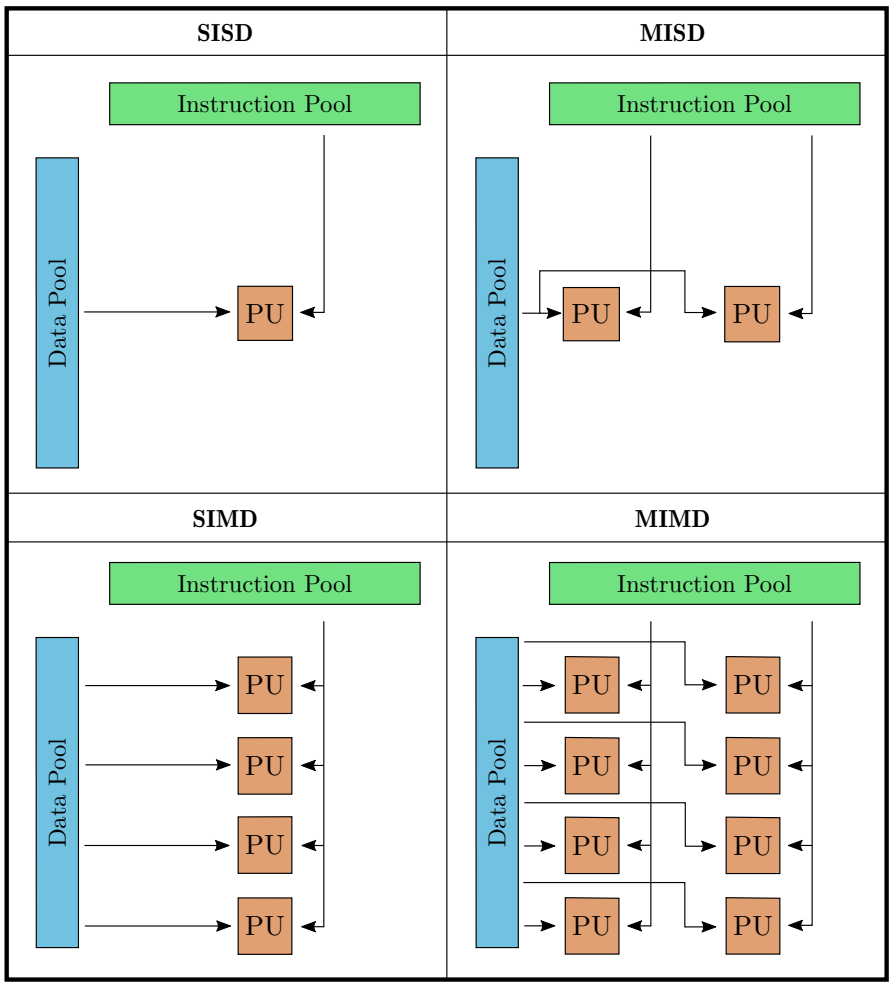


Figure 5.5: Flynn's taxonomies

multaneously on different data. Both are not mutually exclusive, and as the Intel Implicit SPMD Program Compiler (ISPC) [116] shows, the SPMD model can be used to parallelize with the vectorization units.

- MISD *(Multiple Instructions, Single Data)*: different instructions are applied over the same data stream. It is not very used, and when used its typically for fault tolerance.
- MIMD *(Multiple Instructions, Multiple Data)*: the most common type of parallelism, where multiple cores or threads or distributed computers apply different computations over different data, simultaneously. This is found in multi-core processors and distributed systems.

This architecture classification and parallelism levels are based on Hennessy & Patterson literature [77]

This architecture classification is depicted in Figure 5.5. Modern computers are typically SIMD and MIMD combined machines that exploit parallelism in both ways. Parallelism on these architectures

can be exploited at different levels: instruction, data and thread-level parallelism. A fourth level of parallelism can be included when different computers cooperated connected through a network: request-level parallelism, but it is out of the scope of this thesis. These are briefly described here:

- *Instruction level parallelism (ILP)*: This kind of parallelism benefits from the fact that modern processor circuitry expose multiple funcional units that can operate in parallel, inside a single core. It measures the number of instructions that can be executing at the same time. Modern architectures are already limited by the level of ILP that can be reached, where the major limiting factor are the *data dependencies* between instructions which force the instruction to run sequentially. Different techniques are applied to achieve it: *superscalar execution* utilizes multiple functional units from the processors core to execute different instructions simultaneously; *instruction pipelining* benefits from the division of instructions into various steps to overlap these steps. Pipelining allows to deliver more instructions per unit of time without executing them in parallel.

- *Data level parallelism (DLP)*: When multiple different data can be provided to the processor, and the processor has dedicated hardware functional units such as *vector units*, DLP can be exploited. It allows to execute the same instruction over different data in a single instruction; the speedup that can be achieved depends on the number of data units that can be provided to the hardware. Processors with vector units and Graphics Processing Units (GPU) 5.2.2 are examples of hardware that can exploit this type of parallelism.

Ofter refered to as vectorization *or* SIMD.

- *Thread level parallelism (TLP)*: It allows programmers to schedule units of execution that progress concurrently or simultaneously depending on the hardware and software constrains. Two major types of TLP can be identified: *hardware multithreading* occurs when various threads are executed on the same processor and compete for the hardware resources inside the processor; *multiprocessing* schedules the threads to run in different processors where the execution is separated by the hardware.

A specific optimization that combines both ILP and TLP is *Simultaneous multithreading* or *SMT*. SMT exploits TLP to try use the functional units not being used by a thread scheduling a different thread to use them. Two threads can then run in parallel, to a certain degree, inside a single core or processor. SMT is highly dependent on the workload that exploits it and how resources will compete between them [135]. Major processor manufacturers use this technique to improve performance, for instance Intel implements it under the name *Hyperthreading*

Simultaneous multithreading (SMT) is all about keeping superscalar CPU units busy by converting thread-level parallelism (TLP) to instruction-level parallelism (ILP). For applications with high TLP and low ILP, SMT makes sense as a performance optimization. Pekka Enberg.

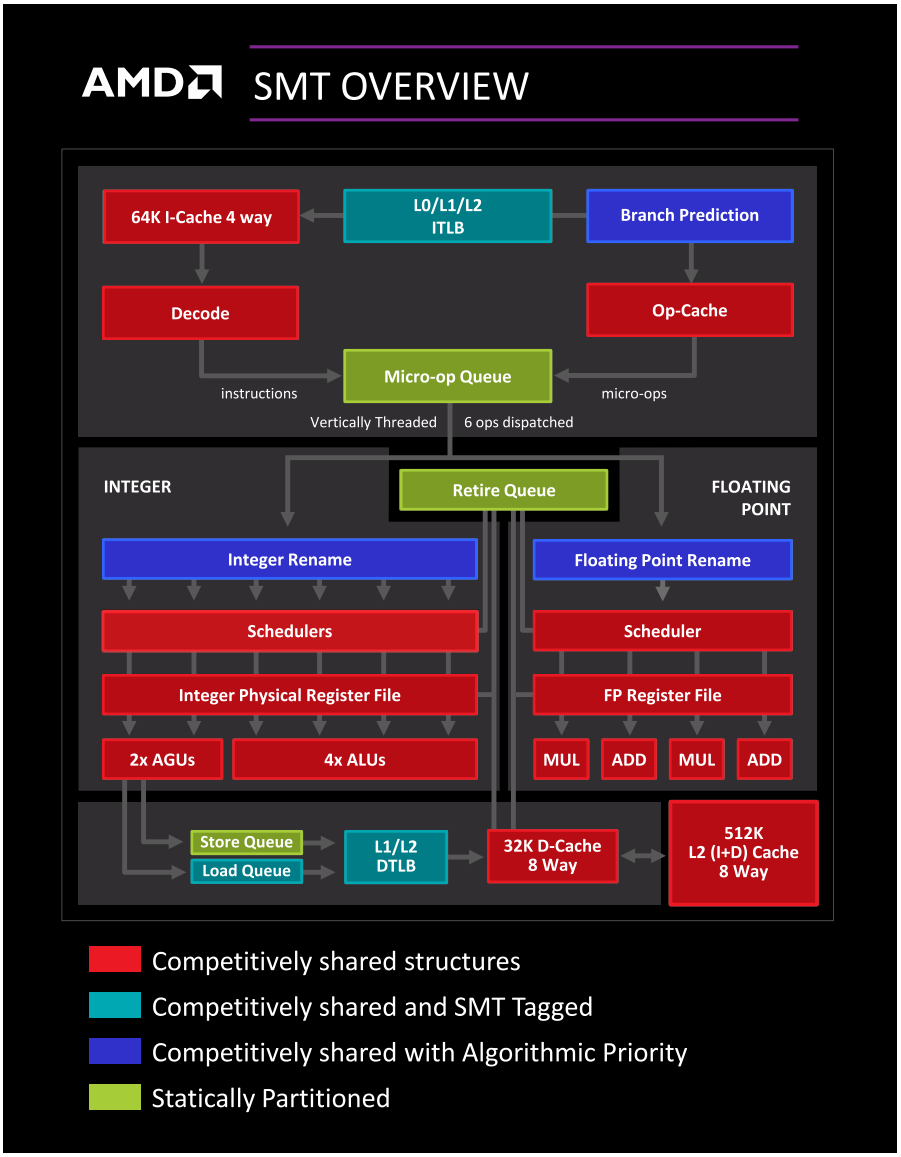


Figure 5.6: AMD implementation of SMT in Zen microarchitecture. Image from [34]

with up to 4-way SMT in the Intel Xeon Phi, IBM implements up to 8-way SMT with its POWER8, and AMD implements 2-way SMT on its Zen architecture, as can be seen in Figure 5.6. In this figure, two threads could run in parallel by using different resources that are competitively shared.

5.2.1 *Memory*

Modern computers present various levels of memory; these are organized in a hierarchical manner. This hierarchy is designed to minimize the effect of the memory wall [112]: as memory capacity grows larger, its latency is increased and dominates computations. To address

this problem a hierarchy of different sized memories is implemented, where the smaller ones are closer to the processor, presenting orders of magnitude smaller latencies, as depicted in Figure 5.7. Caches minimize the memory wall problem but still present problems: these use a percentage of the silicon chip area, and their effectiveness greatly depends on the *locality of reference*.

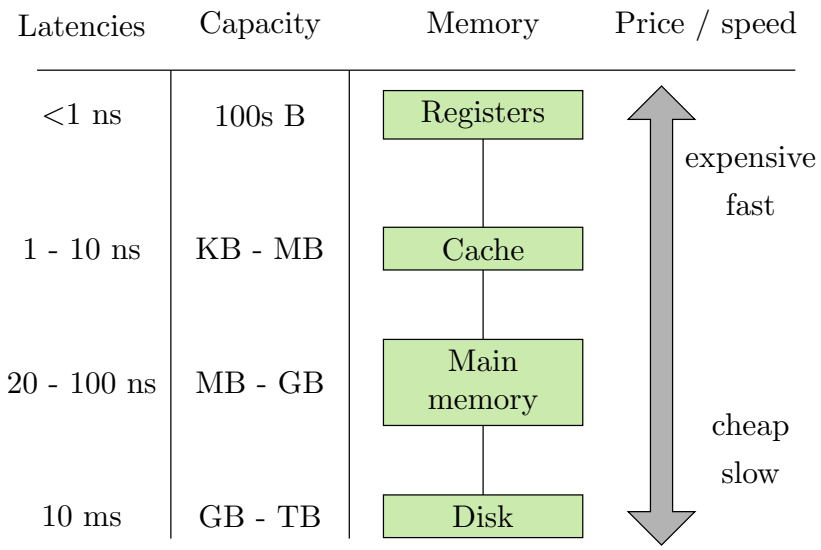


Figure 5.7: Memory hierarchy and its latencies.

The locality of reference indicates that when a processor accessed certain memory locations, it will probably access those locations again. If an application is designed with locality in mind, the probability that those locations are accessed more frequently increases. These diffeernt levels of memory store these locations that the processor *predicts* will be accessed, and if it succeeds it will take less latency to get that data to the processor.

Computer architectures can be categorized based on its memory organization with respect to its processors:

- UMA *(Uniform Memory Access)*: all processors accessing the memory are equally distant from it. This limits its possible speed, but simplifies reasoning about how will memory be accessed. It also limits the possible total bandwidth of the system, as its access occurs through the same channel for all processors.
- NUMA *(Non Uniform Memory Access)*: part of the memory is locally closer to one of the processor, so its access to memory will be faster compared to the others. The memory access time to certain memory location will be different depending on the processor that access it. Applications implemented with this constraints in mind could be faster, but non NUMA aware applications could potentially be slower. With NUMA more memory

controllers are available which increases the possible bandwidth of the system.

According to the memory communication model, computer architectures can be organized in:

- *Shared memory*: all processors use the same virtual memory address space, and memory can be directly accessed.
- *Message passing*: processors access to different virtual memory address space, being limited to directly accessing its own memory.

Multiple data is processed in the form of independent events with hundreds of tracks.

LHCb HLT will focus on parallel data processing architectures (SIMD and MIMD) given the nature of the challenge where multiple different data needs to be processed in real-time. Both UMA and NUMA architectures are being considered, where the NUMA architectures such as the Intel Xeon Phi or the IBM Power processors need of specific optimizations to leverage them. Single nodes are optimized as *shared memory* machines, leaving the communication between the nodes out of the scope of this thesis. Single nodes can be optimized independently as each of them can accommodate hundreds of independent events.

5.2.2 *Accelerators*

Accelerators are used in computing for different specific purposes; all of them share the goal of doing some computation faster by taking advantage of the specifics of the hardware they were designed for. There are accelerators for graphics like Graphics Processing Units (GPUs), Digital Signal Processors (DSPs) for signal processing or general algorithms with Field Programmable Array Gates (FPGAs). For this thesis we focus on GPUs and the Intel Xeon Phi accelerators.

GPUs are also used for general purpose computing, GPGPUs.

5.2.2.1 *GPUs*

GPUs are widely used in scientific and high performance computing for their massively parallel architecture and support for floating point operations. These specialized processors have reduced clock speed compared to CPUs, but are able to include more processing cores on the chip. This design favours high-throughput computing, which is used for some types of computations. A modern scientific GPU like the NVIDIA Tesla V100 features a clock speed of 1.3 GHz, but offers more than 5120 compute units. Its architecture is depicted in Figure 5.8 [84].

The NVIDIA V100 features the Volta architecture.

A GPU like the V100 features dozens of *Streaming Multiprocessors (SM)* as shown in Figure 5.8. These are grouped into Texture Processing Clusters (TPCs) which are then grouped into GPU Processing

Figure 5.8: NVIDIA V100 Full GPU. Image from [110]

Clusters (GPCs). Pairs of memory controllers are connected to the High Bandwidth Memory slots and to the L2 cache. Figure 5.9 shows the architecture of each SM. Memory units are shown in blue where the L1 cache is shared for four processing blocks, and registers and L0 cache is individual to each. Control units warp scheduler and dispatch unit manages threads by assigning resources to the compute units in green. Data transfer between cache and main memory is processed by the load and store units, with special functional units for operations such as square roots or cosines. Compute units are depicted in green, and these divided for floating point in different precisions, integers and tensor cores for machine learning training [52].

GPUs have been used before in the field of high-energy physics with success. The ALICE experiment at CERN implemented track reconstruction in GPUs obtaining different speedups compared to the previously used hardware [123]. We note how the approach we follow is different than the one implemented in ALICE, as we aim to implement the full *High Level Trigger* to run in GPUs, including the decoding and tracking of all subdetectors, thus avoiding much of the needed data transfer between main memory and GPU memory. Other HEP experiments have seen significant improvements when using GPUs to amend the performance of online selection [25, 131], or using a common code base to target both CPUs and GPUs using OpenCL, which shows the performance improvement of GPUs while supporting the x86-64 architecture [64].

Other high-throughput scientific fields, such as DNA sequencing, have improved their computing performance with GPUs. The *Arioc*

Figure 5.9: NVIDIA V100 SM. Image from [110]

read aligner showed how using parallel algorithms with GPUs improved DNA sequencing throughput, achieving an order of magnitude faster alignments [141, 142]. Pawar et al. benchmarked various DNA sequencing algorithms with different GPU-based tools against a CPU one; concluding that GPUs will replace CPUs in DNA sequencing for its higher-throughput processing [114]. Other DNA-related fields exhibit similar speedups: Samsi et al. [128] demonstrated how a single GPU is able to compare millions of DNA samples in seconds, Cadenelli et al. [26] compared offloading a genomics workload into FPGAs and GPUs from a CPU, resulting in the GPU outperforming both, although the GPU consuming more energy.

Other scientific fields benefit from high-throughput, real-time processing in GPUs. Radio telescopes need to filter data in their data acquisition systems; where software frameworks employing GPUs like *Bifrost* [44] have shown significant performance improvements. Other real-time radio telescope experiments studied the viability of using GPUs, where they encountered large computing speedups at a local level, but were limited by I/O when using multiple GPUs [96]. Others in the same field have successfully implemented GPU optimization schemes [80] achieving a 6× speedup compared to the CPU scenario, or used a GPU-based software framework and aggressive optimizations to be able to process data rates close to 1Tbit/s, like the *CHIME Pathfinder* radio telescope [119].

GPUs have also been studied in scenarios requiring real-time processing at fusion experiments [94] greatly reducing the wall-time compared to the CPU version. Real-time split-and-merge executions have been improved in multi-GPU scenarios by Han et al. [74], and X-ray computer tomography reconstruction in GPUs has shown how different optimizations can be implemented and combined to speedup GPU computations [17].

5.2.2.2 *Intel Xeon Phi*

The Intel Xeon Phi KNL 7210 self-boot processor offers significant improvements in scalar and vector performance over its predecessor Knights Corner. It is especially well-suited for highly parallel workloads thanks to its high bandwidth memory and wide Vector Processing Units (VPUs) [83]. Specifically, it is equipped with up to 72 Airmont (Atom) cores, each of which comprises two 512-bit vector units capable of executing AVX-512 SIMD instructions. This processor can be considered a many-core architecture as compared to other x86 processors it presents a higher degree of parallelism with more cores, SMT threads and vectorization units. The chip is thus able to deliver a theoretical peak double precision floating point performance of up to 3 TFLOPs. It is important to note that each core offers 4-way Simultaneous Multithreading (SMT), also known as Hyper-Threading. Finally, the KNL supports up to 384 GB of DDR4 RAM and 8-16 GB of Multi-Channel DRAM (MCDRAM), which provides a bandwidth of 400 GB/s.

As illustrated in Figure 5.10, the cores are interconnected in a 2D mesh of tiles, each one integrating two cores that share a 1 MB L2 cache. In order to exploit the resources provided by this architecture, this mesh can be configured in one of three cluster modes.

- *All-to-all* mode treats all available memory (DRAM and MCDRAM) as a single virtual NUMA node. With this mode, the OS is not aware of the underlying NUMA architecture, so memory cannot be explicitly allocated to specific physical nodes.

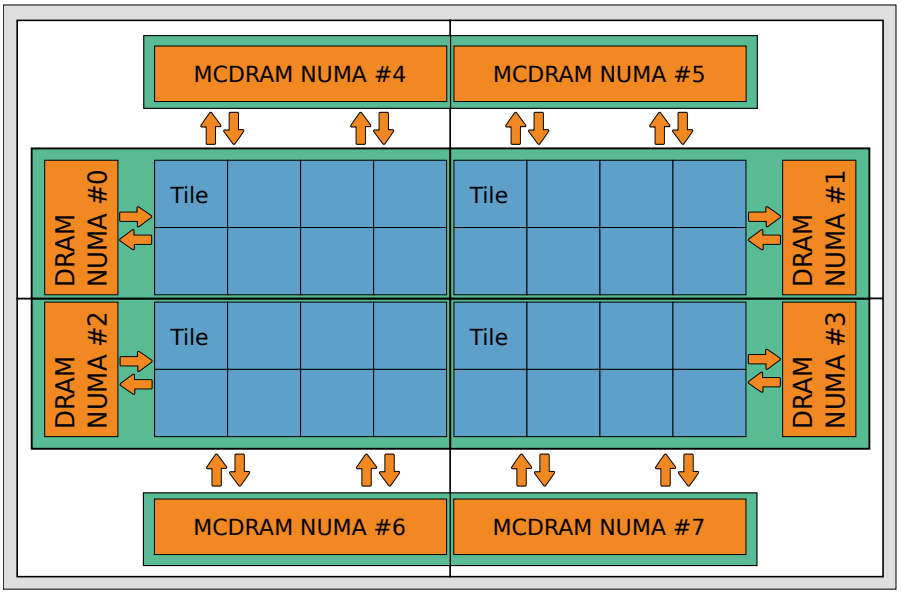


Figure 5.10: Intel Xeon Phi Knights Landing SNC-4 cluster mode.

- *Quadrant* mode divides the mesh into four virtual quadrants, but they are exposed to the OS as a single NUMA domain. With this mode, software does not need any further optimization in order to take advantage of the architecture features.
- *SNC-4* mode also divides the mesh into four quadrants. However, each of them is presented as a different NUMA node to the OS. This setting offers lower latencies but the software needs to be NUMA-aware to benefit from it.

Figure 5.10 shows the SNC-4 cluster with Flat memory configuration. This mode creates 4 clusters with equal number of tiles (two core units) each. Every cluster groups two NUMA nodes (surrounded by green boxes) where one is the high-bandwidth memory MCDRAM, and the other contains both the cores and the DRAM memory. Cores are represented with tiles, and the memory in orange boxes.

Furthermore, the KNL architecture has the ability to be configured in one of three memory modes: cache, flat or hybrid. Cache mode treats the MCDRAM as a traditional last-level cache (LLC), which is used transparently by the cores on the NUMA node. In contrast, the flat mode, both DRAM and MCDRAM are on separated NUMA nodes. Hybrid mode splits the MCDRAM in half, hosting 8 GB for LLC and the remaining 8 GB for an independent NUMA node, combining cache and flat mode.

5.3 SUMMARY

This chapter gives an overview of the parallel computing aspects relevant to this thesis. It covers the motivation to create parallel

hardware and software that exploits it, and describes different ways to measure the gains that can be obtained by parallelizing the work. The main types of parallel architectures and parallelization levels are shown, including how memory can be organized both at a physical and logical level. Finally two main types of accelerators are described as part of the heterogeneous computing that involves highly parallel processors: GPUs and the Intel Xeon Phi, both used extensively in this thesis.

Part III

PARTICLE TRACKING IN HIGH-ENERGY PHYSICS

6

KALMAN FILTER OPTIMIZATION FOR HEP WORKLOADS

Various Kalman filter implementations are used at LHCb for different stages of the reconstruction algorithms. Both HLT1 and HLT2 use a Kalman filter at the end of their reconstruction chain; once a track is reconstructed with the hits from all the subdetectors, the Kalman filter goes over all the hits in the track. Kalman filters are also used as part of the reconstruction algorithms of a subdetector, i.e. implementations of the VELO reconstruction often use a simplified Kalman filter as part of the process. As part of the HLT2, a vectorized implementation was developed which is used for developments and optimizations in this thesis [28]. This Kalman filter algorithm is implemented in the LHCb framework with the name *TrackVectorFitter* and provides a SIMD optimized version for LHCb reconstruction. A standalone implementation of the algorithm is available under the name *cross-kalman*, which provides a way to test the vectorization capabilities across different hardware architectures. This version implements a simple parallelization scheme where each event is processed in a different thread.

This chapter is composed by two main sections:

- Section 6.1: The implementations and optimizations performed to a reference Kalman filter implementation to parallelize it intra-event, targeting the SMT capabilities of the Intel Xeon Phi processor.
- Section 6.2: A version using generic parallel patterns to simplify its parallelization, simulate a better real-world scenario of real-time data-taking, and exploit the Intel Xeon Phi characteristics with the generic parallel patterns implementation.

Some parts of this chapter have been published in the following journal/conference papers:

- Placido Fernandez Declara et al. «A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures.» In: *IEEE Access* 7 (2019), pp. 91612–91626. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927261

6.1 SMT MULTI-THREAD KALMAN FILTER FOR INTEL XEON PHI

High-energy physics software commonly benefits from an *embarrassingly parallel* distribution of tasks: there are multiple events per second

to process and each event produces multiple particle trajectories. Processing these workloads involves multiple independent inputs in the form of different events, particles or tracks, which allows to process them independent from each other. The LHCb experiment processes millions of collisions every second, and in the previous runs each event was processed in parallel as a simple but effective way of distributing the work for the available hardware. As events get larger and more collisions are recorded per second after each upgrade, opportunities for intra-event parallelization will allow for a better optimization of modern hardware with multiple parallelization levels. Inside each event, dozens to hundreds of particle tracks are to be reconstructed, where each track can be processed independently from each other. At a lower level, the algorithms used to reconstruct these tracks expose different opportunities for parallelization which can be exploited.

Kalman filters are a naturally sequential problem; the next node can not be processed until the result from the previous one is computed. This is the case for problems where nodes of the Kalman filter are added with each measurement, like a moving object that keeps updating its position, often in real-time. At LHCb all track measurements from the detector for each event are received as if these happened at the same time; there is no time coordinate to be considered. This particularity allows to process the first, last and all measurements in between at the same time, which presents opportunities to distribute the work in a parallel way.

Various levels of parallelization are identified in this thesis for LHCb's Kalman filter. At a higher level it can be divided into three stages: *forward*, *backwards* and *smoother*. Depending on the implementation, the operations performed in the *backwards* and *smoother* stages can be combined to form a single step for computing efficiency reasons, but the same mathematical operations are performed. These stages correspond to the processing of a full track in one direction, and can be described as follows:

- *Forward*: The Kalman filter is applied along the track in forward direction, from the collision point to the rest of the subdetectors.
- *Backwards*: The Kalman filter is applied to the same track in the opposite backwards direction, from the latests subdetectors (Muon subdetectors) to the collision point.
- *Smoother*: The resulting states from the *forward* and *backwards* fits are combined to further reduce the error of the track.

This steps can be represented in Figure 6.1, where the forward stage would process a track from the VELO to the SciFi Tracker, the backwards stage would process the whole track in the opposite direction from the SciFi Tracker to the VELO, and the Smoother stage can be process in any direction by combining the two. By performing two fits

in opposite directions and combining the measurements, the resulting error for the track can be further reduced at the beginning and end of the track. If only one fit in one direction was to be performed, the initial covariance matrix which presents a higher uncertainty would result in the first hits having a higher error. During the *forward* step the fitted nodes are saved for later use and computations. While the *backwards* step is processed, as soon as a resulting node is computed it can be already *smoothed*. This optimization can be applied for computing performance reasons in some implementations.

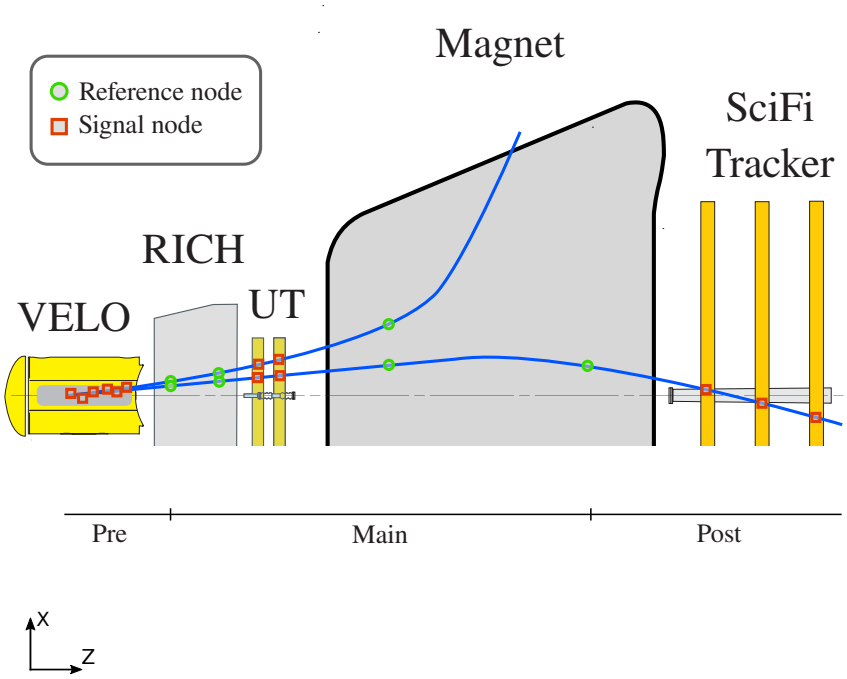


Figure 6.1: Reference hits and signal hits with sections of a track.

The input for the Kalman filter are the reconstructed tracks that traversed the tracking subdetectors. The selected hits from the VELO, UT and SciFi subdetectors are used as the input track the Kalman filter receives. The VELO and UT subdetectors are separated about one meter with the RICH being placed in between them. The distance between the UT and the SciFi amounts for over four meters, accommodating the magnet in between them. This distance between subdetectors increases the uncertainty of the filter to correctly predict and update the next hit position in the track. In some implementations *reference hits* are introduced in the regions in between tracking subdetectors to mitigate the effects of the error in the Kalman filter as depicted in Figure 6.1. Hits triggered by the tracking subdetectors are called *signal hits*, whereas the *reference nodes* are virtual hits for large track sections that do not have a subdetector signal. This distinction allows to differentiate sections of the track where different computations are applied. When processing a track it can be divided into sections that

are processed differently, where this sections are identified by the *reference hits*.

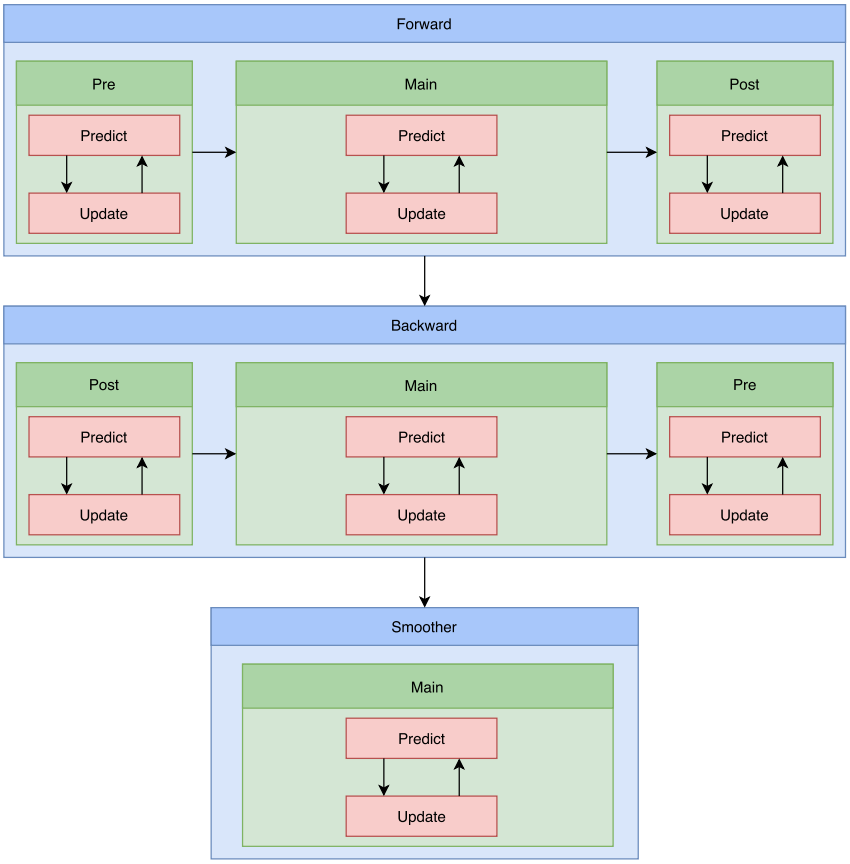


Figure 6.2: Kalman filter overall steps, with the forward, backward and smoother, each with various predict and update steps

The processing along the track differs depending on the section of the track. A different math is applied to the first section of the track compared to the main and last ones. This presents three versions of the Kalman filter that can be separated and scheduled in a different manner. To differentiate these sections of the track, the reference and signal hits are used: reference hits are virtually placed where a hit would have been triggered if a tracking subdetector was to be found in that region; signal hits are the actual hits where tracking subdetectors are placed. Three sections of the track are distinguished by the placement of these hits, as depicted in Figure 6.1:

- *Pre section*: It includes the signal hits until the first reference hit is found.
- *Main section*: It includes all reference and signal hits until the last reference hit is found.

- *Post section*: It is composed by the remaining hits from the last reference hit.

This logical separation for the track can be applied in both forward and backwards directions depending on the computations being applied. The overall steps are shown in Figure 6.2. These steps allow for parallelization opportunities inside a single track and potentially introduce speedups.

The finest grain of parallelization can be achieved by splitting the *predict* and *update* steps that are found in every Kalman filter, as explained in Section 4.4. Inside each track and *pre*, *main* or *post* section, various nodes are processed following the equations that involve these two final steps. The *update* step cannot be computed without the result of the *predict* step, when both the prediction and the measurement are combined to give the final results for the node.

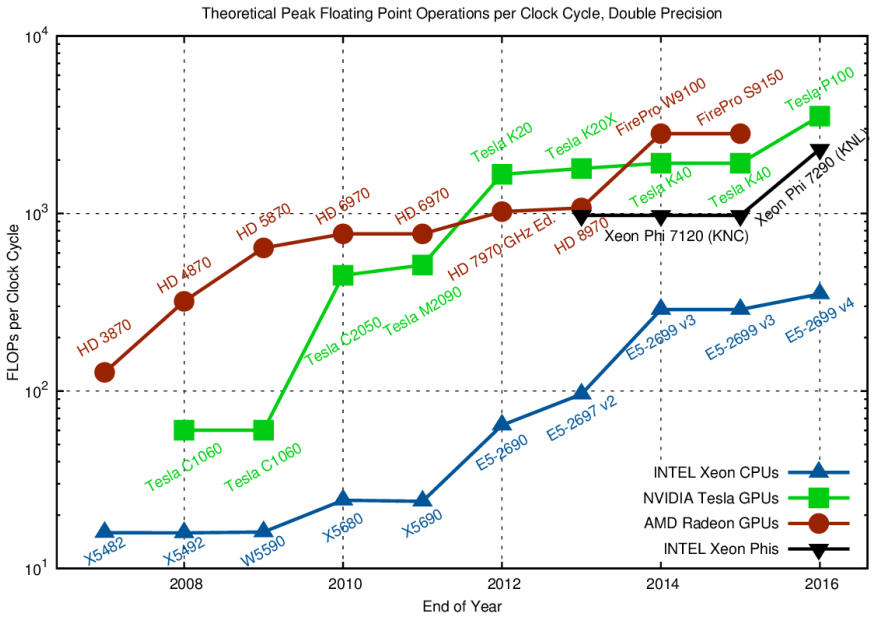


Figure 6.3: Intel Xeon Phi, Intel Xeon, NVIDIA Tesla and AMD Radeon architectures FLOPS compared. Image from [126]

As there are thousands of events per second, each containing dozens or hundreds of tracks, the Kalman filter processing of every track in real-time is a good candidate for highly-parallel architectures, such as the Intel Xeon Phi presented in 5.2.2.2. The lower clock frequency of its architecture allows to introduce a higher number of processing cores, which increases the maximum theoretical peak floating point operations per second (FLOPS) compared to a standard Intel Xeon architecture. This architectural change brings the floating point performance closer to the ones found in GPUs, as seen in Figure 6.3. This figure shows the evolution in terms of FLOPS for the major GPU architectures, the Intel Xeon and Intel Xeon Phi architectures. The Intel Xeon Phi is situated below the top scientific NVIDIA GPU, the

Tesla P100, but closer to it in comparison with the same year Xeon processor: the E5-2699 v4.

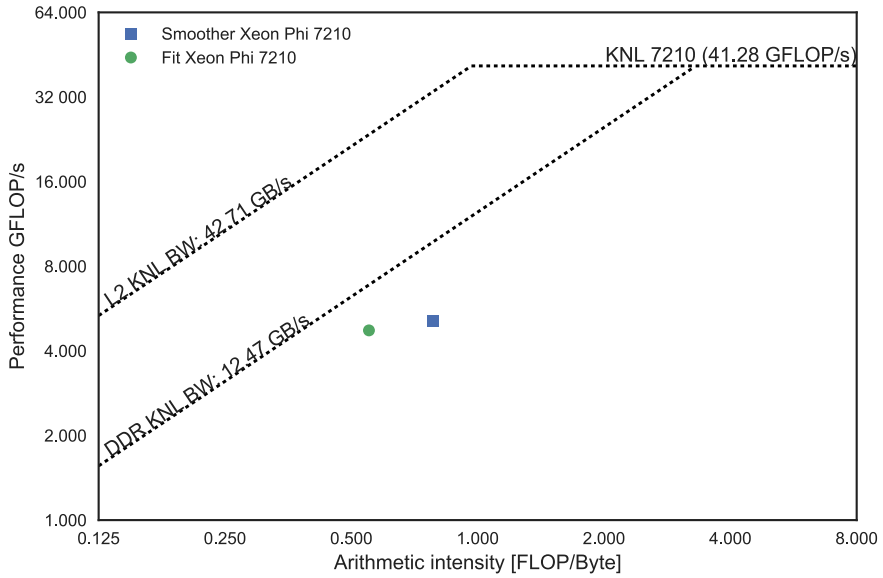


Figure 6.4: Intel Xeon Phi roofline plot. Image from [115]

The *cross-kalman* implementation has already vectorization optimizations implemented, exploiting processor vector lanes efficiently. A *Roofline model* [140] of this Kalman filter implementation for the Intel Xeon Phi indicates that while this implementation is close to the optimal achievable performance, there is still some room for improvement, as depicted in Figure 6.4. This plot shows the performance of the application being run and its arithmetic intensity on each axis. The arithmetic intensity is the number of FLOPS the application computed for the amount of bytes needed to compute each FLOP, showing how memory bound an application is. The plot also shows the maximum performance that an application can reach for that processor as lines that start diagonal and become horizontal: for a given type of memory, the diagonal shows how many FLOPS can be achieved if the accesses for that memory are always successful. The horizontal line indicates a compute limitation of the processor, where the actual algorithm needs to be changed to improve. For the application to further improve, three approaches can be followed: maximizing the in-core performance, maximizing bandwidth or minimizing traffic.

Types of memory are indicated as L2 or DRAM for example.

A different insight into the algorithm performance can be analyzed with its scalability. As shown in Section 5.2.2.2, the Intel Xeon Phi contains between 64 and 72 cores with 4 *hyperthreads* each. This is reflected in Figure 6.5 where the application shows great scalability while using the cores, but throughput diverges more from the *perfect achievable speedup* once the hyperthreads start being used. Although this behaviour is expected because the hyperthreads need to compete for resources, an optimized use of the SMT capabilities of the processor

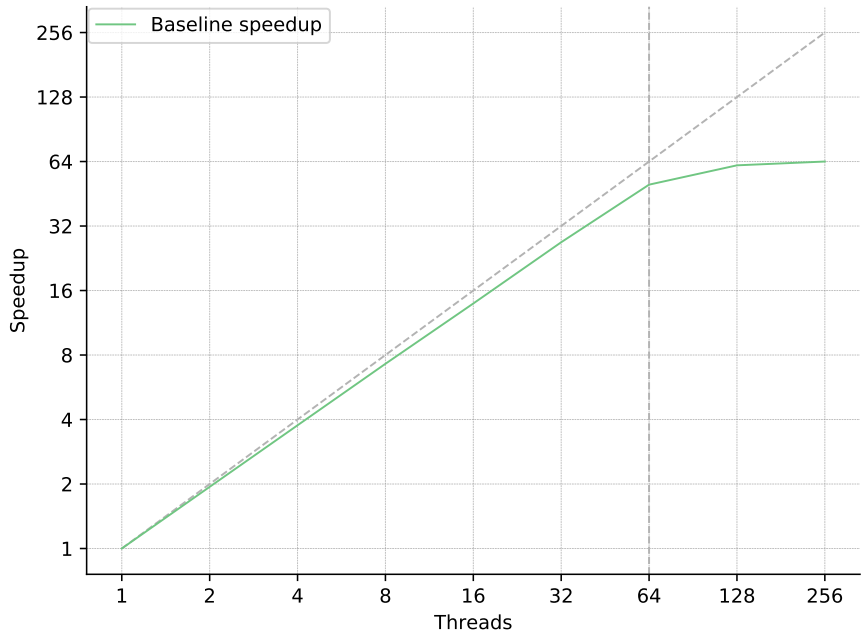


Figure 6.5: Intel Xeon Phi - Kalman filter scalability plot.

can lead to a good usage of the hypethreads so these continue scaling well.

Different parallelization schemes are implemented to further parallelize the Kalman filter at different grain levels. As shown in Figure 6.2 the highest level of parallelization consists on running *Forward*, *Backwards* and *Smoother* in parallel. Dependencies between the steps exist: the *Smoother* cannot be computed without the result from both the *Forward* and *Backwards*; the *Backwards* step needs the result of the *Forward* computations in this implementation. These can be processed in parallel by using a pipeline pattern, where with the continuous input of tracks the different stages are processed in parallel.

The following subsections describe the implemented parallelization schemes. The implementations go from finer grained to coarser grained tasks, as different levels of parallelization are identified. All the parallelization schemes are pipeline patterns, as all the identified steps depend on a previous step, being the sequential nature of the Kalman filter with its two stages, *predict* and *update*, the hits that compose a track which need to be processed one after the other, the sections of the track, or the overall procedure where a track is processed first in one direction, then the opposite and finally a combination of the two. These pipelines are implemented using Intel TBB for the parallelization, which includes the pipeline as one of its patterns. All schemes are based on a already parallel, vectorized and hand-tuned version of the Kalman filter.

6.1.1 *Predict-Update pipeline*

At the finer grain level, the *predict* and *update* steps both require from different computations and are separated steps. The *update* step presents a dependency of the *predict*, where a two-stages pipeline can be used to be filled with the hits from the track. This pipeline is represented in Figure 6.6 showing how the two stages of the pipeline are filled with various tracks. A pipeline parallel pattern allows to run both stages in parallel with a constant input of hits from the tracks, to feed the two-stage pipeline. The SMT capabilities of the processor can be exploited by implementing this pipeline. Tracks are scheduled to run in parallel at the core level, each with core its own L1 cache. As there are 4 hyperthreads per core and a 2-step pipeline is implemented, pairs of hyperthreads per core will be scheduled to run the pipeline.

Pairs of cores share a L2 cache.

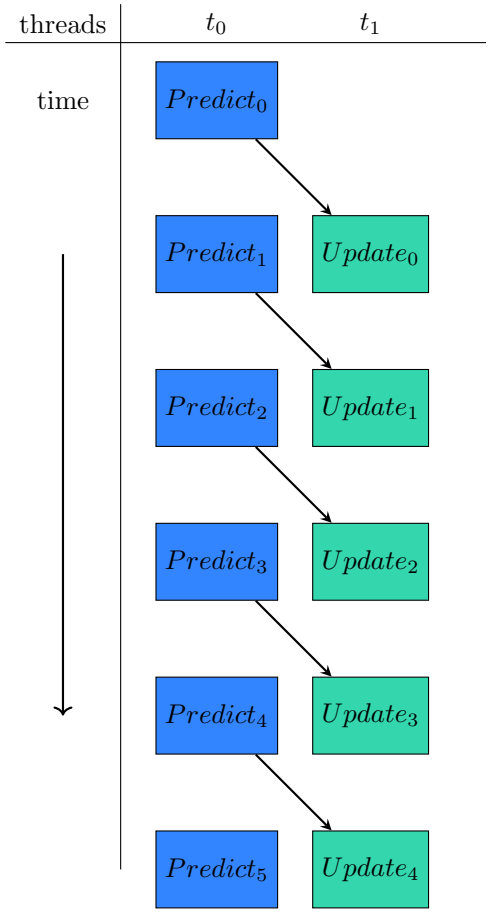


Figure 6.6: Predict - Update pipeline

To test the performance of the different implementations an Intel Xeon Phi 7210 Knights Landing is used. It is configured in *Flat* memory mode and *Quadrant* cluster mode. All benchmarks where compiled with `gcc 6.2.0` and used C++ threads for multithreading.

As input for the Kalman filter, Monte Carlo simulated events are used, processing 100000 events for each benchmark.

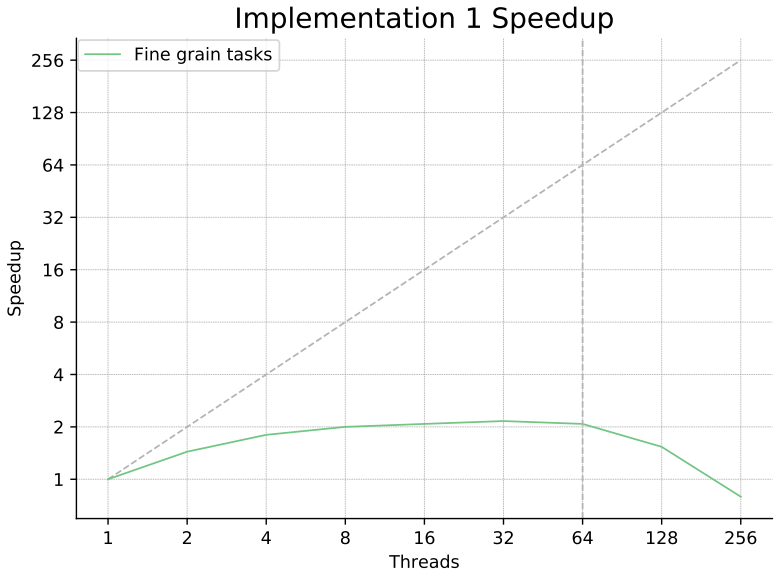


Figure 6.7: Predict - Update parallelization speedup

The results from this implementation are depicted in Figure 6.7. The scalability shows a slight improvement of performance when using the physical cores which flattens between 32 and 64 cores; using more cores does not yield an improvement in throughput from 32 cores. The scalability shows the degradation in performance when the hyperthreads are scheduled to be used. For this implementation multiple parallel two-stage pipelines are used for the input tracks, where *predict* and *update* stages for the same track run in the same core. In the theoretical best scenario where a total of 256 threads are used, two pipelines for different tracks run in the same core using the four hyperthreads. This impacts both the L1 and L2 cache performance: the L2 cache is shared by *tiles* in the Intel Xeon Phi, each *tile* containing a pair of cores, where a total of four tracks compete for cache resources without sharing any data between them. The L1 cache is affected in the same way as two different tracks are processed per core. The degradation in performance from the usage of the hyperthreads can be analzyed more in detail by running the algorithm in just one core, isolating it to run with 4 threads. The results are depicted in Figure 6.8, showing a slight improvement when using two hyperthreads.

For SMT resources, pairs of hyperthreads compete in the same core. As there are 4 hyperthreads per core, an optimal situation would be a four-step pipeline instead of a two-step one. In the latter, pairs of hyperthreads still perform exactly the same computations, either the *predict* or *update* ones, which do not allow to properly share SMT resources inside the core. Using Intel VTune, a performance analysis is run to analyze the threads' behaviour; the results are depicted in

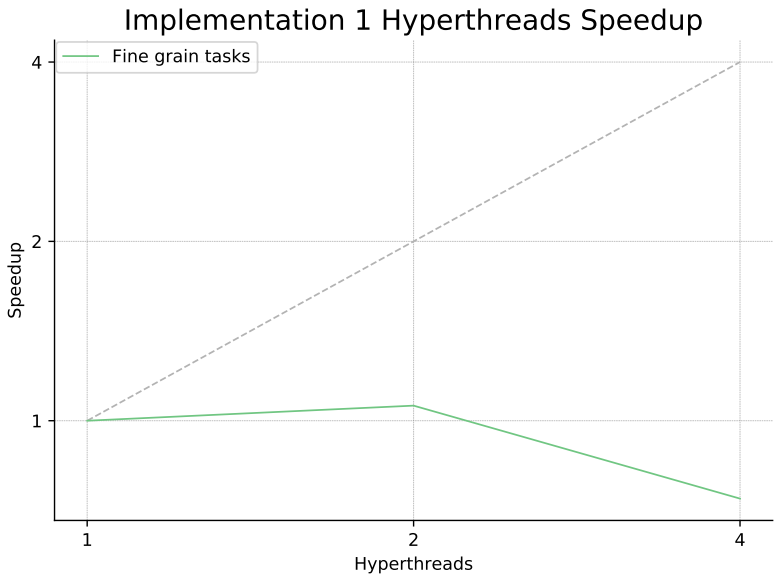


Figure 6.8: Predict - Update parallelization speedup in one core

Figure 6.9. Threads are idle most of the computing time, which indicates that the main bottleneck is the overhead of the parallelization. Because LHCb uses a small state vector and covariance matrix, the actual computation for the Kalman filter in the predict and update stages can be computed too fast to compensate the overhead of spawning a thread. The parallelization overhead dominates the computation, creating a bottleneck.

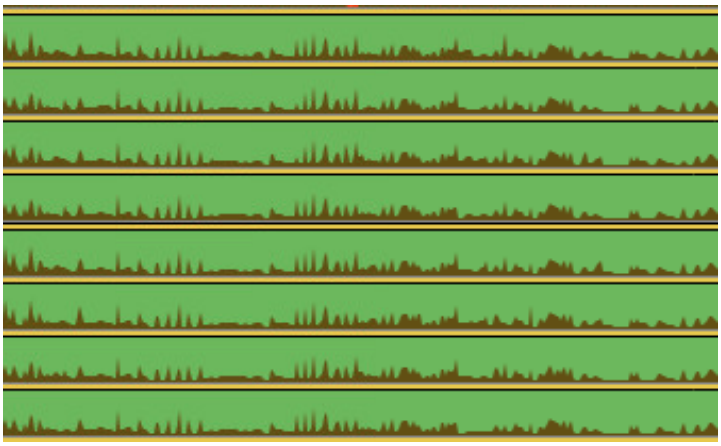


Figure 6.9: Predict - Update parallelization overhead. X axis indicates time; Y axis indicates different threads.

6.1.2 *Four-stage track sections pipeline*

To reduce the parallelization overhead, a different scheme with a coarser parallelization grain is implemented. As stated before, adding

more stages to the pipeline would be beneficial to help reduce the competition for SMT resources between the hyperthreads inside a core. As the parallelization overhead dominates the computation, increasing the grain size for the threads would reduce the bottleneck. To overcome these limitations a different parallelization scheme is followed, implementing a pipeline of four stages with the *input*, *pre*, *main* and *post* stages. This pipeline is depicted in Figure 6.10, where the four stages of the pipeline are filled using four threads after three stages.

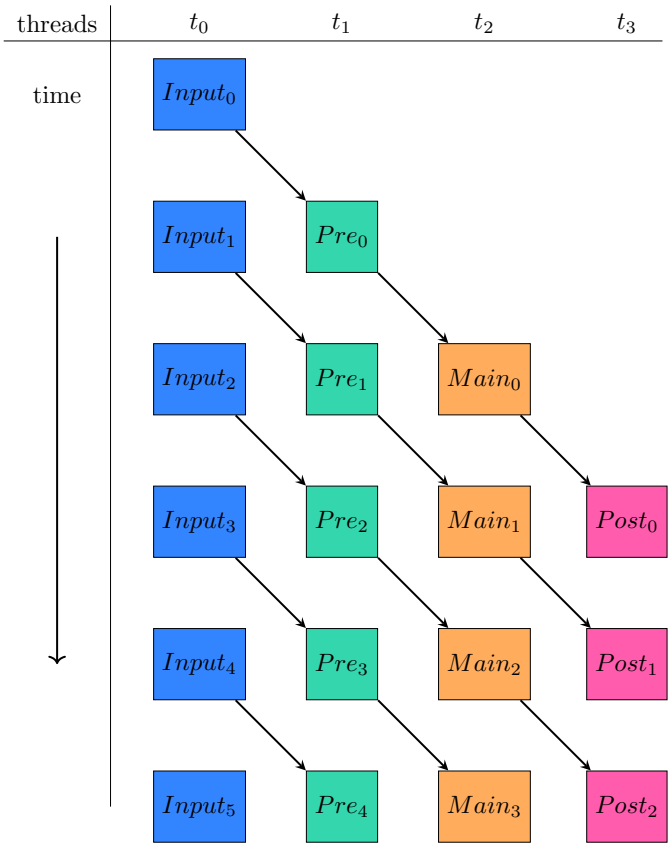


Figure 6.10: Four-stage pipeline

The implementation of this pipeline matches the number of hyperthreads in each core with the number of pipeline stages, allowing for different computation workloads across the hyperthreads. The increased workload where each stage computes various hit nodes allows for a reduced parallelization overhead.

The scalability of the four-stage pipeline implementation is shown in Figure 6.11. The throughput performance of this implementation improved compared to the previous two-stage pipeline implemented with the *predict* and *update* stages. Compared to the original implementation the scalability remains worse, where both cores and the hyperthreads present a lower throughput. The workload of the dif-

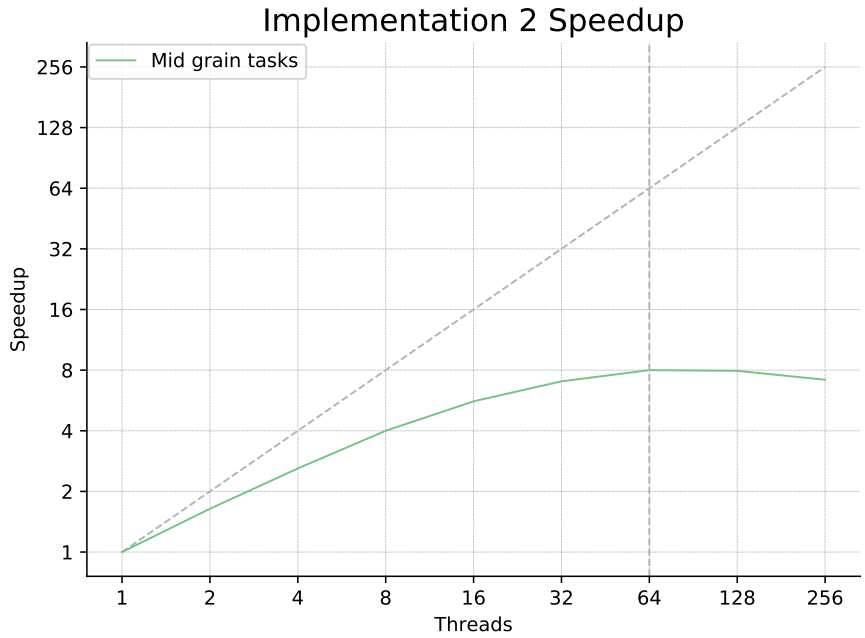


Figure 6.11: Four-stage parallelization speedup with input, pre, main and post stages.

ferent stages in this implementation is very similar as for the SMT resources: the equations to apply in the different stages of the pipeline are nearly identical, causing the different threads to compete for the same SMT resources. As the *pre*, *main* and *post* stages are computing various nodes each with their own *predict* and *update* stages, these remain very similar in workload. Another bottleneck for the pipeline is the unbalanced stages of it. As depicted in Figure 6.2, the *main* stage concentrates a majority of the hits in the track, making it the slowest stage and reducing the performance of the other.

To isolate the scalability of the hyperthreads a benchmark in one core is performed. The results are shown in Figure 6.12. Hyperthreads throughput outperform the previous two-stage implementation, with the best improvement found when using all the hyperthreads. The scalability is shown to be worse with both the cores and the hyperthreads, exposing the parallelization communication overhead of this implementation. The bigger grain size makes the overhead smaller compared to the two-stage pipeline, but the communication overhead dominates the computation.

The parallelization overhead is depicted in Figure 6.13 which shows how the overhead still dominates in spite of the bigger grain size. A group of threads is shown from Intel VTune, where the dark brown area shows the useful work being done by a thread. It shows how threads are still idle in all threads, with varying amounts of CPU usage. Compared to the previous implementation, the CPU usage is higher

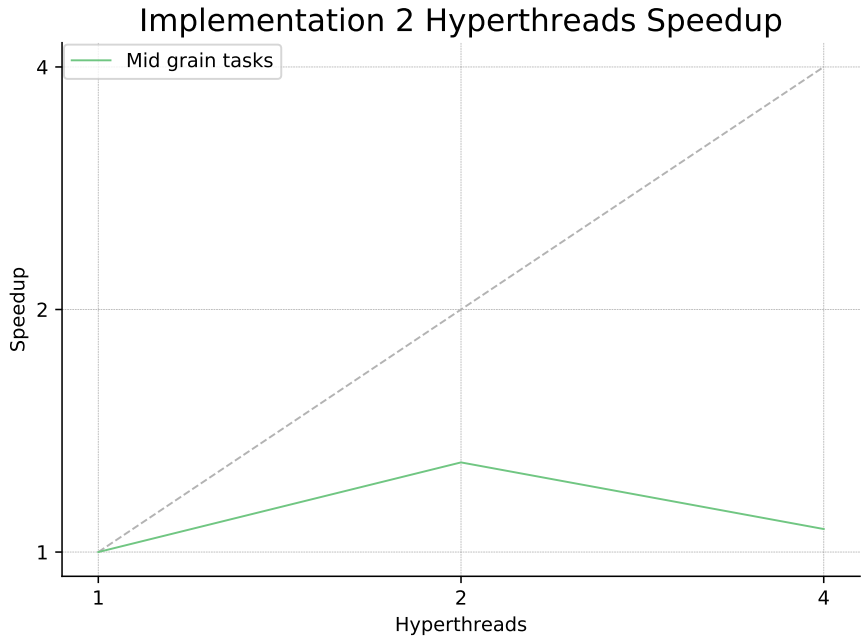


Figure 6.12: Four-stage parallelization speedup in one core with input, pre, main and post stages.

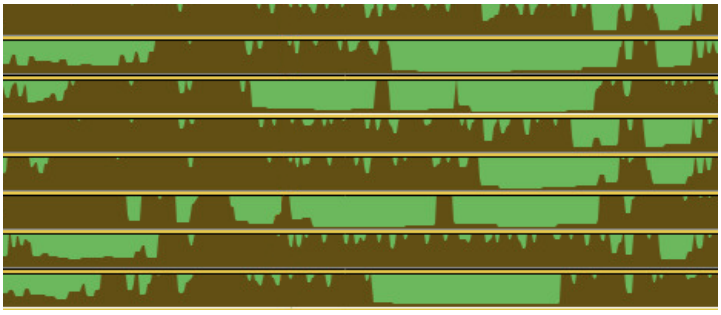


Figure 6.13: Four-stage parallelization overhead for various threads with input, pre, main and post stages. X axis indicates time; Y axis indicates different threads.

for all threads, which confirms the reduced parallelization overhead for the same algorithm with a different pipeline implementation.

6.1.3 *Forward-backwards-smoother pipeline*

The final, highest level of parallelization in the presented Kalman filter increases the grain size to the biggest possible while parallelizing intra-track. By performing full track fits in the *Forward*, *Backwards* and *Smoother* stages, each pipeline stage is assigned to process the track from the first to the last hit in a given direction. An *Input* and *Final steps* stages are included in the pipeline creating a five-stage pipeline.

This pipeline is represented in Figure 6.14, where all stages of the pipeline are filled after four stages to process tracks in parallel.

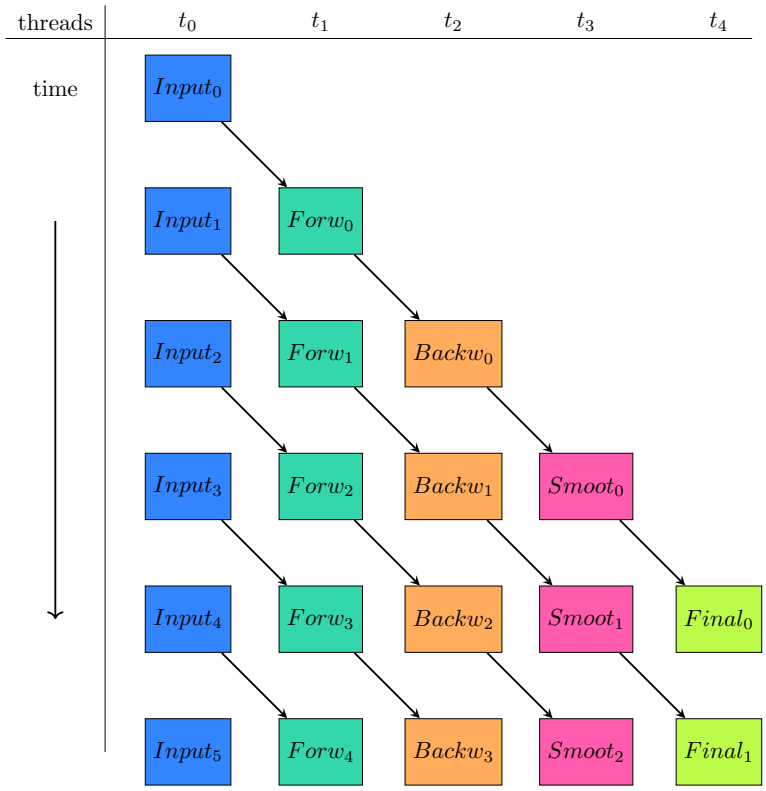


Figure 6.14: Forward-Backwards-Smoother pipeline

The main workload of the fitting stages includes the stages described previously without parallelizing them: the *pre*, *main* and *post* stages, each with their set of *predict* and *update* stages for each node, increasing the grain size of the threads and reducing the parallelization overhead.

The scalability of the five-stage pipeline is depicted in Figure 6.15. It shows an extra improvement in throughput compared to the previous implementations. It better scales due to the bigger grain size implementation, where each stage of the pipeline includes all the stages from the previous pipelines. The thread communication overhead is reduced which allows for the tasks to do more work in bigger chunks. The bigger size of the tasks, where a complete Kalman filter is applied along the track, better suits the parallelization framework.

The hyperthread usage of the five-stage pipeline is shown in Figure 6.16. It depicts the usage in a single core with its four hyperthreads, where the main improvement comes from the usage of the hyperthreads. The higher number of pipeline stages with different workloads allow the hyperthreads to better compete for resources for the different workloads. The thread communication overhead and pipeline imbalance, dominates the computation when using all available threads. The *Input* and *Final steps* stages hold the smallest

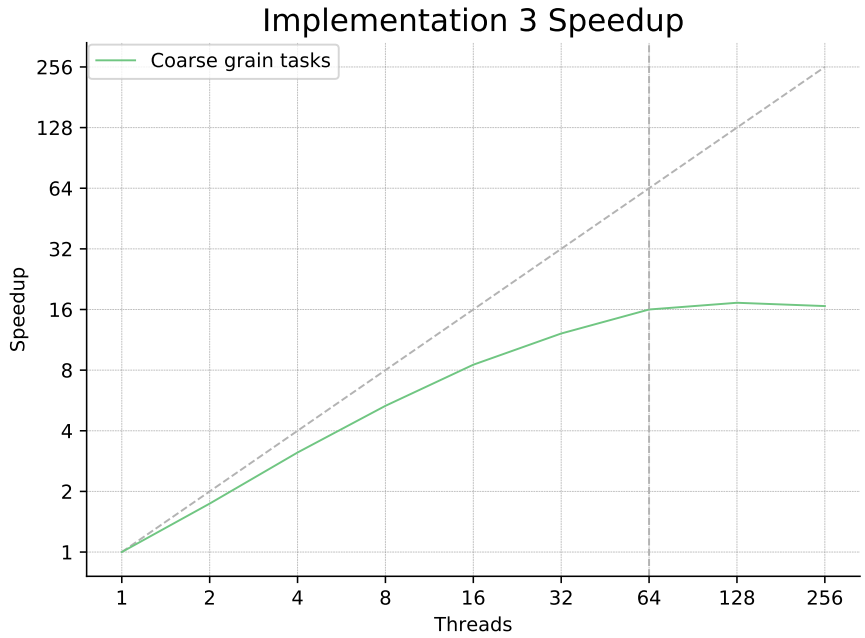


Figure 6.15: Predict - Update parallelization speedup with Input, Forward, Backwards, Smoother and Final steps stages.

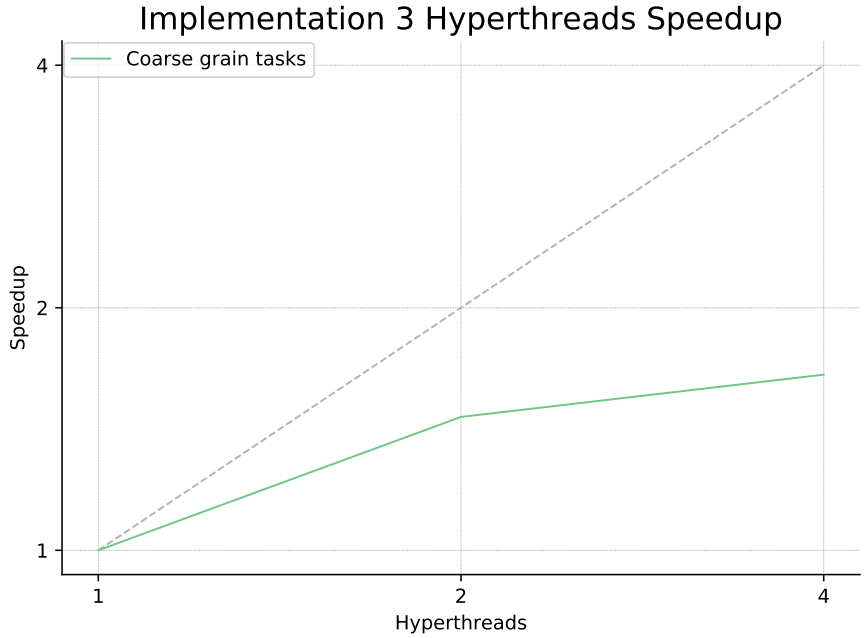


Figure 6.16: Predict - Update parallelization speedup in one core with Input, Forward, Backwards, Smoother and Final steps stages.

workload compared to the main ones, where the *forward*, *backward* and *smoother* stages remain more balanced between them. A more balanced pipeline without a stage creating a bottleneck for the others allows for better throughput processing.

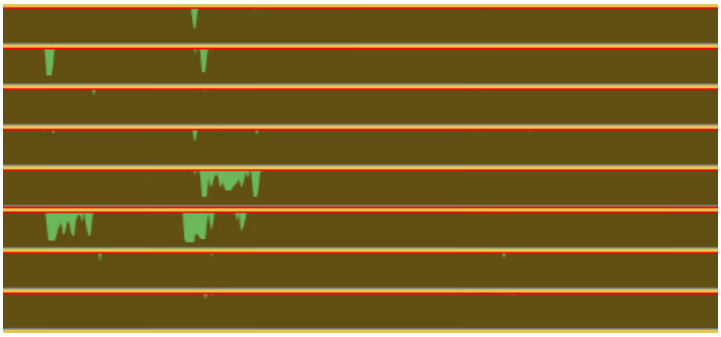


Figure 6.17: Predict - Update parallelization overhead in one core with Input, Forward, Backwards, Smoother and Final steps stages. X axis indicates time; Y axis indicates different threads.

An analysis of the parallel overhead is shown in Figure 6.17, where the parallel overhead is reduced compared to the previous implementations. The CPU usage of the different threads is close to being maximized, which indicates the better parallel utilization of the cores and hyperthreads with this parallelization scheme. This implementation achieves a percentage of speedup with every increase in number of threads used, except for when the processor is using more than 50% of the threads, this is, when going from 128 threads to 256 threads. As the resources of every core are being shared between 3 to 4 threads in this case, some functional units will not be available for every thread at a given time.

6.1.4 *Closing for the SMT multi-threaded Kalman filter*

The overhead cost of running various threads to parallelize within a single track remains too high to compensate the use of the hyperthreads. The size of tracks at LHCb experiment is small to be parallelized with the presented parallelization library, TBB. At the Kalman filter stage, where input tracks are already reconstructed by previous algorithms, there is no combinatorics problem to test multiple hit candidates as a good fit for the track. This makes the algorithm simpler in complexity, where the computing challenge remains in the big amount of tracks needed to be processed per second. Future changes and upgrades to the detector will increase the amount and detail of the collected data, which will naturally increase the grain size of the tasks of the parallelization schemes presented here. This will improve the performance of the different pipelines presented here as the parallelization overhead will be further reduced.

Having explored the parallelization opportunities that the LHCb Kalman filter presents, a research and development is performed by exploring how to better express the Kalman filter by using generic parallel patterns. These generic patterns are optimized to exploit the architecture of the Intel Xeon Phi, and selected patterns are used

to utilize a more realistic scenario of event processing that better represents the real experimental set-up of the experiment.

6.2 PATTERN BASED LHCB-KALMAN FOR THE INTEL KNL

Algorithm development and usage for simulation and data-taking in high-energy physics experiments go through long life-cycles. The expected long life cycle of these applications demands, not only a high degree of performance optimization, but also maintainability and portability. These goals are, of course, ubiquitous in scientific and engineering software and numerous solutions have been proposed. Among these, parallel programming patterns [102] offer a structured approach to parallel programming similar to software design patterns [66] by Gamma et al.

This section explores and demonstrates how the aforementioned goals can be achieved on the use case of LHCb experiment software framework. More specifically it explores the uses, advantages as well as limitations of generic parallel programming patterns in the context of high-energy physics analysis software. As baseline to test it the standalone *cross-kalman* application is used as a high-performance Kalman filter implementation. Furthermore, it focuses on the Intel Xeon-Phi Knight's Landing (KNL) platform as a pertinent example of modern HPC platform and possible choice for LHCb's computing infrastructure upgrade. This specialized processor has been successfully used in other scientific fields, as an example, GADGET, a toolbox for computational astrophysics problems [11] uses different data layouts and vectorization techniques targetting amongst others the KNL platform. Another example presents itself through the work of Madhavan et al. [95], who adapted molecular dynamics packages, such as NAMD, LAMMPS, GROMACS and CP2K, for the Phi KNL architecture and compared the benefits with respect to using a multi-core Intel Xeon Broadwell processor.

A number of studies exist, which investigate different core and memory affinity strategies for improving application performance on multi-/many-core processors.

Dependent and independent thread pinning strategies show improvements in performance in [100], and in more recent work applying dynamic thread affinity between parallel regions at run-time in [101]. Isolating workloads using CPU affinity techniques for performance and energy efficiency were studied in [117].

Furthermore, a large body of research is found that uses parallel patterns to enhance the performance and maintainability of scientific and industrial applications. For example, DMLL, an intermediate language that uses parallel patterns was developed for efficient resource usage on different platforms in [24], whereas code annotations were also used on stream processing high frequency trading applications

in [46]. FastFlow, a parallel programming framework has been used to adapt scientific bioinformatic applications including CPU affinity techniques in [104].

The adoption of shared-memory parallel frameworks in high-energy physics is relatively recent. Numerous results improving the performance of DAQs using parallel frameworks have been presented. GooFit [8] is a highly efficient numerical library, which can make use of OpenMP and CUDA back ends, offering evaluation functions for normalization integrals used in particle physics applications. At the same time, CERN experiments largely rely on TBB or custom C++ implementations to express parallelism [3, 13, 76].

6.2.1 *Generic parallel patterns*

A *parallel pattern* has been described as a recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design [102]. Patterns are applied in different areas of computing such as programming, architecture or compilers. Parallel patterns in particular are useful to describe, design and implement systems that expose parallelism at some level, and these are independent from the programming language or underlying hardware that is used: these are generic parallel patterns [99]. Parallel patterns can be classified into two main categories: data and streaming patterns, which mainly differ in the information about the size of the collection to operate with [121], as depicted in Figure 6.18

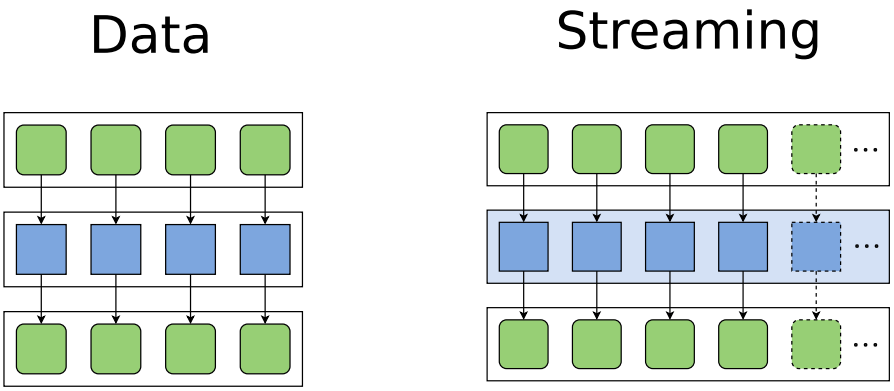


Figure 6.18: Data and streaming patterns represented.

- *Data parallel patterns*: operate over a set of data which size is known before applying the computation. The dependencies between the items to operate over in parallel is also known. Examples of these patterns are *Map*, *Reduce* or *Stencil*.
- *Streaming parallel patterns*: differ from the data parallel patterns in that the size of the input is not known in advance, and it is expected to have a channel where more items to parallelize

arrive constantly. This scenario adds a level of complexity which encourages different parallel patterns to be used to make the computation more efficient. Streaming patterns may have one or more scheduling elements to distribute the items to be parallelized as these arrive. Example of these patterns are *Pipeline*, *Farm* or *Filter*.

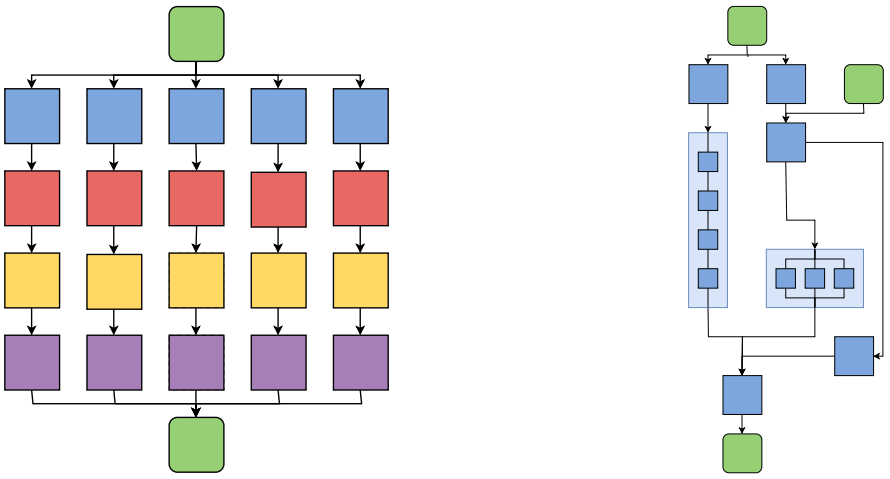


Figure 6.19: Parallel patterns composition and nesting.

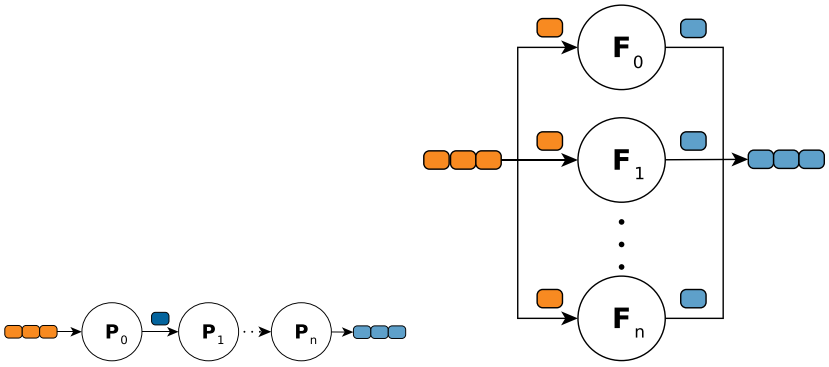
Generic parallel patterns can be composed and nested, where each defined element could be intra-parallelized with a parallel pattern, as depicted in Figure 6.19: the figure in the left represents several pipelines running in parallel with a shared input; the figure in the right represents a graph of tasks where some elements are replaced by patterns such as a pipeline or a map. By reasoning about parallelization in terms of these patterns, parallel programming can be described in a richer manner; taking into account parallel hardware by design rids the programmer from going into the details about synchronization and scheduling that parallelization demands. This allows to introduce changes to the design of a system described in terms of parallel patterns in a way that scales in both directions.

6.2.2 *Parallel patterns for the Kalman filter*

These parallel patterns allow to simplify the development and design of parallel applications. Furthermore it simplifies the process of testing the impact of different patterns, and offers a catalog of well-known patterns that better fit in different architectures.

GrPPI is a generic and reusable parallel pattern interface for C++ applications [120]. Specifically, GrPPI takes full advantage of modern C++ features, metaprogramming concepts and generic programming to act as a common interface for the execution environments OpenMP, C++ threads, Intel TBB and CUDA Thrust. Its design allows users to

leverage the aforementioned execution frameworks, hiding away the complexity behind the use of concurrency mechanisms. Furthermore, the modularity of GrPPI permits to easily integrate new patterns, while combining them to create more complex constructions. Thanks to these properties, this interface can be used to implement a wide range of stream-processing and data-intensive applications with relatively little effort. Resulting codes are portable and can be run on any platform supported by the underlying parallelization frameworks.



(a) Pipeline (b) Farm

Figure 6.20: *pipeline* and *farm* pattern diagrams.

In the context of cross-kalman filter, the *pipeline* and *farm* data stream processing patterns are selected as the best candidates to implement the pattern-based version of the Kalman filter. Figure 6.20 depicts the diagrams for the aforementioned patterns: the *pipeline* pattern receives incoming items in the input, each stage of the pipeline processes an item and passes it to the next stage until the final one outputs the result items. The *farm* pattern applies a function to each item appearing in the input, where each item can be processed independently from the others.

Different phases that may run in parallel were identified in Section 6.1 for LHCb's Kalman filter. To apply the generic parallel patterns, a similar structure is used for the selected *pipeline* and *farm* patterns. A more general display of the different phases is shown in Figure 6.21, summarizing the data flow of the phases of the reconstruction algorithm with an initial *input* phase.

From finer to coarser grained tasks, the Kalman filter can be split at different levels for parallelism. At the lowest level or finer grained, the amount of work for each step is demonstrated to be too small to compensate for the parallelization overhead. At a higher level or coarser grained, the best-case scenario that was tested to parallelize tracks during the processing is to divide the *forward* and *backwards* with the extra *input* and *latest iterations* steps. These can be computed in parallel using a pipeline pattern as demonstrated. This can be arranged like that due to the particle trajectories of each event being

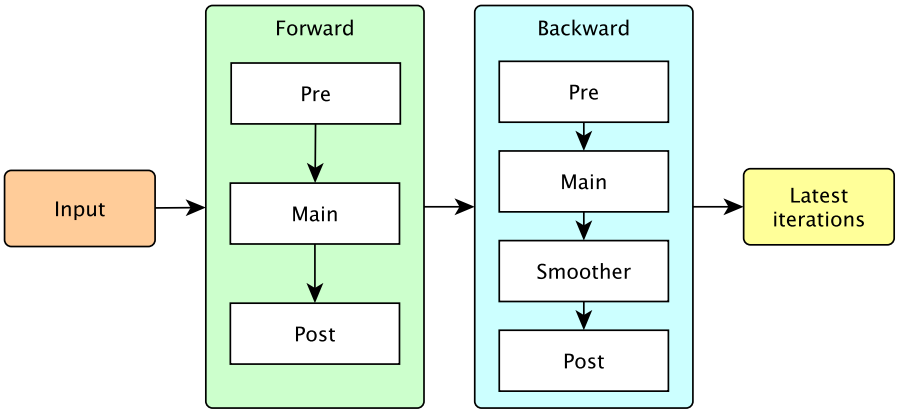


Figure 6.21: Phases of the Cross-Kalman algorithm. Each phase can be decomposed in logical steps to get more fine-grained parallelism. The arrows show the order in which the phases and steps must processed.

independent from each other. At the highest level, each particle trajectory can be computed sequentially, where the parallelism is given by processing multiple particle simultaneously. This kind of parallelism can be exploited by using a *farm* pattern.

6.2.2.1 *Pattern-based implementation*

The baseline parallel version of the Cross-Kalman algorithm is based on a TBB `parallel for` in charge of individually processing trajectories, which in turn leverages SIMD processor capabilities as nested parallelism. However, a data-parallel approach is not well suited for the LHCb framework, since the data should be processed as soon as it arrives from the detector in near real-time. Indeed, the data stream processing paradigm seems to be a better fit for the LHCb framework.

Also, given the expected long life cycle of this framework, it is worth using high-level parallel abstractions in order to make the application maintainable and portable. Using high-level abstractions when programming has shown to improve the programming performance of an application [85][54]. To accomplish these goals, the baseline application is expressed in terms of GrPPI parallel patterns, allowing parallelism at both phase and trajectory levels.

Figure 6.22 represents a CK-*pipeline* where different stages are visually related to their respective pattern, the different functions can run in different threads in parallel. The parallelization at stage level is achieved by means of the *pipeline* pattern composed with the *farm* construction, as shown in Listing 6.1. This composed pattern creates a pipeline of four stages, where the three last stages are nested *farm* parallel patterns. As can be seen, the *forward*, *backward* and *output*

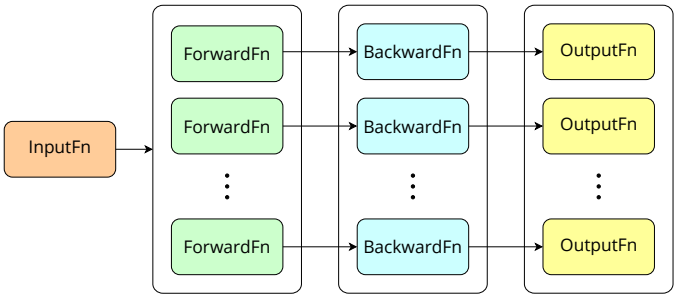


Figure 6.22: CK-*pipeline*

```
// Pipeline pattern implementation
grppi::pipeline(parallel_execution_native{},
  input_func,
  grppi::farm(n_thr1, forward_func),
  grppi::farm(n_thr2, backward_func),
  grppi::farm(n_thr3, output_func)
);
```

Code 6.1: Parallelism at phase level.

stages, representing *pipeline* stages, are computed in parallel by the corresponding *farm* patterns.

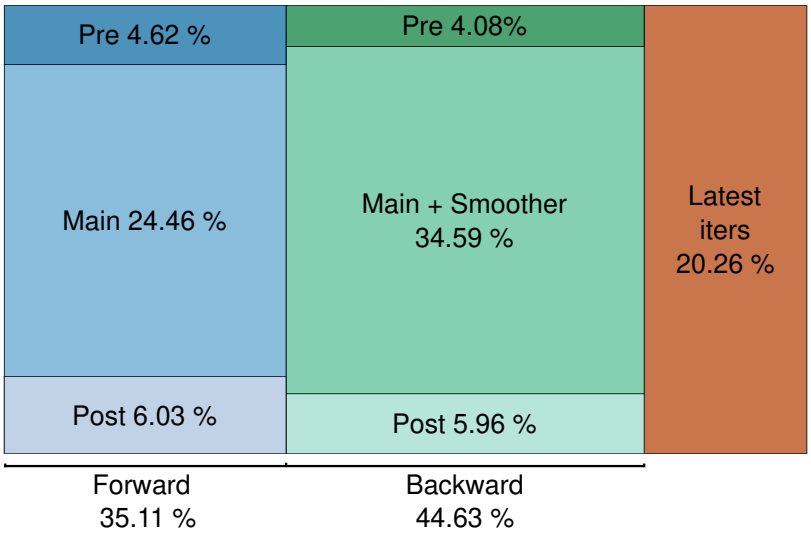


Figure 6.23: Percentage of execution time per algorithm stage. The overall time is roughly split into three parts, *forward*, *backward* and *latest iterations*. The former two are dominated by the main iterations.

Since the overall *pipeline* performance is bound by the throughput of its slowest stage, it is essential to ensure that all *pipeline* stages exhibit a balanced throughput. To improve the overall throughput, worker threads are distributed proportionally among the nested *farm* stages according to their execution times. Figure 6.23 shows the execution time of the different *pipeline* stages for the CK-*pipeline* version. The

```
// TBB Pipeline
auto inFn = [&] (tbb::flow_control& fc) -> PC* {...};
...
auto inFilter = tbb::make_filter< void, PC* >
    (tbb::filter::serial_out_of_order, inFn);
auto fwdFilter = tbb::make_filter<PC*, PC*>
    (tbb::filter::parallel, fwdFn);
auto bwdFilter = tbb::make_filter<PD*, PC*>
    (tbb::filter::parallel, bwdFn);
auto literFilter = tbb::make_filter<PD*, void>
    (tbb::filter::parallel, literFn);
...
tbb::parallel_pipeline (numTokens, inFilter & fwdFilter & bwdFilter & literFilter);
```

Code 6.2: TBB pipeline implementation

second and third *pipeline* stages consume roughly 35 % and 45 % of the time, while the latest iterations only consume about 20 %.

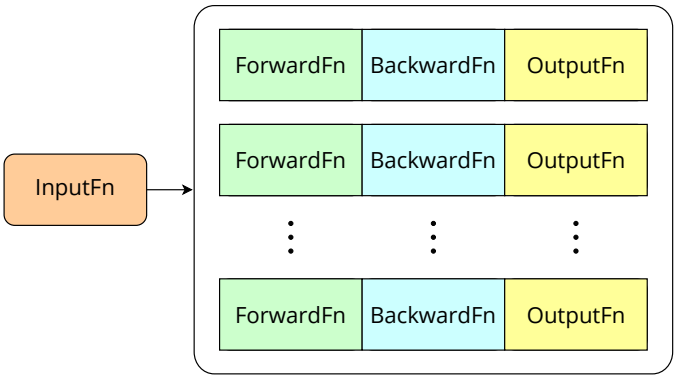


Figure 6.24: CK-*farm*

Figure 6.24 visually represents the CK-*farm*, where it can be seen how forward, backward and output functions run sequentially. The parallelization at trajectory level makes use of the *farm* pattern. In this configuration, all phases of the Cross-Kalman algorithm are collapsed in a single work task, processed by the different worker threads (see Listing 6.4). At construction the pattern is assigned two functions. The first implementing the stream producer, while the second implements the Kalman filter. Using this pattern, the input function is executed by a single thread that receives the input data. The data is then forwarded and distributed among the different worker threads in the *farm* for the Kalman filtering.

In general, using GrPPI for expressing parallelism at pattern level abstracts users from the specifics of the parallel frameworks used underneath. Furthermore, thanks to the high-level pattern interfaces the application source code is more readable, structured and maintainable than in the original version.

```
hwloc_topology_t topo;
hwloc_topology_init(&topo);
hwloc_topology_load(topo);
// CPU affinity
hwloc_bitmap_t cpu_set = hwloc_bitmap_alloc();
hwloc_bitmap_set(cpu_set, core_id);
hwloc_set_cpubind(topo, cpu_set, HWLOC_CPUBIND_THREAD);
// NUMA affinity
hwloc_bitmap_t numa_set = hwloc_bitmap_alloc();
hwloc_bitmap_set(numa_set, numa_node_id);
hwloc_set_membind_nodeset(topo, numa_set, HWLOC_MEMBIND_BIND, HWLOC_MEMBIND_THREAD);
```

Code 6.3: Example of `hwloc` for CPU and NUMA affinity.

```
// Farm pattern implementation
grppi::farm(p, input_func, process_func);
```

Code 6.4: Parallelism at trajectory level.

The current GrPPI framework is not yet able to control data locality nor thread-core affinity. Affinity techniques *i)* prevent thread migration; and *ii)* allow the application to efficiently exploit the memory bandwidth provided by ccNUMA architectures. Therefore, the ability of controlling affinity becomes extremely important especially on architectures such as the Intel Xeon Phi KNL. In the next section, the extensions made in GrPPI for supporting thread and NUMA affinity during the pattern execution is described in detail.

6.2.2.2 *Support for CPU and NUMA affinity*

To support thread and NUMA affinity, the Portable Hardware Locality library (`hwloc`) [23] is used, a software package providing portable hierarchical topology abstractions of the underlying architecture in terms of NUMA nodes, CPU sockets, shared caches, private caches, SMT, etc. [70]. Thanks to this software, it is possible to set the CPU and memory affinity to each of the threads involved in a parallel computation. Listing 6.3 shows an example where the current thread is pinned to a given core and NUMA node by means of the `hwloc` API.

To provide support for both CPU and NUMA affinity in GrPPI, the execution models are extended with two new functions that allow defining a set of specific cores and NUMA nodes that a given thread can use. Listing 6.5 shows the new GrPPI interfaces defined for establishing the CPU (`set_thread_affinity`) and NUMA affinity (`set_numa_affinity`). As it can be seen, both functions receive two arguments: the first refers to the thread ID within the execution model selected, while the second is a C++ `std::vector<T>` that determines the cores or NUMA nodes where the thread can run or allocate mem-

```
parallel_execution_native p{};
p.set_thread_affinity(tid, cores);
p.set_numa_affinity(tid, numa_nodes);
```

Code 6.5: GrPPI interfaces for CPU and NUMA affinity.

ory. Note that these functions call themselves the `hwloc` routines in Listing 6.3.

While providing support in TBB for NUMA architectures is possible using more advanced features, such as `tbb::task_arena` alongside TBB task affinity, the GrPPI solution is believed to be much simpler.

Focusing on the KNL architecture and the nature of the Cross-Kalman algorithm, it is noted that the application can greatly benefit from thread and data locality techniques if they are properly tuned. Indeed, memory access latencies can be reduced if data is located on a NUMA node belonging to the core where the thread runs. To support these affinity features, the presented *pipeline* and *farm* versions of the Cross-Kalman algorithm are modified to efficiently use the KNL core and memory architecture with the SNC-4 clustering mode flat memory mode. This cluster mode of the KNL makes it possible to the spawned patterns, both CK-*pipeline* and CK-*farm*, to efficiently access the memory that is closest to each cluster. Four instances are spawned, one on each quadrant and with the corresponding number of threads of the quadrant. Each instance allocates a `parallel_execution_native` execution policy and sets different thread and NUMA affinities depending on the target quadrant where the pattern will be executed.

6.2.2.3 *Experimental evaluation*

In order to evaluate the behaviour of the implementation, computer experiments are performed comparing the baseline TBB implementation with the GrPPI CK-*pipeline* and CK-*farm* patterned versions of the Cross-Kalman algorithm. For the benchmarks the following hardware and software setup was used:

- *Hardware*: Intel Xeon Phi 7210 Knight Landing with 64 cores and 256 hardware threads, 214 GB of DRAM and 16 GB of MCDRAM. A comparison of the different cluster and memory configurations available is presented for the KNL, and it is determined that the setup with the best performance for our use-case is SNC-4 and *Flat* memory model. SNC-4 allows us to explicitly pin threads to NUMA nodes and benefit from memory locality. The *Flat* memory model enables the programmer to explicitly control data placement, while *Cache* mode will move data to MCDRAM according to the caching policy of the KNL memory management unit.

- *Software*: All benchmarks were compiled with `gcc 6.2.0`, with `-O2 -march=native` optimization flags, which was determined to be the most efficient configuration. GrPPI uses for its execution environment C++11 threads. The baseline uses Intel TBB for multithreading and Agner Fog's Vectorclass [63] for vectorization. For instrumentation and profiling Intel VTune Amplifier 2017 was used.
- *Benchmark*: All benchmarks were run with Monte Carlo simulated events generated using the LHCb simulation framework. Each test ran 400 000 events, distributed equally across all NUMA nodes. Each NUMA node hosts one instance of the parallel pattern with at least two threads. For the runs on 1, 2 or 4 cores the parallel patterns were executed in sequential mode allocating thus only one thread per pattern. For all tests, a thread local to the NUMA node loaded the events.

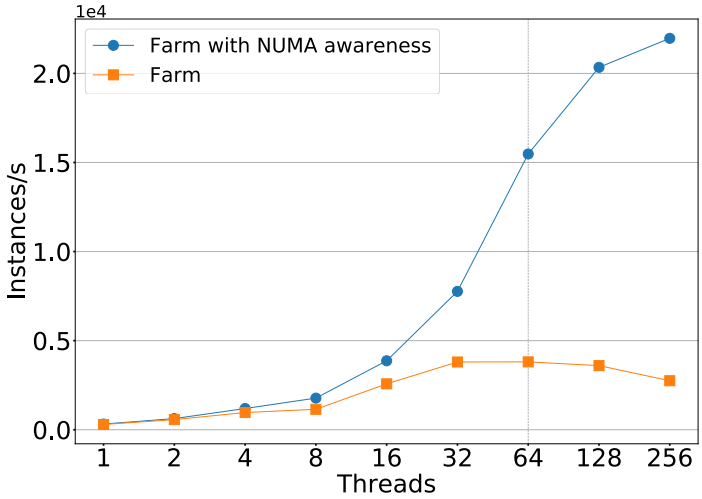


Figure 6.25: GrPPI with and without NUMA awareness for the CK-*farm* pattern. Introducing NUMA awareness has a strong impact on throughput on the Intel Knights Landing platform.

Each event contains one or more instances or groups of tracks. The performance of the different implementations is measured as the throughput of processed events per second.

Figure 6.25 shows the difference in throughput between the Kalman filter benchmark parallelized using the production version of GrPPI and the GrPPI+NUMA library. The performance of the two versions is comparable up to 16 threads. For higher thread counts, however, the performance of the production, non NUMA-aware version of GrPPI quickly degrades. The overall performance of the application even decreases beyond 32 threads. As for the GrPPI+NUMA, note that the optimizations presented allows the application to deliver good

performance for the full chip. Note that throughput degrades beyond 128 threads as the KNL cores become oversubscribed (4 hyperthreads per core).

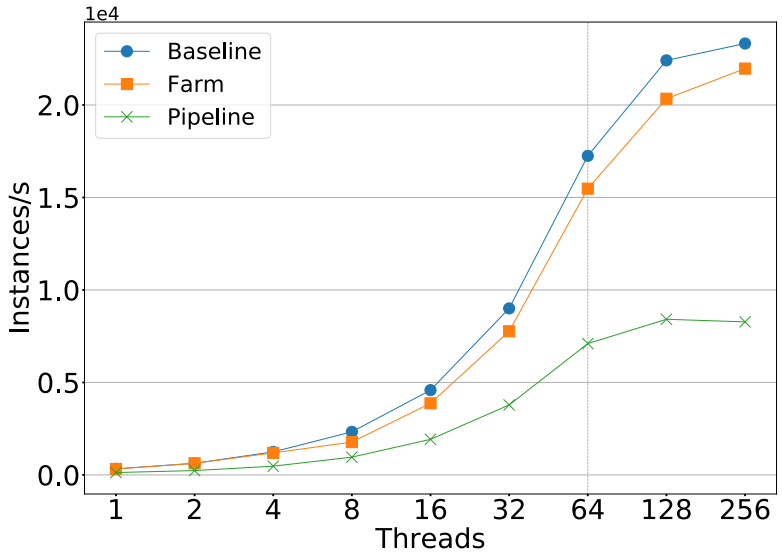


Figure 6.26: Throughput comparison for baseline, CK-*farm* and CK-*pipeline*

Figure 6.28 shows a comparison of the impact of different parallel patterns, solving the same problem, on the cache hit rate. While all three patterns exhibit similar hit rates (about 80 %) for the L1 cache, note a 7 % improvement when using CK-*pipeline* over the baseline implementation or CK-*farm* pattern. The improved hit rate is explained by the fact that on the KNL architecture, L2 caches are shared by the two cores of each tile (cf. Figure 5.10). CK-*pipeline* exerts less pressure on the shared cache as the two cores of the tile are both reusing the instances of the same event improving both temporal and spatial locality of memory access. In contrast, in the baseline and CK-*farm* implementations, the two cores of the same tile are always processing different events at the same time, resulting in a higher cache invalidation rate.

Unfortunately, demonstrated in Figures 6.26 and 6.27, the improvements in cache efficiency are completely dominated by the parallelization overhead, rendering the CK-*pipeline* variant significantly less efficient than the baseline and CK-*farm* implementations.

A comparison of the performance between the two main GrPPI pattern based versions with the baseline is presented. Figure 6.26 shows the throughput as a function of the number of threads. As it can be easily seen, the baseline implementation shows a 10 % higher throughput. This improved performance could only be achieved by hand tuning both the code and the execution environment. While the CK-*farm* implementation is slightly lower, it was both easier to

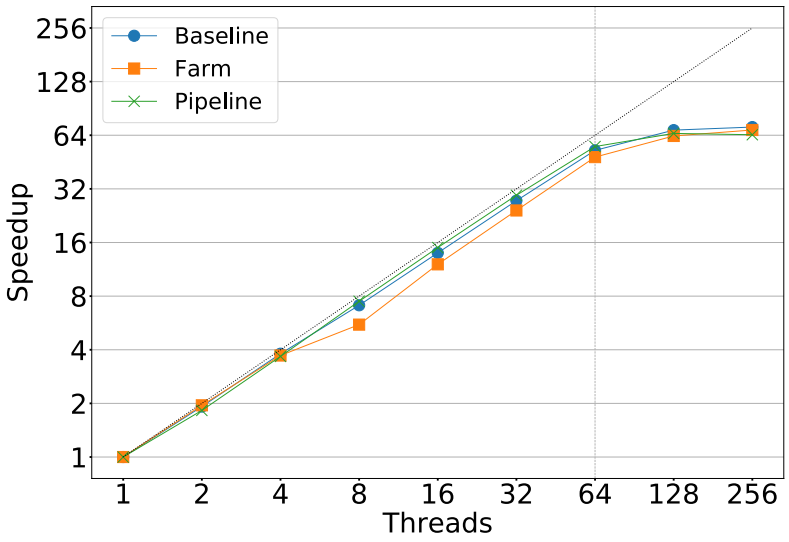


Figure 6.27: Scalability comparison for baseline, CK-*farm* and CK-*pipeline*

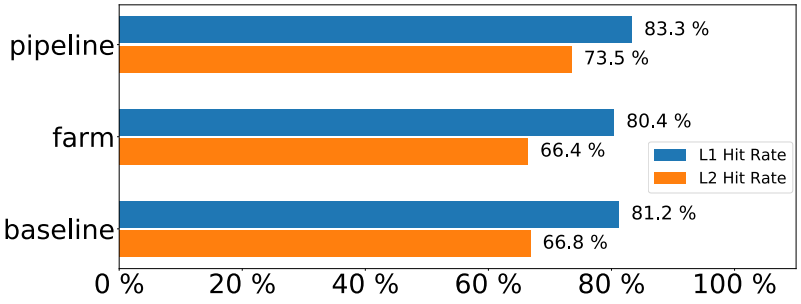


Figure 6.28: Comparison of L1 & L2 cache hit rates between the baseline, CK-*farm* and CK-*pipeline* pattern implementations. The *pipeline* pattern has a 7 % higher L2 cache hit rate than the other implementations.

implement and required no special set-up in the environment. As mentioned, the performance of CK-*pipeline* implementation is well below the other alternatives. Figure 6.27 is focused on the parallel speedup of the different implementations. Both throughput and scalability benchmarks were performed in a strong scaling scenario. Notice that despite absolute throughput varying across our different implementations, their scalability is similar, the parallel efficiency being 84 %, 76 %, 87 % for the baseline, CK-*farm* and CK-*pipeline* respectively. Beyond 64 threads the benchmarks are competing for the resources of the cores, which are shared among Hyperthreads, which leads to degradation in performance. The CK-*farm* implementation, although being vastly more efficient than CK-*pipeline*, exhibits a lower parallel efficiency starting at 8 threads. This behaviour can be explained by breaking down the total wall-clock times.

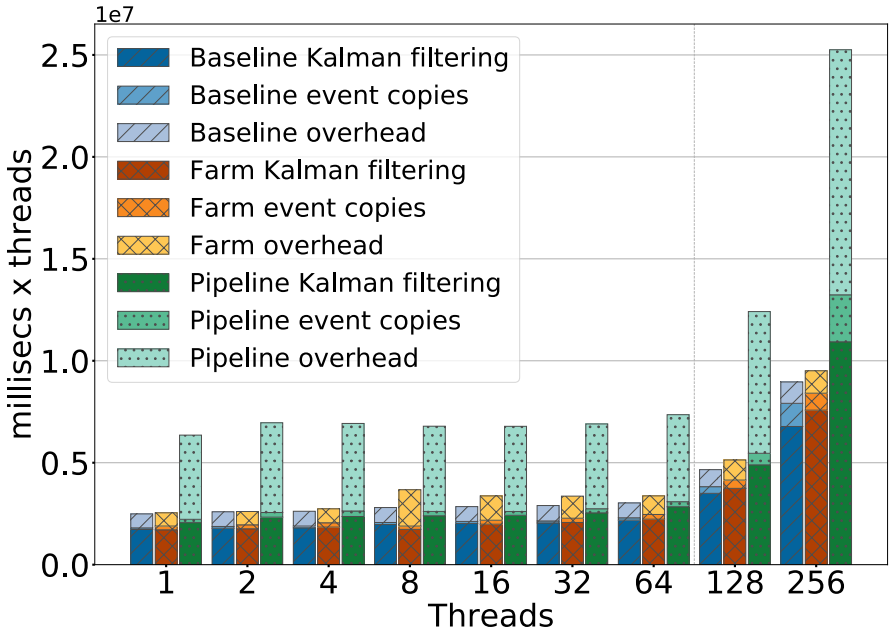


Figure 6.29: A decomposition of the wall-clock times into Kalman filter processing, data copies and framework overhead. The framework overhead in CK-*pipeline* is substantially higher than the baseline or CK-*farm* implementation.

Figure 6.29 depicts a comparison of the wall-clock time broken down into compute (Kalman filtering), data rearrangement (copies) and framework overhead between the 3 implementations. Note that, the times are multiplied by the number of threads to allow us to more easily compare the runs across the scale. As seen above, the baseline outperforms both our GrPPI implementations. Overall the TBB implementation exhibits a lower parallelization overhead, which is ascribed to TBB's work-stealing scheduler and KNL specific optimizations in the framework. The CK-*pipeline* implementation is dominated by the overhead in the parallelization framework, which is mostly due to

Table 6.1: Time spent in the producer and consumer tasks when using 8 threads with the GrPPI *farm*.

Task type	Task Time	Percentage on total
consumer_task	2620.818 s	98.23 %
producer_task	0.288 s	0.01 %

the pipeline steps being too fine-grained [82]. The aforementioned decrease in parallel efficiency of the CK-*farm* implementation can be seen here as a significant increase of overhead in the framework. Starting from 8 threads, the CK-*farm* employs a producer-consumer architecture to distribute the work packages among worker threads. The 8 threads are distributed as 2 threads per NUMA node, which means that effectively only the 4 consumer threads are working, while the other 4 producer threads are distributing the events. At least one producer and one consumer thread per NUMA node are needed. While the producer Table 6.1 shows the CPU time of the consumer and producer tasks. While the worker threads are executing the consumer tasks, the management threads are mostly idle. The idle time of the management threads is seen as parallel framework overhead. The remaining 2 % of CPU time is not reported here, which is spent elsewhere in the application.

6.3 SUMMARY

This chapter covers two main research efforts that includes both the LHCb Kalman filter and the Intel Xeon Phi KNL processor: the parallelization of the algorithm at an intra-track level focusing on improving SMT usage; and a generic parallel pattern-based implementation that focuses on the KNL processor and achieving a comparable performance with improved maintainability.

The intra-track parallelization shows the importance of the introduced overhead when running various tasks in parallel. For tasks that are too small, the parallelization overhead can dominate the overall computation, resulting in poor scaling and throughput improvements or event degraded performance. The LHCb tracks and associated data structures are shown to be too small to compensate the intra-track parallelization. This is demonstrated with different levels of fine grained tasks: the finer grained tasks show poor performance, but the coarser grained tasks present good scalability and performance. Scheduling a parallelization over the full tracks remains the highest throughput option until tracks hold a larger number of hits, or hits with more information.

The use of parallel patterns suitable for stream processing of data in the context of high-energy physics experiments was discussed in

this chapter. The LHCb proto-application Cross Kalman [27] is used to study generic parallel patterns through GrPPI and their impact on computational performance and programmer productivity. The GrPPI library was enhanced by adding NUMA awareness, including thread and memory binding using the `hwloc` API. GrPPI allows to easily use generic parallel patterns through its interface, where otherwise explicit parallel implementations are needed. The library supports multiple backends such as C++ threads or OpenMP, with almost no changes to its interface. Using the enhancements many-core hardware platforms are targeted, specifically the Intel Xeon Phi Knights Landing.

The implementation shows that through the use of abstractions for parallel processing, target applications are able to harness the full capabilities of modern hardware architectures with little effort from the programmer. The baseline implementation was benchmarked and compared against two streaming patterns achieving in the best case comparable performance without the need for hand tuning.

Other GrPPI backends could support NUMA awareness in the future, and a broader topology awareness through the `hwloc` API. This aims to provide a portable solution beyond the C++ threads and Intel Xeon Phi KNL demonstrated in this work. Other future improvements include extending patterns to being able to receive more than one execution model, thus yielding more flexible thread scheduling.

By modifying the proto-application to use streaming parallel patterns instead of data parallel patterns, the implementation models the real world scenario more closely and is able to better predict throughput bounds for the LHCb data acquisition cluster. The aforementioned use of abstractions also allows to more easily test other parallel patterns as well as different back-ends. A broader support for NUMA architectures can extend GrPPI's support for a more complete topology awareness.

7

HEP PARTICLE TRACKING WITH GPU ARCHITECTURES

The LHCb experiment is considering different hardware architectures for the coming upgrade. The compute power will have to increase to handle the continuous deluge of data from the detector. The big cost of the necessary increase in compute power, leads to the exploration of alternative hardware architectures. As heterogeneous data centers comprised with multi- and many-core CPUs and coprocessors/accelerators emerge, LHCb and other CERN experiments are currently considering different hardware options to reach the aforementioned performance goals for the coming years. The current LHCb computing farm consists of servers based on the x86-64 architecture. However, alternative architectures and accelerators are being tested in different trigger systems [25, 137, 143]. This is an indication that systems requiring high-throughput can be met in such alternative architectures.

LHCb computing farm needs to treat 30 million events per second, producing around 10^9 particles per second. Introducing an architectural change, poses multiple challenges in terms of software to perform particle tracking in real-time. Existing algorithms must be redesigned to fully exploit parallel architectures. Furthermore, the expected long life cycle of these algorithms demands not only a high degree of performance optimization but also maintainability and portability. Those goals are ubiquitous in the scientific and engineering software areas and different solutions have been proposed. Among these, GPU-based approaches have been a successful alternative in providing high-throughput in different scenarios [32, 111, 122]. This chapter presents the implementation of a data-oriented approach, focusing on creating algorithms for SIMD (Single Instruction Multiple Data) architectures, minimizing thread divergence, reducing data movements and memory footprint of the algorithm, which have been successful strategies to optimize algorithms for GPUs [79, 139]. It runs as part of the LHCb GPU sequence framework defined in [45], which allows multiple concurrent GPU stream execution.

This chapter is structured as follows: In Section 7.1 other scientific fields using GPUs are discussed, Section 7.2 introduces the GPU framework, and Section 7.3 explains how particle reconstruction is performed for the UT. The following three sections present the following:

a) Section 7.4 introduces a parallel version for the decoding of the raw input data, which ensures coalesced data write patterns and

produces a sorted SoA data structure, beneficial to the tracking algorithm.

b) Section 7.5 presents a fast tracking algorithm for high-energy physics detectors targeting SIMD architectures called *Compass*. The proposed algorithm can deal with deviated particle trajectories by a magnetic field.

c) Section 7.6 investigates the impact of the algorithm configuration on the physics quality of the results and analyze its computing performance on a variety of GPUs and CPUs.

Finally, Section 7.7 gives a summary of this chapter.

Some parts of this chapter have been published in the following journal/conference papers:

- Placido Fernandez Declara et al. «A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures.» In: *IEEE Access* 7 (2019), pp. 91612–91626. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927261

7.1 GPUS IN REAL-TIME, HIGH-THROUGHPUT SCIENTIFIC FIELDS

This work is focused on high-throughput computing workloads, which is the type of workload high-energy physics experiments like LHCb have. Other scientific fields also process large datasets and present similarities with LHCb, this is, they process numerous small units of work. A discussion of other real-time approaches and scientific applications which deliver high-throughput is discussed in this section.

GPUs have been used before in the field of high-energy physics with success. The ALICE experiment at CERN implemented track reconstruction in GPUs obtaining different speedups compared to the previously used hardware [123]. Note how the approach followed in this work is different than the one implemented in ALICE, as this work contributes to the implementation of the full *High Level Trigger* to run in GPUs, including the decoding and tracking of all subdetectors, thus avoiding much of the needed data transmission between main memory and GPU memory. Other HEP experiments have seen significant improvements when using GPUs to amend the performance of online selection [25, 131], or using a common code base to target both CPUs and GPUs using OpenCL, which shows the performance improvement of GPUs while supporting the x86-64 architecture [64].

The performance of DNA sequencing problems has been improved with GPUs in different high-throughput scenarios. The *Arioc* read aligner showed how using parallel algorithms with GPUs improved DNA sequencing throughput, achieving an order of magnitude faster

alignments [141, 142]. Pawar et al. benchmarked various DNA sequencing algorithms with different GPU-based tools against a CPU one; concluding that GPUs will replace CPUs in DNA sequencing for its higher-throughput processing [114]. Other DNA-related fields exhibit similar speedups: Samsi et al. [128] demonstrated how a single GPU is able to compare millions of DNA samples in seconds, Cadenelli et al. [26] compared offloading a genomics workload into FPGAs and GPUs from a CPU, resulting in the GPU outperforming both, although the GPU consuming more energy.

Other scientific fields benefit from high-throughput, real-time processing in GPUs. Radio telescopes need to filter data in their data acquisition systems; where software frameworks employing GPUs like *Bifrost* [44] have shown significant performance improvements. Other real-time radio telescope experiments studied the viability of using GPUs, where they encountered large computing speedups at a local level, but were limited by I/O when using multiple GPUs [96]. Others in the same field have successfully implemented GPU optimization schemes [80] achieving a 6× speedup compared to the CPU scenario, or used a GPU-based software framework and aggressive optimizations to be able to process data rates close to 1Tbit/s, like the *CHIME Pathfinder* radio telescope [119].

GPUs have also been studied in scenarios requiring real-time processing at fusion experiments [94] greatly reducing the wall-time compared to the CPU version. Real-time split-and-merge executions have been improved in multi-GPU scenarios by Han et al. [74], and X-ray computer tomography reconstruction in GPUs has shown how different optimizations can be implemented and combined to speedup GPU computations [17].

The approach presented here for using GPUs in high-energy physics delivers a parallel tracking algorithm which reconstruct particle trajectories that are bent under the influence of a magnet, describing a non-straight trajectory. It is focused on achieving high-throughput to meet the collision rate and real-time constraints of the LHC at CERN. Other scientific fields have been successful on implementing real-time high-throughput solutions with GPUs, where fields like DNA sequencing are already ditching CPU-based architectures to process their large datasets. Successful results in the HEP fields suggest that implementing a full filter with GPUs, including the decoding and tracking of charged particles, is a feasible task that will increase the filtering throughput capabilities of LHCb.

7.2 GPU FRAMEWORK FOR HLT1

Chapter 3 briefly introduced the Allen and Gaudi frameworks. Gaudi is the framework used for both HLT1 and HLT2, and it is used as a base for other applications at LHCb. It supports the execution

of source code in accelerators or coprocessors [9], allowing GPUs to interact with Gaudi to run specific applications or algorithms. To benefit from the GPU architecture it would need to run hundreds to thousands of events in parallel, as this is the design that Gaudi implements: each event is processed by one thread. This limitation would not allow to efficiently process events in GPUs where memory is scarce compared to the amounts of memory that can be used in a x86 workstation. Allen framework was created to overcome this limitation and focus its design around massively parallel accelerators and processors. It supports CUDA C++ to develop the algorithms, but the requirements for CUDA features are the minimum supported by the latest NVIDIA compiler. This allows the source code to be easily compiled for other architectures, such as x86. It does not renounce to advanced CUDA features to develop algorithms; the kernels that use this functionalities to provide better throughput are provided with a compatible version that supports simple CUDA C++ and older GPUs.

Allen framework is multithreaded, running various events in parallel. It is designed to run various CUDA *streams* in parallel to maximize the usage of the hardware. Algorithms run in a non-blocking asynchronous way, where the communication between the host and the GPU allows to send and receive data in a concurrent manner. Data transmission between host and GPU is effectively hidden as the amount of computation done in the device overcomes that of the data transmission. Allen is capable of running both CPU and GPU algorithms, which can be alternated by handling the necessary data transmission between them. Memory is managed by a custom manager which allocates memory for a whole *stream*. The custom memory management provided by Allen allows to overcome the memory limitations of GPUs and greatly reduced blocking in the GPU, as allocation and deallocation of memory in the device is a blocking operation. To overcome the memory limitations, dynamic memory allocations are not allowed in Allen: a size must be given in a predetermined way. Some kernels are dedicated to compute the sizes of some buffers to allocate memory for it by the custom memory manager. This design decision benefits the framework in terms of throughput performance.

7.3 UT PARTICLE RECONSTRUCTION

The UT reconstruction process is part of a chain of algorithms needed to run all the HLT1. The complete chain of algorithms is depicted in Figure 7.1 where the UT is highlighted. The UT subdetector system is described in Chapter 2. The UT subdetector receives as input reconstructed VELO tracks and UT raw banks. These raw banks need to be decoded before the tracking is performed to complete the whole UT reconstruction.

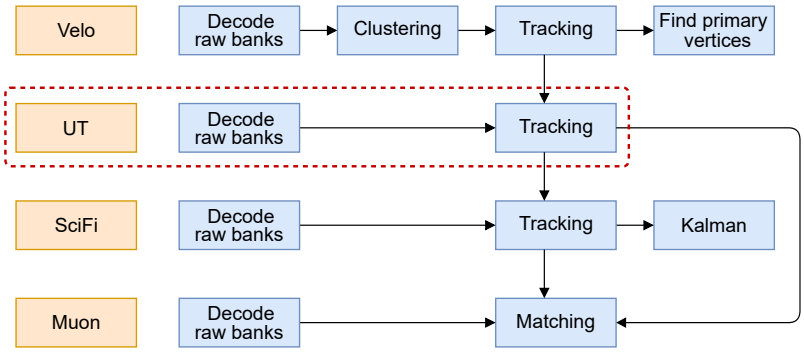


Figure 7.1: Complete High Level Trigger 1 sequence of algorithms at LHCb. The UT algorithms described in this chapter (dotted lines) are highlighted. UT is the second tracking sub-detector in the chain of algorithms, and it receives input from the UT raw banks and the VELO tracks. UT outputs reconstructed tracks for other sub-detectors.

Particles collide at the interaction point, and the resulting particles from the collisions are first reconstructed by the VELO sub-detector. A percentage of those particles travel out of the acceptance range of the UT, and the rest of them, in acceptance, leave activation signals with a high probability which are decoded in software to hit information. Using the VELO tracks and the UT hit information, combined with the geometry information and magnetic field influence from the magnet, UT tracking can be performed.

Tracking is performed by finding matching UT hits for every input VELO track, where a VELO track is a straight line. UT hits are considered to be compatible with a VELO track, resulting in a curved track bent proportionally to the track momentum. As the UT sub-detector is under the influence of the magnetic field, multiple possible matching hits can be matched for different slightly bent tracks [33]. This situation is represented in Figure 7.2, where a real situation is better represented with hundreds of tracks, and makes the problem of finding matching hits an exponential combinatorics problem [22].

The p-Kick method [20] is used to estimate the momentum of the track. Using it allows to perform a χ^2 fit providing the momentum of the particle. This method is used instead of a Kalman filter, used in other tracking algorithms, as it yields a better computing performance [21]. As this algorithm is focused on delivering high throughput in real-time, a faster method is favoured in comparison with the Kalman filter used in the HLT2 discussed in the previous chapter. To take into account the magnetic field during the algorithm, look-up tables are used, which give quick access to the influence of the magnetic field in different parts of the particle trajectory. Using the look-up tables, the deflection a track is expected to experience can be determined.

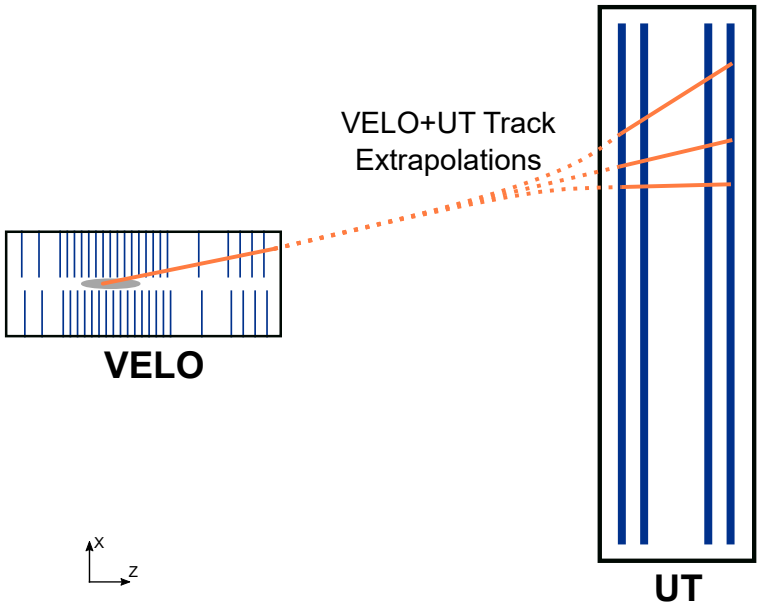


Figure 7.2: VELO track extrapolation to UT hits. A VELO track can be associated to various UT hits, where the UT track extrapolation does not necessarily follow a straight line. This leads to high combinatorics between the hits in the four panels, holding the main complexity of the algorithm.

A UT tracking algorithm is expected to achieve a high reconstruction efficiency with a low fake and clone rates for various types of tracks. The computing performance of the algorithm is determined by how many events per second can be processed for a given hardware configuration. This is a key aspect of event filtering in high-energy physics, especially for the LHCb experiment which will only rely on software for its event filter system. The combination of hardware and optimised software for it will need to process the 30 MHz rate of events in real-time.

7.4 UT DECODING

Before being able to execute the tracking algorithm, the raw input from the subdetector needs to be decoded into hit information. The decoding step needs to perform efficiently to run in real-time. This is parallelised by processing different chunks of raw input using GPUs, as it is a fundamental previous step for the tracking algorithm.

UT detector data is encoded into raw banks, in a highly compact format, containing the information required to obtain the UT hits. These raw banks are decoded into the parameters that define a UT hit. These are reduced to the minimum necessary to run the UT tracking algorithm, lowering the memory footprint of the algorithm. The decoded parameters are the following:

- *LHCbID*: a unique 32 bit identifier for the hit, which indicates the spatial position of the detection element.
- Z *at* Y = 0: the Z coordinate of the hit at the Y = 0 position, which is the centre of the panel in the Y axis. The Z coordinate indicates the panel for a specific hit.
- X *at* Y = 0: similarly to the previous parameter, this is the X position at the centre of the panel in the Y axis. This coordinate is given by the activated strip in a sector and it is different for the U and V layers.
- *yBegin* and *yEnd*: as the UT subdetector is a strip detector where the strips are arranged vertically, the specific Y coordinate of a hit cannot be obtained. Instead, a range on the Y axis delimits where the hit is located.
- *weight*: the uncertainty of the hit position.

The decoded parameters are stored in a structure of arrays (SoA). A SoA layout is used for storing the hits in a coalesced manner to maximise the memory bandwidth usage. To access the hits efficiently, a separated array is used to store the offsets between the hits. Using the offsets, one is able to determine which panel wants to refer to when accessing the hits, and so every GPU thread can access its specific hit. Events are computed by processing them in parallel, assigning single events to single blocks to distribute them in the GPU. An event results in various tracks, where different nested parallelisation schemes are applied for different kernels, which are described here.

Table 7.1: Kernel configuration for UT decoding. $events\_in\_execution$ are the number of selected events to process, where $array\_size$ is defined as the $events\_in\_execution \times 84$. 84 is the number of pre-defined sectors, where the number 4 used in various kernels is the number of panels. Threads with two arguments is the kernel execution configuration for thread blocks and threads in a block.

kernel	blocks	threads
calculate number of hits	$events\_in\_execution$	(64,4)
prefix sum reduce	$(array\_size + 511)/512$	256
prefix sum single block	1	1024
prefix sum single scan	$((array\_size + 511)/512) - 1$	512
pre-decode	$events\_in\_execution$	(64,4)
find permutation	$(events\_in\_execution, 84)$	16
decode raw banks in order	$(events\_in\_execution, 4)$	64

Decoded hits are grouped into *sector groups*, which are composed of various sensors. Each *sector group* carries a number of hits that

are guaranteed to be within certain X coordinates. Within a *sector group* hits are not sorted by X coordinate, making it faster to sort by avoiding the arrange of all hits inside the *sector group*. This also allows for quick look-up of hits in the tracking algorithm, targeting specific sector groups and searching hits only in those. Hits are sorted into pre-defined regions of the *sector groups*, then sorted by Y coordinate within the *sector group*. The whole decoding process is divided into 7 GPU kernels, where the configuration in Table 7.1 was found to be the fastest for the UT decoding.

- *Calculate number of hits*: the first kernel uses pre-defined regions in the X axis, where the regions in the center of the panel are narrower due to the increased number of tracks expected based on previous LHCb data takings. Raw banks are processed to calculate the number of hits, used to create the array to store the offsets between the hits in memory, in a coalesced manner. To process the raw banks in parallel a two-dimensional kernel is set, parallelising over the raw banks and over the number of hits in each raw bank.

- *Prefix sum*: a parallel prefix sum of the hits is implemented, specifically a *two-step Blelloch scan* composed of a reduce and down sweep operations. It results in an array with the sums of the offsets, so their positions and sizes can be obtained [75]. After doing the prefix sum the total number of hits is obtained, which allows us to pre-allocate the memory for the hits. The prefix sum is implemented here in three separate kernels, as shown in Table 7.1.

- *Pre-decode*: using the data structure created during the prefix sum, the coordinates of the hits for each raw bank can be decoded. Parallelising over the raw banks and over the number of hits in each raw bank, the strip information to get the subdetector region, panel and sector of the hit is extracted. Using this information the *X at Y=0*, and *yBegin* coordinates are decoded to delimit the hit in the Y axis.

- *Find permutation*: it calculates the required permutations to sort the hits by Y coordinate, based on their decoded Y coordinate limits. Hits are sorted within every group defined by the previously decoded X coordinate. An insertion sort is implemented in shared memory, storing the Y coordinate in it, and parallelising over the hits found in each sector group.

- *Decode raw banks in order*: to perform the actual decoding of the UT hits, a gather operation is used. It gets geometry and panel information from the subdetector, and stores the parameters in a coalesced manner. The hit information is stored in its correct

position using the pre-defined X coordinate regions and the permutations calculated in the previous kernel. For this kernel, a parallelisation over the hits found on each layer is implemented.

7.5 COMPASS TRACKING ALGORITHM

The *Compass* tracking algorithm was designed so it can be configured by two parameters: the number of sectors to search for hit candidates, and the number of valid found candidates to test to form a track. Different configurations of these parameters give a configurable trade-off between computing and physics efficiency performance.

Compass is focused on the SIMD many-core parallelism offered by GPUs and its memory characteristics to develop a high-throughput algorithm. To achieve high-throughput, tracking on thousands of tracks in parallel is performed in real-time, where each particle trajectory can be computed independently one from each other. The implementation benefits from this to design the algorithm around a SIMD model, where GPUs implement it in a SIMT (Single Instruction Multiple Thread) execution model. The operations needed to calculate the particle trajectories require arithmetic and matrix operations with single precision floating point numbers, where GPUs have shown to offer speed-ups in scientific computations. The decoded window ranges stored are accessed in a SoA data layout. Other multi-threaded architectures like modern x86-64 should also benefit from a SoA layout, as the access pattern by the different threads also benefit from data locality and coalesced access. The NVIDIA Profiler was used to optimize and find the spots to parallelize.

Compass is divided in two main components: searching for the UT window ranges in the indicated sectors, and using those window ranges to perform the tracking. In both cases, VELO tracks are used as input, and are extrapolated to the UT panels.

7.5.1 *Search UT windows*

UT window ranges are defined by the indexes of two hits, one at the beginning of the window and the other at the end, where hits in between these two are considered for creating a track. The search for UT windows is performed using the information about how hits are sorted during the decoding. A two-dimensional kernel is used to search the windows: the first dimension parallelises over the four UT panels, where the second does it over the input VELO tracks. The kernel is defined like this to optimize for the windows' ranges to be stored in SoA layout, where different kernel configurations are tested, concluding this one to yield the best performance. Window ranges are stored in a coalesced manner for a panel, where panels are also stored contiguously between them. The two-dimensional kernel is

used to favour the access pattern, first over the panels, then over the different tracks. This configuration was found to be faster than setting the kernel the opposite way, or just parallelising over the tracks in a one-dimensional kernel.

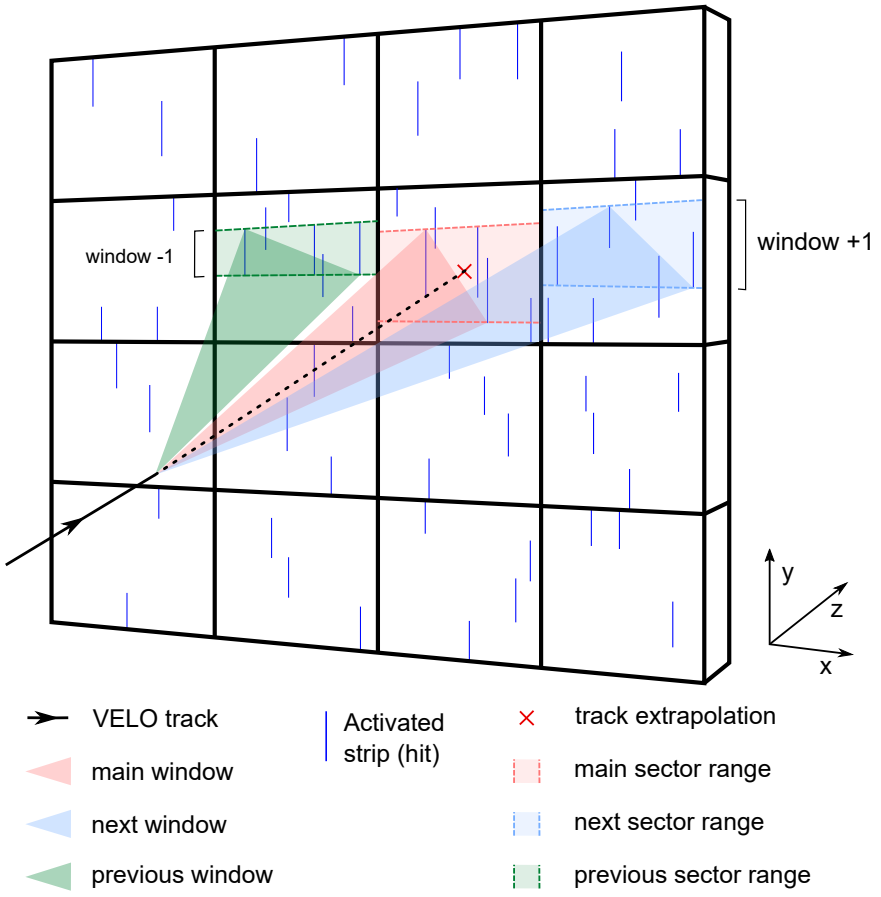


Figure 7.3: UT window ranges: representation of a VELO track extrapolation to a sector. Window ranges are set for the sector and its neighbours. Several hits lie within the range of the windows, which are considered for UT tracking

For each input VELO track, the extrapolation to the UT panels is calculated taking into account the magnetic field. The extrapolation defines the sector group in the UT to search for. Since sector groups are sorted by X into known regions, a binary search is used to efficiently locate the region where the extrapolation is pointing to. With the region delimited by X, a tolerance window based on the VELO track extrapolation is used to delimit the Y region. Searching with two binary searches over the Y axis, one to delimit the beginning of the region and another to delimit the end of it, leaves us with the window range that indicates the valid UT hits for the associated VELO track. Only two pointers to the hits are used to indicate a window range, instead of storing all the valid hits or pointers to all valid hits. Finally the window range is refined by checking the hits to be valid within

the VELO tolerance window. Iterating forward for the beginning hit, and backwards for the end hit, hits are tested to meet the conditions for the VELO track tolerance. This calculation is performed here to reduce the window ranges, which was found to be faster compared to only perform it in the tracklet finding kernel. When computing the tracking kernel combinations between the hits in different panels are tested. Using a larger window range during the tracking has a larger impact in the complexity to compute the kernel compared to refining the window range during the window search. As the hits in a sector group are not sorted, the VELO tolerance check has to still be performed again in the tracking kernel because hits could be out of the tolerance window.

When looking for window ranges, a VELO track may be outside the UT acceptance region or may be directed in backwards direction, making the track unsuitable for UT tracking. When a thread is assigned to a track that meets any of those conditions, the whole thread is left unused until the rest of the threads in its *warp* finish finding the window regions. Some threads are left unused for every event, lowering the throughput capacity of the algorithm. To maximize thread occupancy, an array of pointers to tracks in shared memory is used, which is filled with valid tracks only. The array is filled until it holds at least the same amount of tracks as number of threads per block. Windows are searched parallelising over the array of pointers to valid tracks, maximizing thread occupation.

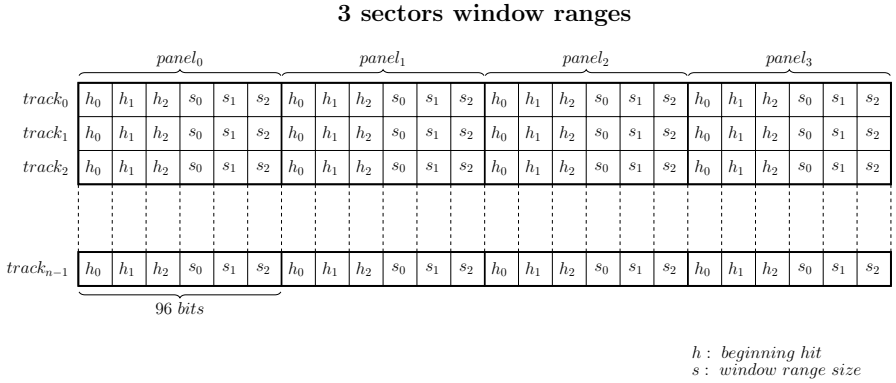


Figure 7.4: Memory layout of window ranges. A beginning hit, and a size are stored per window range, using 16 bits for each element. In this figure, a 3 sectors window ranges is shown, where each elements has a size of 16 bits, making it a total of 96 bits for all the elements of a panel.

The window search is implemented to look for hits in one, three or five sectors. This is done because it was found that the number of hits selected in only one sector to be insufficient to achieve good enough physics performance. The selected sector and its neighbours are used to get hit candidates, as can be seen in the Figure 7.3. If the extrapolated VELO track is pointing to a sector close to the borders of

the UT panel, less sectors are searched. The window ranges are stored in a pre-allocated memory space, as the number of sectors to use and VELO track is already known, so they can be stored in parallel for every thread. When an invalid window range is found, it is stored with `(-1, -1)`, indicating that no valid hits were found. By doing this the kernel presents a lower branching ratio, leaving a similar code path for all tracks searching the windows, making it efficient for GPUs.

Finally, window ranges are stored as pairs composed of a beginning hit and the size of the window. As the implementation iterates over the hits in the window, knowing in which window the hit starts and the size of the window is all the information needed to access the hits. To store the hit and the size of each window two signed 16-bit types (`short`) are used. The hit index is set to be relative to its own track, for all the possible indexes to fit in a `short` type, thus reducing the memory footprint. Hit pointers and window range sizes are stored grouped so all hits are contiguous between them, and per track, as can be seen in Figure 7.4.

7.5.2 *Tracklet finding*

To perform UT tracking, a search for the best compatible hits needs to be performed in all the UT panels to form a tracklet. A tracklet is composed of at least 3 hits on different panels. The combination that best matches the extrapolation from the VELO track is searched, considering the influence of the magnetic field that introduces a small kink in the particle trajectory. The window ranges calculated in the previous kernel are used to find a tracklet of one hit per UT panel, allowing for one missing UT hit. The main complexity of *Compass* lies in the tracklet search, where compatible hits between all panels are tested for compatibility, increasing the multiplicity of the combinations.

When a valid hit is found in the first panel, it is selected to be combined with a valid hit from the third panel. If a valid hit is also found in the latter, the slope formed between them is calculated. The just calculated slope and the one of the VELO track are used to define a tolerance window in the second and fourth panels. Compatible hits are searched in these panels to form the final tracklet, as can be seen in Figure 7.5. Finding a third hit is enough to from a tracklet, where a tracklet of four is preferred if it is found. The complexity of tracklet search is $O(n^3)$, as the search for third and fourth hits are not nested between them. The tracklet search is performed both in forward and backwards directions, where the same algorithm is applied changing the order of the panels. Forward and backwards search is merged into one single loop, where hits are searched first in forward direction and if no hits are found, the backwards direction is tested to find a tracklet.

The algorithm may be configured to use more than one window range, in this chapter for one, three or five window ranges. Instead of looping independently over the ranges to find a tracklet, these are combined into one single loop, as if these were one single range. A pointer to a selected hit within a window range is used to iterate. The ranges are combined so the central one is used first, then its immediate neighbours. If five sectors were selected, the sectors in the extremes are searched the last. Forward and backward searches are combined, as it was found this way of iterating over the hits to be faster than performing two separate searches for forward and backward direction, as thread divergence is removed. The searches for every VELO track are parallelized, where all the threads in a *warp* will have to wait if a divergent branch is encountered in one of the threads. When the hit search is split into two loops, a divergent branch is introduced if different tracks are searching in forward and backward direction within a *warp*. A small divergent branch is introduced at the beginning of the loop when combining the window ranges. This is done to set the pointer to the correct hit, which allows the *warp* to run all tracks in a parallel fashion even if they diverge in both ranges or direction.

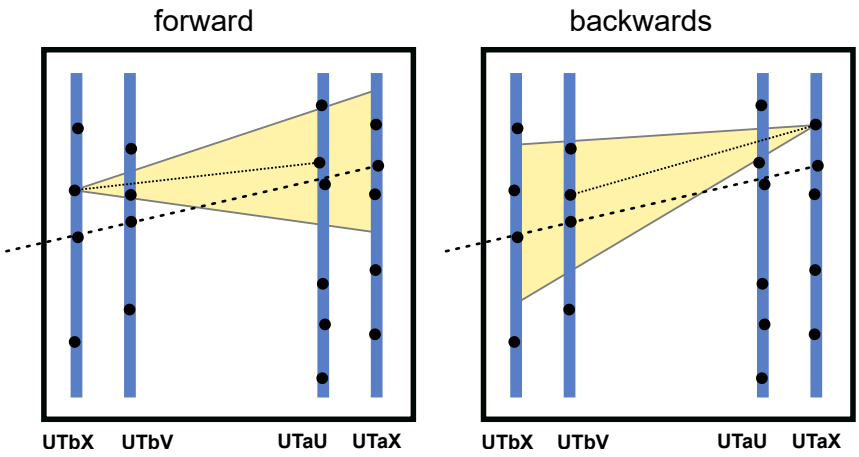


Figure 7.5: Tracklet finding kernel. Combinatorics between all 4 panels when searching for hits candidates to form a tracklet are shown. The fine dotted line represents the slope between the two first hits found in the first and third panels. The coarse dotted line represents the VELO track slope. A tolerance window defined by them is calculated to search for a tracklet.

Compass implements a configurable number of search hit candidates that will be considered. When a valid tracklet is found, if more than one candidate was configured, the next valid hits within the window ranges are tested to form a different tracklet. For every tracklet the χ^2 fit of the track is obtained in combination with the VELO track. If more than one tracklet is found, a selection is performed favouring tracklets with 4 hits instead of 3, and with the lowest χ^2 fit value.

The algorithm keeps searching for a better tracklet according to the configured hit candidates value.

Compass is parallelised over the VELO tracks, where each thread processes the tracklet search for each track. When processing VELO tracks, a similar filtering mechanism is applied as when searching for the window ranges explained in subsection 7.5.1. It differs in the conditions to save a valid track, looking for the track to be within UT acceptance, not backwards and to have at least one valid window range. Only the size of the window range is checked to be different from -1 to indicate a window range with at least one valid hit.

The implementation takes advantage of the GPU shared memory to cache the window ranges, as these are accessed during the tracklet search. A shared memory array of size $num\_threads \times num\_panels \times size\_window\_range$ is used to accommodate all the window ranges in a block. As in the search window ranges kernel, the window ranges are stored using a signed 16-bit type to save in memory. When processing a valid track, the window ranges for that track are copied to its correct position relative to the block size into shared memory, where only the pointers to the shared memory array are used afterwards. This was found to be faster in all the tested configurations and GPUs.

When a final tracklet is selected as the best one, the found hits are stored and associated to its VELO track as a VELO+UT track. Alongside the hits, the charge of the particle, calculated from the momentum of the track from the χ^2 fit, and the index of the track within the event are stored, obtained by atomic addition of the track number for this event.

7.5.3 *CPU implementation*

A CPU version of *Compass* tracking is implemented to compare its computing performance against the baseline GPU implementation. To port the algorithm part of the structure of the algorithm is modified. The GPU specific optimizations are removed, which cannot be exploited in a non-GPU architecture. On the baseline GPU version thread divergence is minimized and store various structures into shared memory, whereas the impact of branches is minimized by design in a CPU architecture compared to a GPU architecture [93] [49]. The impact of using shared memory and caching the window ranges in the ported version is considered to be better managed by the large caches found in a modern CPU, compared to the ones in the GPUs. The computation of searching window ranges and tracklet finding is not split into separated kernels, where the window searches are calculated for every VELO track in-place before doing the tracklet search. This is done to benefit from cache locality, as the just calculated window ranges will be used by the tracklet search algorithm.

7.6 EXPERIMENTAL EVALUATION

This section covers the performance and physics efficiency evaluation of the proposed algorithms. Multiple micro-benchmarks are conducted using different configurations for both the number of sectors and the number of candidates.

7.6.1 *Experimental setup*

Four GPUs and a x86-64 CPU were used for the benchmarks. Two consumer-grade GPUs of different generations and two server-grade GPUs are employed. A dual socket server-grade CPU is used for the *Compass* tracking CPU implementation. The specifics of the hardware are detailed in Table 7.2.

The software relies on CUDA 10.0 and `gcc` 7.3.0 under the `-O3` optimisation flag. The following compilation flags were used: `-use_fast_math -expt-relaxed-constexpr` and `-maxrregcount=63`. The use of those flags were beneficial for the overall execution time of the algorithm [109].

All the benchmarks use the same sets of Monte Carlo simulated events, generated using the LHCb simulation framework. Two different testbeds of events are evaluated: the *minbias* set for throughput performance and the *BsPhiPhi* to check reconstruction efficiency. The *minbias* (minimum bias) set is a realistic simulation of the current expected physics, where data rate and therefore computing performance obtained with it match the realistically expected one. The *BsPhiPhi* set contains more tracks from the rare decay $B_s \to \phi\phi$. This allows to determine the track reconstruction efficiency for these physically interesting decays with higher statistical significance. It is important to highlight that the same reconstruction efficiency can be achieved in both testbeds. However, more *minbias* samples would be needed to obtain the same number of tracks from the rare $B_s \to \phi\phi$ decay. Each set contains 1,000 events. For the throughput measurements, 40 iterations over the *minbias* events are performed to get a sustained throughput. Both server grade GPUs are set to ECC (Error-Correcting Code) memory disabled. The evaluation metrics shown in this chapter correspond with the average value of 10 consecutive executions.

7.6.2 *Compass tracking physics performance and throughput*

The computing performance of the algorithm is measured in terms of throughput of events per second. Different configurations of the algorithm are evaluated, taking measurements when looking into 1, 3, and 5 sectors and different number of hit candidates for 1 to 16 when looking for a better tracklet.

The obtained physics efficiency is shown in Table 7.3 for the long and VELO+UT tracks. A focus on the long tracks is given, as these

Table 7.2: GPU and CPU hardware employed for the evaluation. Two high-end consumer graphics cards (GeForce GTX 1080Ti and GeForce RTX 2080Ti), two server-grade cards (Tesla T4 and Tesla V100), and an Intel Xeon CPU are compared. It shows the number of cores of each processor, where for the GPUs it counts the CUDA cores only (no RT cores or Tensor cores are used in the benchmarks). The MSRP (manufacturer suggested retail price) is used for each hardware unit used here. The price for a single Intel Xeon CPU is shown, whereas for the benchmarks a dual socket server with two Intel Xeon CPUs is used. This is reflected in the price performance figure.

Unit	# cores	Max freq. (GHz)	Cache (MiB - L2)	DRAM (GiB)	TDP (W)	MSRP ($)
GeForce GTX 1080 Ti	3,584	1.67	2.75	10.92 GDDR5	250	699
GeForce RTX 2080 Ti	4,352	1.54	6	10.92 GDDR5	250	1,199
Tesla T4	2,560	1.59	6	16 GDDR6	70	2,350
Tesla V100 V100	5,120	1.37	6	16 HBM2	250	8,899
Intel Xeon E5-2678W v3	20	3.50	25 (L3)	64 DDR4	160	2,145

are the preferred ones for analysis. Long tracks carry more information about the momentum resolution. The VELO+UT tracks are also analyzed, as these are constructed with the two main inputs of the *Compass* algorithm, VELO tracks and UT hits [89]. Note how for the 3 sector cases, when searching for more hit candidates, the physics efficiency improves. The biggest improvements are achieved in track reconstruction efficiency, where the clone rate increases by less than 0.1% in all cases. Note how the reconstruction efficiency gains flattens when using more hit candidates. While the number of hit candidates is increased exponentially, the track reconstruction efficiency gains do not follow the same increase pattern, but the opposite. This behaviour matches our expectations, as in most of the cases, the best tracklet is found in the first set of hit candidates, and therefore, the subsequent ones do not yield a better hit tracklet as often. Calculating the subsequent tracklets has an impact on the throughput performance even if no better tracklet is found, where the physics performance does not improve. The fake rate decreases when using more sectors and candidates, with differences in the range of 1% across the whole scope of benchmarks. Note how the impact of both changing the sectors and candidates has little effect on the clone and fake rates, whereas it has a big impact in the reconstruction efficiency rate.

Table 7.3: Comparison between searching in 1, 3 or 5 sector groups, and using 1 to 16 hit candidates. Two type of tracks are compared: long tracks and VELO+UT tracks. For each type of track, the track reconstruction efficiency and track clone rate achieved are presented. The obtained fake rate for each case is also shown.

Number of sectors	Number of candidates	Long tracks		VELO+UT tracks		Fake rate
		reco. efficiency	clone rate	reco. efficiency	clone rate	
1 sector	1	71.91%	0.36%	61.88%	0.32%	7.73%
	2	76.53%	0.36%	69.99%	0.32%	7.78%
	4	79.09%	0.31%	74.31%	0.32%	7.70%
	8	80.36%	0.34%	76.58%	0.35%	7.61%
	16	80.52%	0.34%	77.04%	0.35%	7.52%
3 sectors	1	84.70%	0.39%	66.87%	0.32%	7.64%
	2	90.07%	0.38%	75.61%	0.33%	7.62%
	4	93.31%	0.35%	80.32%	0.32%	7.52%
	8	94.72%	0.36%	82.66%	0.35%	7.43%
	16	94.94%	0.36%	83.19%	0.35%	7.33%
5 sectors	1	85.23%	0.39%	67.10%	0.31%	7.70%
	2	90.65%	0.38%	75.84%	0.32%	7.67%
	4	93.89%	0.35%	80.52%	0.32%	7.56%
	8	95.27%	0.36%	82.87%	0.35%	7.47%
	16	95.49%	0.36%	83.40%	0.35%	7.38%

The reconstruction efficiency achieved when searching in one sector does not reach 90% for long tracks nor 80% for VELO+UT tracks for any number of hit candidates. These reconstruction efficiency does not meet the requirements for the LHCb UT reconstruction, and therefore, the one sector configuration is discarded in the following analysis.

Figure 7.6 plots the differences in throughput between all the configurations, using 3 and 5 sectors, and from 1 to 16 candidates. Note how searching for more candidates decreases the throughput, as it needs to iterate over more hits in a $O(n^3)$ algorithm to find a better hit tracklet. The performance degrades more when using more candidates, contrary to what was observed with the physics performance, where the gains were very small by doubling the number of candidates when using the bigger number of candidates. When searching for more hit candidates, the hit tracklet needs to be constructed, and their χ^2 calculated, even if for most of the cases the last calculated hit tracklet does not improve over the previous one.

The difference in performance between the four evaluated GPUs devices is highlighted. The 1080Ti and Tesla T4 have a comparable performance despite of the difference in terms of number of cores. The comparable performance between the two cards is attributed to the bigger cache size encountered in the Tesla T4 and its faster

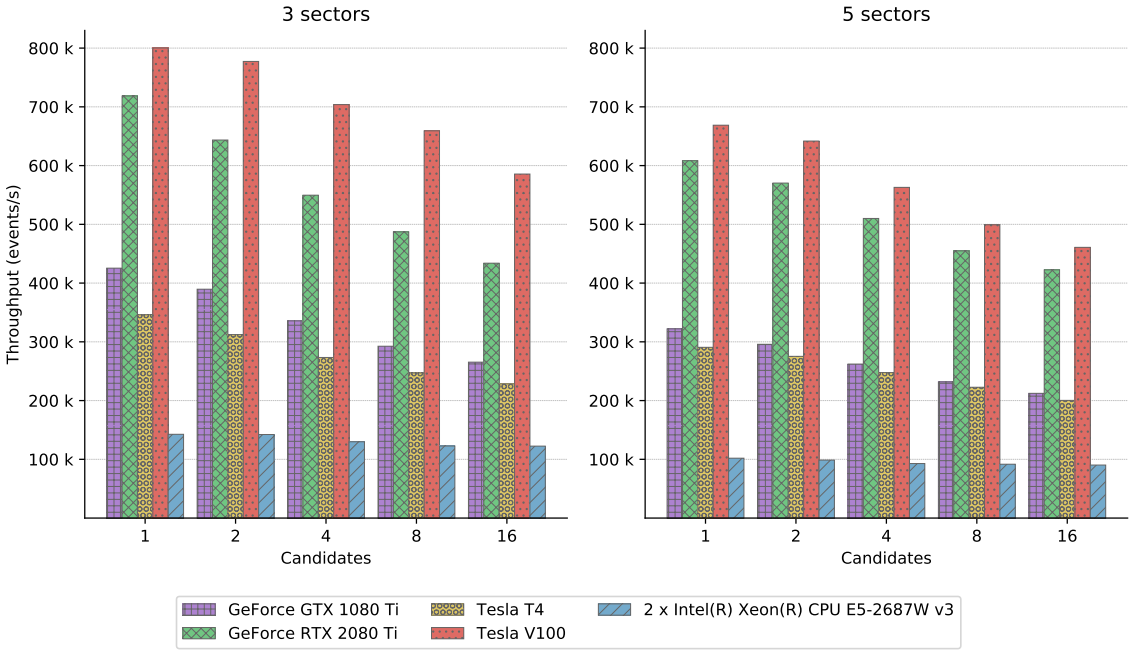


Figure 7.6: 3 vs 5 sectors *Compass* tracking comparison. Throughput comparison between the two consumer grade GPUs, two server grade GPUs and a dual socket Intel Xeon CPU, comparing with 1 to 16 number of hit candidates. The throughput shown here corresponds to running the *Compass* algorithm. The figure in the left plots the throughput when looking for hits in 3 sectors. The right figure depicts the throughput when looking for hits in 5 sectors, adding an extra neighbour sector on each side with respect to the 3 sectors case.

GDDR6 memory. The difference in thermal design power (TDP) is very significant, where the 1080Ti consumes 3× more compared to the Tesla T4 to deliver a comparable throughput. The difference in performance between the 1080Ti / T4 compared to the 2080Ti is bigger than the difference found between the 2080Ti and the Tesla V100, with closer comparable performance when using 5 sectors compared to 3. Tesla V100 outperforms the rest of the GPUs due to its High Bandwith Memory (HBM) and increased number of cores, having double the number of cores compared to the T4, 15% more compared to the 2080Ti, and 30% more compared to the 1080Ti as show in Table 7.2. One generation difference for the high-end consumer cards yields double the throughput for the 1080Ti compared to the 2080Ti for this algorithm.

Note the difference in performance for comparable physics efficiency on different results. A comparable physics efficiency is observed in the long tracks between the 5 sectors - 8 candidates case, and the 3 sectors - 16 candidates case. Taking the Tesla V100 as reference example, a difference in performance of roughly 15% (500k vs 585k)

is observed, whereas the difference in physics efficiency is below 1%. The throughput differences change between the tested hardware for different number of candidates and sectors. Not that for comparable physics performance, the 5 sectors version performs better in throughput.

The *Compass* tracking algorithm is ported so that it runs on architectures other than the GPUs, to perform a cross-architecture tracking performance comparison. The CPU version differentiate from the GPU version in the implemented optimizations but computes the same algorithm and uses the same data layout and access patterns, as explained in 7.5.3. OpenMP is used to parallelise over the events and tracks, following the same parallelisation scheme as in the GPU version. All cores are ensured to be used in both the CPU and GPU versions for the comparison. Note how the parallelisation differs in the SIMD approach of the GPUs compared to the multi-threaded version of the CPUs, where the CPU version relies on the improvements made by the compiler due to the SoA data layout to exploit the SIMD capabilities of the CPU. The performance difference between a dual socket Intel Xeon CPU and the 1080Ti GPU and Telsa T4 is found to be up to 3× faster for the GPUs, up to 6× faster for the 2080Ti, and more than 6× faster for the V100. Note how the CPU version of the algorithm degrades less its performance compared to the GPUs when increasing both the number of sectors and candidates. This is attributed to better branch prediction in the CPU, and the impact of divergent threads on the GPU, where the GPU runtime performance is affected more by the increased number of branches, and the work imbalance keeps warps active with low occupation, due to the increased number of candidates and sectors.

Figure 7.7 plots the price performance ratio for the different target GPUs. This figure shows the case for best physics performance with 5 sectors, using 16 hit candidates. It is normalised to the Tesla V100 and compares the other analysed hardware accelerators in terms of achieved speedup in terms of price/performance. Note how the price performance achieved for all the evaluated hardware is given for its MSRP with the prices shown in Table 7.2. Tesla V100 performs the worst in all the tested GPUs for its price performance, while it achieves the best throughput. Note the comparable price performance between the server grade Tesla GPUs compared to the consumer GPUs, where the consumer GPUs perform around 5× better than the server grade ones despite their differences in throughput. Note a 1.7× speedup between the Tesla V100 and the Tesla T4, and a 1.15× speedup between the 1080Ti and the 2080Ti, being the consumer grade GPUs close in price performance despite the 2080Ti doubling the 1080Ti in throughput. The achieved price performance speedup between the Tesla V100 and the 1080Ti is 5.9×, and 6.7× for the 2080Ti. The 2080Ti obtains the best price performance due to the achieved high

The prices shown in this chapter are collected from those recommend by NVIDIA and Intel web site or Amazon.com otherwise.

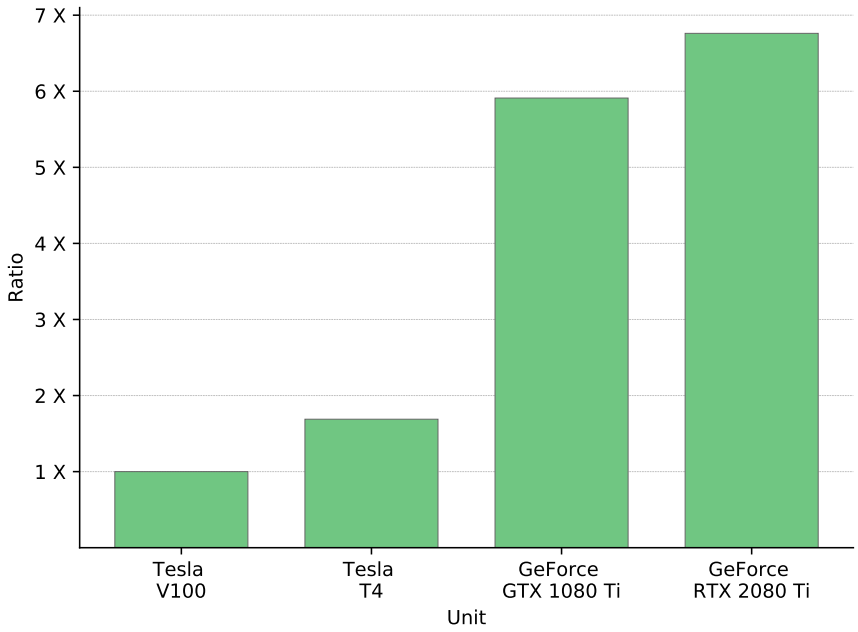


Figure 7.7: Price performance ratio for *Compass* in GPU. All prices are factored to MSRP price indicated in Table 7.2. The price performance of the 5 sectors case is compared, for the best physics efficiency case with 16 candidates.

throughput and low unit price. The 2080Ti delivers a throughput close to the Tesla V100 with significant less price due the lack of some server-grade characteristics such as HBM or ECC memory.

7.6.3 *UT decoding and tracking performance*

Figure 7.8 shows the speedup achieved for various iterations of optimizations, compared to the initial GPU implementation. Various small improvements and optimizations are grouped into the 11 steps presented in Figure 7.8. The first working version that implements the main ideas of the algorithm is refered to as *baseline implementation* and it applies various optimization on top of it to achieve the final 2.6× speedup. For *floats unroll*, the biggest improvement of 35% is observed. Various small modifications were first applied to the algorithm: mainly changing all the floating point variables to single precision ones, unrolling some loops manually, and by giving compiler hints with the use of `#pragma`. Note how the change from double to single precision does not affect the physics efficiency. The *reduced complexity* of the window range search was done by splitting the algorithm in various kernels and re-writing the tracklet finding to be simpler to process when searching in more than one sector, to get a 28% improvement. The window ranges storage was improved to be *windows SoA* to get an extra 15%, and configured it to store only

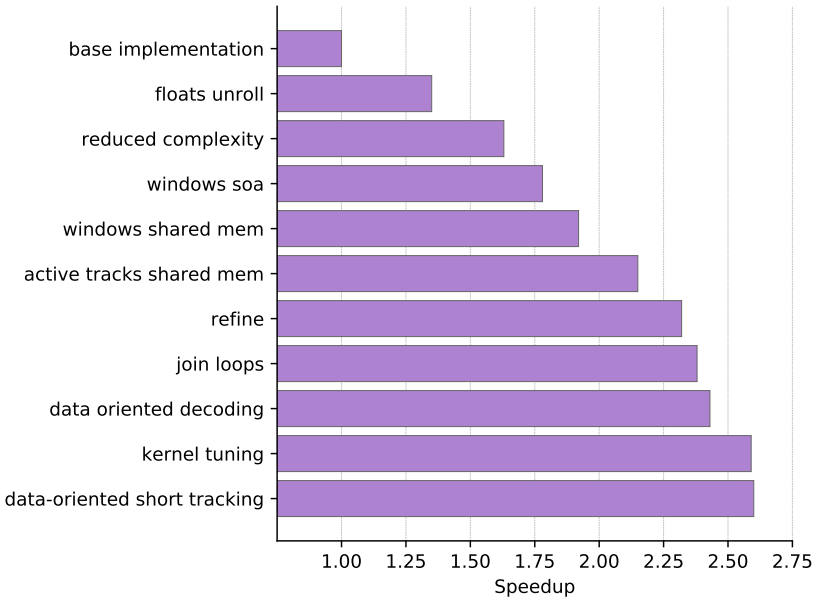


Figure 7.8: Incremental optimizations speedup. Speedup achieved after applying different optimizations to the baseline code. A maximum speedup of 2.6× is achieved in the final version, compared to the baseline implementation. Various small optimizations and changes are grouped into steps.

one hit and the size of the window, sorting them to be efficient for the access pattern. The *windows to shared memory* are copied to cache them and improve the access pattern when searching the tracklet. The speedup achieved by filtering the tracks in the shared memory array is 23%, shown in *active tracks shared mem*. When calculating the window ranges, the window is *refined* by checking the hits in both extremes, instead of calculating all the window range validity in the tracking algorithm. The complexity of the tracklet finding was further reduced by *joining the loops* and reducing thread divergence, where it got to 2.37×. Various small optimizations were grouped to the raw bank decoding, making the data types smaller, aligned and more efficient to be a *data oriented decoding*. An extra 16% was improved by *tuning the kernel* parameters of all the kernels in the decoding and *Compass*, changing to multi-dimension kernels and changing how the kernels are parallelised. Finally, The memory footprint was reduced and the copies made faster by reducing further the data types, by storing types in signed 16-bit instead of 32-bits structures to get the final overall speedup of 2.6×.

Figure 7.9 depicts the runtime distribution of both kernels used to perform the decoding and the kernels of the *Compass* tracking algorithm. The distribution is shown for the best physics case, 5 sectors - 16 candidates, where it encountered similar runtime distributions when using different configurations and different GPUs. Note how

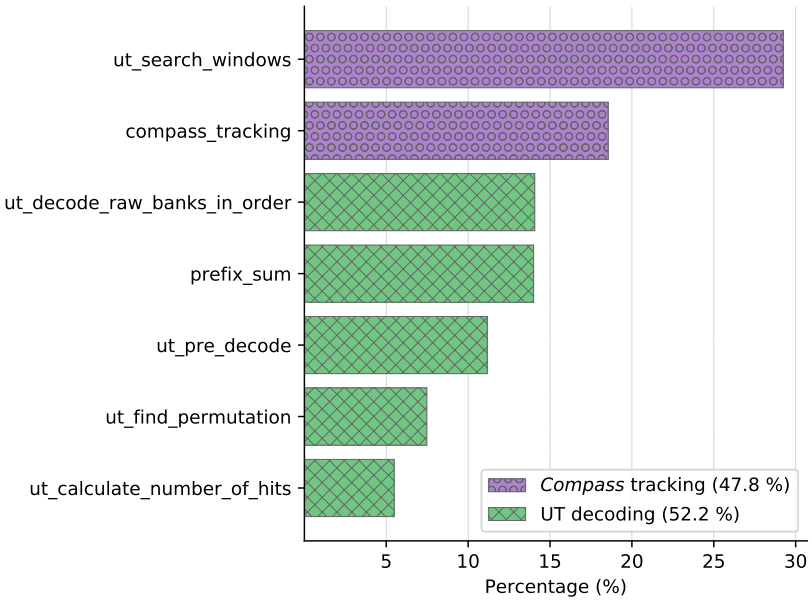


Figure 7.9: Kernels time contribution. Runtime distribution of all the kernels used to compute the decoding and *Compass* algorithm. The best physics efficiency case is used here, with 5 sectors and 16 candidates for the NVIDIA 2080Ti case.

Compass tracking runtime is dominated by the window searching algorithm compared to the tracklet finding. The refining of window ranges was moved from the tracklet finding to the window range search, increasing the time contribution of the kernel while improving the overall throughput. Note how the complete decoding of the UT hits accounts for more than half the time needed to compute the whole UT sequence.

Finally the complete implementation explained in this chapter is shown, with the decoding and tracking in GPU compared to the equivalent algorithms found in LHCb baseline implementation. It is acknowledged that the results compared here have changed and improved since the publication of these numbers in [47] used for the comparison, where more recent results are not found or published. Comparable conditions are set as those found in [47], where the same Global Event Cut is applied in this implementation, filtering a selection of events, at the beginning of the chain, thus reducing the amount of processing the tracking algorithms need to do. Data preparation kernels are added after the full UT chain is processed, in the form of a prefix sum and consolidation steps to leave the tracks in coalesced memory for the algorithms using UT tracks as input. The LHCb baseline implementation uses a Intel Xeon E5-2630 v4, which delivers a top throughput of 12,400 events per second for the full sequence. Combining the time contributions of the UT decoding and tracking for peak throughput yields the results shown in Figure 7.10.

This CPU differs from the one used for the benchmarks.

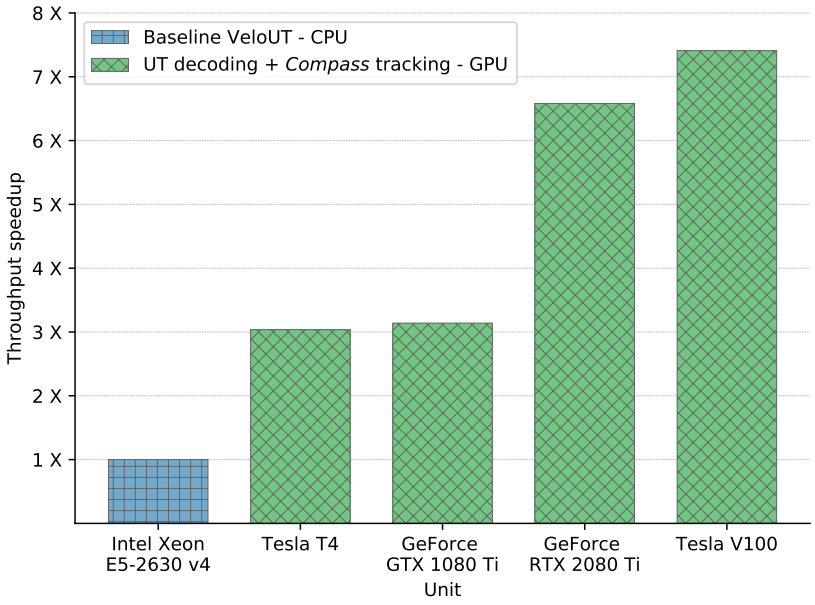


Figure 7.10: Basline LHCb vs GPU decoding + *Compass* tracking throughput speedup comparison. Throughput speedup of the full UT chain of kernels, including the decoding and *Compass* tracking, compared to the baseline LHCb CPU implementation as stated in Section 7.10. The LHCb baseline (blue) is compared with the *Compass* over different GPUs (green).

These results are compared to the full UT decoding and *Compass* tracking presented in this chapter. The throughput speedup shown corresponds to our *Compass* implementation using the configuration for 5 sectors and 8 candidates. Both the Tesla T4 and 1080Ti achieve roughly a 3× speedup, where the latter performs slightly better than the T4. The 2080Ti achieves a speedup of 6.5× and the Tesla V100 achieves the best speedup at 7.4×. The obtained physics results in both implementation are comparable, but yield different results due to the different algorithms used.

7.7 SUMMARY

The presented algorithm, *Compass*, is designed for parallel GPU architectures with focus to perform efficiently on GPUs. It is designed so that it maximises throughput processing on GPUs by being data-oriented, minimizing branching, reducing the memory footprint of the algorithm and taking advantage of the architectural characteristics of GPUs.

A SIMD parallel UT raw data decoding algorithm is presented; it is data-oriented and specifically optimized for GPUs. It demonstrated a new hit organization that stores hits in SoA, in a parallel and coalesced manner, where groups of hits are sorted into regions for fast decoding.

It benefits from the new hit organization to search efficiently for sector regions, defining window ranges that indicate where compatible hits are found. Windows are stored efficiently for parallel architectures.

Compass is designed to be configurable in both number of sectors to search for, and number of hit clusters to test for a tracklet. The physics efficiency results are shown when searching in one sector, proving it to yield too low reconstruction efficiency rate to be considered for performance benchmarks. The performance for searching in three and five sectors is compared and tested with different number of hit candidates. The algorithms are validated with Monte Carlo simulated data to verify the physics performance of the results, getting comparable physics performance.

A CPU tracking implementation was developed and analysed our algorithm in different parallel architectures, focusing on GPU architectures and comparing them against the parallel CPU implementation of the same algorithm. The differences in performance across the analysed hardware are shown. A physics performance close to 95% in track reconstruction is achieved with various configurations of the algorithm, where a configuration using 5 sectors and 8 hit candidates yields a throughput of 231k events per second in the 1080 Ti, 222k in the Tesla T4, 454k in the 2080 Ti, 499k in the Tesla V100 and 92k in the dual socket Intel Xeon CPU, for the *Compass* tracking. The 5% of tracks that were not reconstructed correctly do not satisfy the assumptions and selections made in this algorithm. These are not due to computational precision, as has been verified switching from single to double precision obtaining the same results.

This configuration is considered to be the best trade-off for this algorithm considering the achieved physics efficiency and the performance. The baseline LHCb results are compared for the full UT decoding and tracking, where our GPU implementation delivers up to 7.4× more throughput with the Tesla V100, and 6.5× when comparing with 2080Ti.

8

VECTORIZED SPMD ALGORITHMS

The Allen's framework was designed around massively parallel hardware architectures, specifically for GPUs. These architectures benefit from data-parallel algorithms that are well suited for parallel computations. Algorithms developed to compute the HLT1 with Allen's framework follow a data-oriented design: its data layout is optimized with small aligned structures, and to have the data that is most accessed coalesced, so that groups of threads access a contiguous chunk of memory efficiently. Data access and processing of these algorithms is optimized to have a small memory footprint, and the GPU shared memory is used whenever possible to further reduce the latency to access these structures. While these principles have been demonstrated to be successful with GPUs in the context of high-energy physics under Allen's framework, its performance under other architectures remains to be exploited.

The same principles of data locality and optimization apply to other parallel architectures. For instance the main target architecture in high-energy physics, x86 processors, have a high degree of parallelism that can be leveraged by the programmer. Optimizations for multi-threading are common in current software in HEP, and efforts to utilize the vector units are developed to achieve higher throughput per chip. Vectorization or SIMD processing allow the programmer to compute multiple data that will compute the same instruction at the same time. This principle is similar to what GPUs use, where groups of threads, *warps* in the CUDA programming language, compute multiple data over various thread that have simpler control units.

Allen's framework design allows the chain of algorithms that compute the full HLT1 to compile for both GPU and CPU architectures [2]. Compilation for CPUs in Allen supports basic multithreading, and does not vectorize the algorithms; any vectorization performed by the compiler is in the form of auto-vectorization, which often give poor performance gains. Vectorizing the source often requires the writing of intrinsics and manual tuning for the compiler to be able to efficiently vectorize. Support for different vector instruction sets such as SSE4, AVX or AVX-2, which are only available in different processors, requires extra effort or the intermediate library to support it. The same target source code in Allen is able to exploit all threads in a GPUs through its *warps* in an efficient manner, but when compiled for CPUs these do not map to vector lanes, and the vectorization units will remain underutilized. Explicit support in the form of vector

instructions, intermediate libraries or directives is needed to efficiently exploit these resources.

This chapter is structured as follows: In Section 8.1 the Intel Implicit SPMD Program Compiler is introduced, Section 8.2 explains how the Allen framework is adapted to include ISPC algorithms, Section 8.3 explains how the ISPC kernels are implemented, Section 8.4 presents the evaluation of the implemented algorithms. Section 8.5 gives a summary of this chapter.

Some parts of this chapter have been published in the following journal/conference papers:

- Placido Fernandez Declara and J. Daniel Garcia. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2020. Accepted for publication at CHEP 2019 Proceedings, Adelaide, Australia

8.1 INTEL IMPLICIT SPMD PROGRAM COMPILER

SIMD programming or vectorization can enhance the performance of an application by using the available vector processing units of a processor. To achieve this, it applies the same operation or instruction in various data lanes that hold different data. For larger number of available lanes in a CPU, more performance can be achieved. A different approach from auto-vectorization, intermediate libraries, intrinsics or in-line assembly is explored in this chapter.

The Intel Implicit SPMD Program Compiler (ISPC) [116] uses a variant of the C language to write sequential-like algorithms, but its execution model executes various *program instances* that run in parallel through the vector lanes. It presents an alternative way to exploit vectorization units by writing algorithms in a similar way as for GPUs. Cross-compilation is supported for Windows, Linux, MacOS, Android, iOS, PlayStation 4 and FreeBSD. It abstracts the programmer from the different widths that may be supported by different processors. Only a flag in the compiler needs to be changed for it to target the specified vectorization instruction set.

ISPC offers a similar approach to these languages, but offers some differences:

- The source code written before, or in between `foreach()` control structures will map predictably to a hardware thread.
- The iterations of SPMD `foreach()` will map to single hardware threads.
- Inside a single kernel, `foreach()` clauses can be nested inside `for()` loops.

When implementing an ISPC program the variables that will run with different data values across the vector lanes are explicitly indicated through the keywords `uniform` and `varying`. This allows the

```
typedef float<3> float3;

export void chi2_ispc (
  uniform const size_t N, uniform float chi2 [],
  uniform float3 x [], uniform float3 y [],
  uniform const float m, uniform const float q) {
    foreach(i = 0 ... N) {
      varying float3 expected_y = m * x[i] + q;
      chi2[i] = (y[i].x - expected_y.x) + (y[i].x - expected_y.x) +
                (y[i].y - expected_y.y) + (y[i].y - expected_y.y) +
                (y[i].z - expected_y.z) + (y[i].z - expected_y.z);
    }
}
```

Code 8.1: ISPC source code sample

compiler to better optimize the generated assembly to produce a vectorized version of the source code. ISPC includes other constructs that allow to compute efficiently with the SPMD model. An ISPC example is depicted in Figure 8.1 which shows what appears to be a C language program with extra reserved words. In this example, input variables are marked as `uniform` to indicate that these will hold the same value for all the vector lanes. The `foreach` construct indicates a parallel loop that will compute a different result for the variable `expected_y` and populate the `chi2` array with different values for each vector lane.

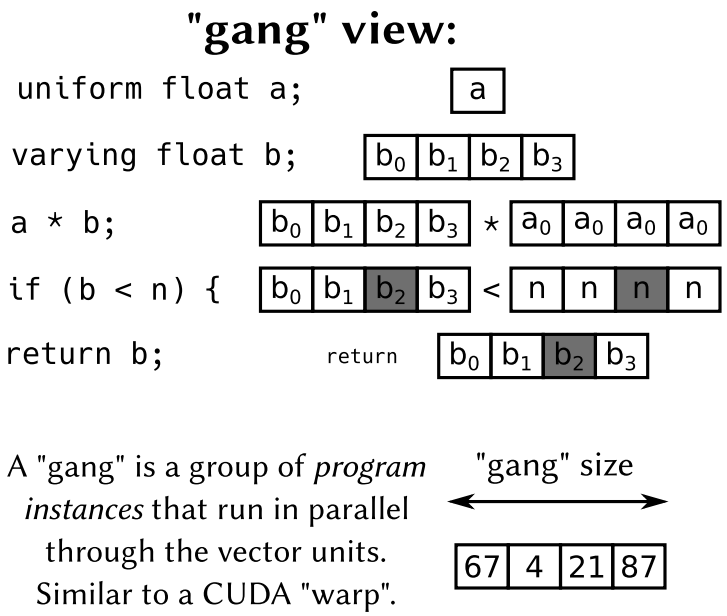


Figure 8.1: ISPC gangs

A key concept to ISPC are *gangs*. These are analogous to *warps* in the CUDA language, and are a group of program instances that run in parallel through the vector lanes. As shown in Figure 8.1 the way a *gang* interprets variables differs from it being `uniform` or `varying`.

For `varying` variables a *gang* will apply operations with the vector instructions for all the elements in parallel. This applies to control flow elements like an `if`, where if an element of the *gang* does not meet the condition it is masked to not take the result into account, even if it will be computed. ISPC adds a series of special constructs for control flow like `foreach`, `foreach_tiled()` or `cif` among others. The `foreach()` statements in ISPC indicates the part of the code that will be vectorized if the appropriate `varying` variables are used in it. It is interoperable with C/C++ which allows to compile an algorithm with ISPC and use it with a C/C++ regular source code.

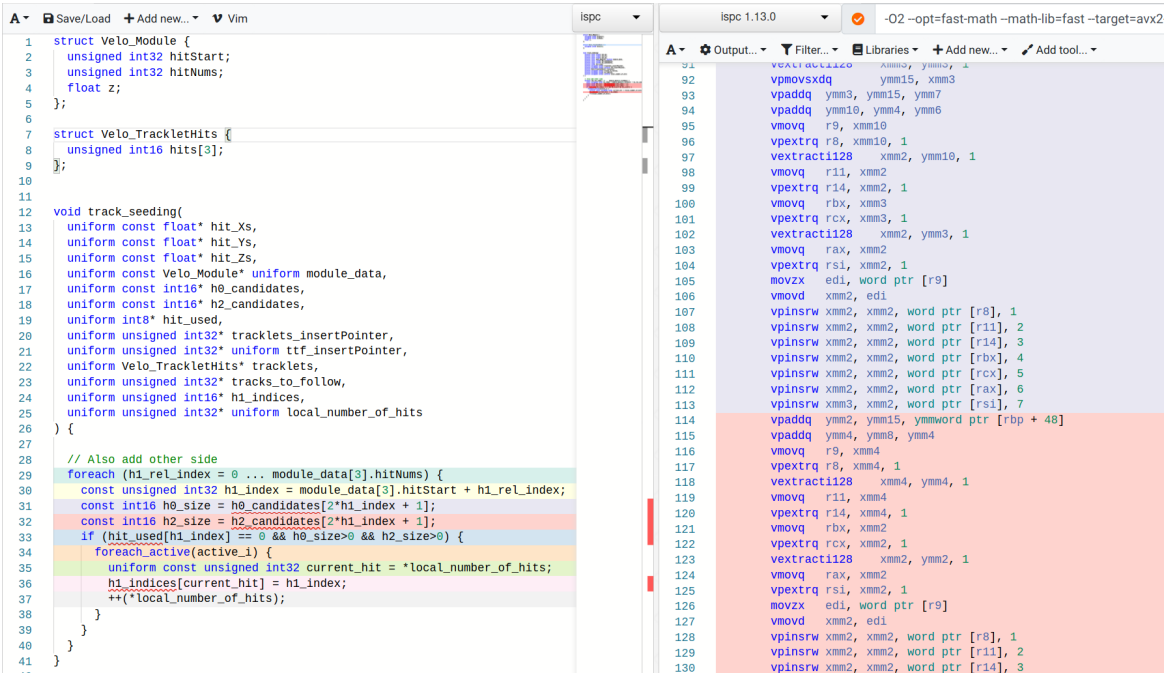


Figure 8.2: ISPC assembly vector instructions

Other SPMD programming languages - compilers include CUDA, OpenCL, Vulkan Compute or DirectX Compute Shader. The advantage of these languages is that they allow to write sequential-like kernels that are compiled to target SIMD or SIMT architectures in an efficient manner, both from the performance and programmability point of view. Figure 8.2 presents a ISPC kernel and the generated assembly on the right part of the image using Godbolt [69]. Lines are highlighted with different colors in the ISPC source code, which are then mapped using the same color to the generated assembly code. In this example similar flags to those used in this chapter are used to generate the assembly: `-O2 -opt=fast-math -math-lib=fast -target=avx2-i32-8`. The `-target` flag indicates the compiler which vector instruction set to try to compile for, followed by mask size and gang size to be used. Table 8.1 depicts the available *target - mask size - gang size* options indicating its corresponding ISA (Instruction Set Architecture).

Table 8.1: ISPC ISA targets with mask and gang size combinations

SSE2	SSE4
`sse2-i32x4`	`sse4-i8x16`
`sse2-i32x8`	`sse4-i16x8`
	`sse4-i32x4`
	`sse4-i32x8`
AVX	AVX2
`avx1-i32x4`	`avx2-i32x4`
`avx1-i32x8`	`avx2-i32x8`
`avx1-i32x16`	`avx2-i32x16`
`avx1-i64x4`	`avx2-i64x4`
AVX-512	NEON
`avx512knl-i32x16`	`neon-i8x16`
`avx512skx-i32x8`	`neon-i16x8`
`avx512skx-i32x16`	`neon-i32x4`
	`neon-i32x8`

ISPC central feature are the `uniform` and `varying` keywords. When programming ISPC kernels, one way to start using an algorithm is to declare everything `uniform`, so that the application will not vectorize at all, but it will work as a regular sequential algorithm. From there, one can analyze the loops and parts of the code that are good opportunities for vectorization and change those to use `varying` variables and pointers to indicate the compiler that those can be vectorized and hold different values inside the `foreach()` loops. This way small expensive loops can be easily vectorize, but also big loops can be adapted for vectorization. Pointers in ISPC need special attention as these take a pair of `uniform`, `varying` keywords. The first one indicates if the pointed structure is to be vectorized or not, and the second one would indicate if the actual pointer is to be vectorized or not, like the following: `varying struct* uniform pointer`, where a uniform pointer is used pointing to a varying varying data structure. ISPC also offers "coherent" control flow structures, like `cif` and `cfor`, to indicate that an specific control flow structure is expected to be coherent, this is, that the executing instances in parallel are expected to offer the same result for those control flow structures.

8.2 ADAPTING ALLEN FOR ISPC ALGORITHMS

To accommodate and include ISPC algorithms into Allen, the framework needs to be modified to adapt for CPU algorithms that support

```
...
spmd_velo_search_by_triplet_t,
spmd_velo_weak_tracks_adder_t,
copy_and_prefix_sum_single_block_velo_t, // non-SPMD
copy_velo_track_hit_number_t, // non-SPMD
prefix_sum_velo_track_hit_number_t, // non-SPMD
spmd_consolidate_velo_tracks_t,
spmd_ut_calculate_number_of_hits_t,
prefix_sum_ut_hits_t, // non-SPMD
spmd_ut_pre_decode_t,
spmd_ut_find_permutation_t,
...
```

Code 8.2: ISPC and GCC algorithms interleaved

```
cudaCheck(cudaMalloc(
    (void**) &m_dev_beamline.get(), data.size()));
// replaced by:
m_dev_beamline.get() =
    reinterpret_cast<float*>(memalign(256, data.size()));
...
cudaCheck(cudaMemcpy(
      m_dev_beamline.get(),
      data.data(), data.size(),
      cudaMemcpyHostToDevice));
// replaced by:
std::memcpy(m_dev_beamline.get(), data.data(), data.size());
```

Code 8.3: Memory allocation and copies change from GPU to CPU

kernels compiled with different compilers, in this case with CUDA's `nvcc`, `gcc` and `ispc`. Algorithms can be interleaved when configuring a sequence of algorithms for the Allen framework as shown in Listing 8.2. For the algorithms to be efficient by avoiding memory copies between the host and the device, a chain of kernels that run on the CPU is used in this chapter.

Calling a kernel in Allen is done by creating a visitor

Adding algorithms that are compiled to be executed in the GPU device is also supported, but when calling the specified GPU kernel, memory needs to be manually managed between the CPU and the GPU. If going from a GPU to CPU kernel or the other way around, a `cudaMemcpyAsync()` must be performed when invoking the algorithm indicating `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`. The allocation of all the other resources used in the Allen framework from the GPU need to be adapted and allocated for the CPU by replacing all the `cudaMalloc()` and `cudaMemcpy()` as depicted in Listing 8.3. This changes include beamline, magnetic field, geometry information, all constants used across the framework, base pointers for the data structures, and all the raw input data from the different subdetectors.

```
1  for (int event_number = 0;
2      event_number < host_buffers.host_number_of_selected_events[0];
3      ++event_number)
4  {
5    state.invoke(
6      arguments.offset<dev_ut_hits>(),
7      arguments.offset<dev_ut_hit_offsets>(),
8      reinterpret_cast<int8_t*>(arguments.offset<dev_velo_track_hits>()),
9      reinterpret_cast<int8_t*>(arguments.offset<dev_velo_states>()),
10     reinterpret_cast<ispc::SPMDPrUTMagnetTool*>(constants.dev_ut_magnet_tool),
11     constants.dev_magnet_polarity,
12     constants.dev_ut_dxDy.data(),
13     constants.dev_unique_sector_xs.data(),
14     reinterpret_cast<ispc::UT_TrackHits*>(arguments.offset<dev_ut_tracks>()),
15     event_number,
16     host_buffers.host_number_of_selected_events[0]);
17 }
```

Code 8.4: Casting types for ISPC.

The ISPC compiled kernels are added through a modified `add_ispc_library()` *CMake* function similar to the one provided by the Intel ISPC compiler. In this function the targets can be set, it handles different operating systems flags and it handles the outputs and linking of the generated library. ISPC algorithms added for Allen use their own *namespace* and can be called explicitly with `ispc::spmd_algorithm`, which allows to differentiate them from the non vectorized CPU version of the algorithm or the GPU version. To include all ISPC kernels a header file containing the files for a specific subdetector is included, which simplifies the process of adding the library. Some data types need to be explicitly *casted* for the compiler to know that the number of bits used in the framework when allocating types will match the data type used inside ISPC algorithms. ISPC uses some modified data types to be explicit about its size. For integer types `int8`, `int16`, `int` or `int32` and `int64` are used by ISPC.

ISPC includes the `unsigned` counterparts for this types

Finally when invoking an ISPC kernel some data types, like `byte` for `int8_t`, need to be *type casted* to be explicit about its type to align it with the ISPC type. The compiler will not allow `structs` to be passed as an argument unless these are *type casted* to the equivalent ISPC type. This is represented in Listing 8.4. Note how arguments are here indicated as `dev` instead of `host`. This is done to keep the same argument pointers for both the GPU and the CPU, in this case where only CPU kernels are being used the CPU simply becomes *the device*. Constant arrays (lines 10,11,12 in Listing 8.4) are also indicated as *device* (`dev_`), but can be accessed here by the CPU as the allocations are changed to use the host main memory.

```
__shared__ float s_y_begin[UT::Decoding::ut_max_hits_shared_sector_group];

if (sector_group_number_of_hits > 0) {
  __syncthreads();

  for (uint i = threadIdx.x;
       i < sector_group_number_of_hits;
       i += blockDim.x)
  {
    s_y_begin[i] = ut_hits.yBegin[sector_group_offset + i];
  }

  __syncthreads();
  find_permutation(...);
}
```

Code 8.5: Shared memory usage within Allen GPU framework

8.3 SPMD TRACKING ALGORITHMS IN ALLEN

In this chapter all kernels from the VELO and UT subdetector are adapted from its GPU implementations to CPU and ISPC implementations. For the CPU implementations a simplified translation is performed by the Allen framework, where the specific GPU constructs are replaced by equivalent standard C++ constructs. This is achieved by using a header file that substitutes the CUDA constructs in the framework.

The VELO and UT kernels are rewritten to compile for the IPSC compiler. These ISPC kernels benefit from the same principles that are applied to the GPU Allen algorithms: data-oriented algorithms that use coalesced memory, data structures that are optimized for the algorithms access patterns, reduced branching to minimize divergence, and other optimizations that are described in Chapter 7. These kernels are mapped to *warps* when used in the default Allen framework. For the ISPC Allen development these kernels will map to vector lanes of the target processor.

For the VELO and UT (*Compass*) algorithms to run in a different architecture, all GPU-specific optimizations need to be removed. Various algorithms are implemented to benefit from *shared memory* in the GPU, i.e. the *Compass* algorithm uses this memory to cache hits that indicate the search windows used to find compatible hits. As CPU cache memory cannot be manually managed in the source code as in the GPU, this optimization is removed in all kernels and the host's main memory is used instead. Hits are directly accessed as these are stored and the CPU will cache them. This is represented in Figures 8.5 and 8.6.

shared memory *is shared across a CUDA block.*

Figure 8.5 shows a representative case of the GPU usage of the *shared memory*, which includes the `__syncthreads()` calls to synchronize

```
uniform float s_y_begin[UT_Decoding_ut_max_hits_shared_sector_group];

if (sector_group_number_of_hits > 0) {
  foreach (i = 0 ... sector_group_number_of_hits) {
    s_y_begin[i] = ut_hits.yBegin[sector_group_offset + i];
  }

  find_permutation(...);
}
```

Code 8.6: `uniform` memory usage within Allen ISPC framework

the CUDA *block* for the *shared memory* correctness and avoid race conditions. Memory barriers used in the GPU to synchronize running threads and guarantee correct execution in parallel, are not needed for the Compass algorithm; *gangs* run in parallel, but these use the vector instructions of the processor which are forced to run all the elements at the same time, removing the need for synchronization in these cases. Memory barriers are offered by ISPC, but these are used to avoid data races between threads. Figure 8.6 shows the implemented version of the same piece of functionality. The *shared_memory* array is changed to a *uniform* array that uses the same type and parameters. *uniform* is used to guarantee that the content of the array will be the same for all the vector lanes. ISPC runtime model does not need to explicitly synchronize for `uniform` arrays that are shared by a vector lane, so all synchronization calls are removed. This occurs because all change from one program instance are visible to the other program instances in the same *gang*. Finally, a `foreach()` call is used to process the elements in parallel with the vector lanes.

The ISPC compiler supports C language features with some extra extensions for the vectorization support, but C++ language features are not supported. As CUDA supports C++ and it is used extensively in the different Allen algorithms, the ISPC algorithms need to be adapted to replace the pieces of the source that use C++ features. This includes functions that are implemented with template metaprogramming and class constructors and methods. This is depicted in Listing 8.7 for templates, where the CUDA Allen function is implemented as a template for any `T` type. In the ISPC implementation function calls like this one are implemented for every needed data type, and the `T` type is replaced in multiple functions by `float`, `int` and others.

When re-implementing the algorithms for ISPC, several functions need to be duplicated as different argument types can be used for that function. In this specific case, it is not because the function was implemented as template, but because ISPC can use different combinations of `uniform` and `varying` for arguments and return types. This causes multiple functions to be repeated in functionality, with the same data types used but with different ISPC attributes depending

```
template<typename T>
__host__ __device__ int binary_search_leftmost(
    const T* array,
    const uint array_size,
    const T& value)
{
  int l = 0;
  int r = array_size;
  while (l < r) {
    const int m = (l + r) / 2;
    const auto array_element = array[m];
    if (value > array_element) {
      l = m + 1;
    } else {
      r = m;
    }
  }
  return l;
}
```

Code 8.7: Template usage in GPU Allen framework

```
namespace UT {
  struct HitOffsets {
    ...
    __device__ __host__
    uint sector_group_offset(const uint sector_group) const
    {
      assert(sector_group <= m_number_of_unique_x_sectors);
      return m_ut_hit_offsets[sector_group];
    }
  }
}
```

Code 8.8: CUDA function for *host* and *device*

```
unsigned int32 UT_HitOffsets_sector_group_offset (
  const UT_HitOffsets& hit_offsets,
  const unsigned int32 sector_group)
{
  assert(sector_group <= hit_offsets.m_number_of_unique_x_sectors);
  return hit_offsets.m_ut_hit_offsets[sector_group];
}

uniform unsigned int32 UT_HitOffsets_sector_group_offset (
  uniform const UT_HitOffsets& hit_offsets,
  uniform const unsigned int32 sector_group)
{
  assert(sector_group <= hit_offsets.m_number_of_unique_x_sectors);
  return hit_offsets.m_ut_hit_offsets[sector_group];
}

unsigned int32 UT_HitOffsets_sector_group_offset (
  uniform const UT_HitOffsets& hit_offsets,
  const unsigned int32 sector_group)
{
  assert(sector_group <= hit_offsets.m_number_of_unique_x_sectors);
  return hit_offsets.m_ut_hit_offsets[sector_group];
}
```

Code 8.9: ISPC function with `uniform` and `varying` variants.

if the arguments or return types are part of a vector computation or from a scalar computation. This situation is depicted with an example in Listings 8.8 and 8.9.

Listing 8.8 shows a simple offset calculation function inside a C++ `struct` that can run both the *host* and the *device*. When implementing this function for the ISPC compiler, a function name that indicates the equivalent namespace, struct and original function name is used to distinguish it from other structs and namespaces. The arguments are also changed, as the C language does not allow to include functions inside a `struct`, so an extra argument to return the correct value is included as an argument. The function is replicated three times with different combinations of return values and arguments, as the function is used in the *Compass* algorithm in different contexts where may use an argument running in a vector lane or may need to return a value to be used in a vector lane. In the third implemented version of the function a combination of `varying` and `uniform` arguments is used. Note that if a `uniform` value is needed to be return by a function, all arguments used in the body of the function need to be passed as `uniform` as well.

Gangs run in parallel using the vector lanes. Basic math operations are computed for all the elements, but when control flow structures are introduced inside a *gang*, the flow of the program can diverge leading to different instructions needed to be applied; the same problem applies for GPUs. The ISPC compiler computes the values that do not

```
const uint16_t value = raw_bank.data[hit_index_inside_raw_bank];
const uint32_t nStripsPerHybrid =
    boards.stripsPerHybrids[raw_bank.sourceID];
const uint32_t channelID =
    (value & UT::Decoding::chan_mask) >> UT::Decoding::chan_offset;
const uint32_t index = channelID / nStripsPerHybrid;
```

Code 8.10: CUDA `half` type and composed value.

```
const unsigned int16 value = raw_bank.data[hit_index_inside_raw_bank];
const unsigned int32 nStripsPerHybrid =
    boards.stripsPerHybrids[raw_bank.sourceID] != 0 ?
    boards.stripsPerHybrids[raw_bank.sourceID] : 1;
const unsigned int32 channelID =
    (value & UT_Decoding_chan_mask) >> UT_Decoding_chan_offset;
const unsigned int32 index = channelID / nStripsPerHybrid;
```

Code 8.11: CUDA `half` type and composed value.

meet a condition if at least one element of the *gang* meets it. For most situations it will not cause a problem, other than the performance implications of the divergence. In some cases, because the vector instructions are doing the actual computation of the value that did not meet the condition, arithmetic exceptions can be raised. This situation is depicted in Listings 8.10 and 8.11.

Listing 8.10 shows the CUDA version where to calculate the variable `index`, a division between `channelID` and `nStripsPerHybrid` is computed without raising any exception or error. The ISPC version presented in Listing 8.11 shows the same `index` calculation, but when getting the value `nStripsPerHybrid` it is checked to be different from zero to avoid dividing by it later. As explained, if the gang size is set to be eight, and there are only seven values to compute, the last value will still be computed (but discarded at the end of the computations) and the division by zero will cause an exception. These cases are implemented specifically for the corner cases that may arise during the implementation of the ISPC VELO and UT algorithms. In all cases the arithmetic exception was caused by a division by zero.

As a data-oriented algorithm, types that do not need to use more space than necessary in variables, are in some cases stored as `half` types. For instance the *UT Pre Decode* kernel benefits from this optimization, where the ISPC compiler offers support for `half` types with specific functions to convert from `float_to_half()` or `half_to_float` using the IEEE 16-bit floating-point format. The `half` type does not exist in the ISPC compiler and math operations with it are not possible, needing a conversion to `float` to operate and then converting back to `half`. A composed value is used to store two `half` types in a 16-bit type. For the ISPC compiler the supported types for this operation

```
const half_t yBegin = __float2half(p0Y + numstrips * dp0diY);
const half_t xAtYEq0_local = __float2half(numstrips * dp0diX);
const short* yBegin_p =
  reinterpret_cast<const short*>(&yBegin);
const short* xAtYEq0_local_p =
  reinterpret_cast<const short*>(&xAtYEq0_local);

const short composed_0 = yBegin_p[0];
short composed_1 = xAtYEq0_local_p[0];
const bool sign_0 = composed_0 & 0x8000;
const bool sign_1 = composed_1 & 0x8000;
if (sign_0 ^ sign_1) composed_1 = -composed_1;

const int composed_value =
  ((composed_0 << 16) & 0xFFFF0000) | (composed_1 & 0x0000FFFF);
const float* composed_value_float =
  reinterpret_cast<const float*>(&composed_value);

...

ut_hits.yBegin[hit_index] = composed_value_float[0];
```

Code 8.12: CUDA `half` type and composed value.

are `int16`, where the float 32-bit types can be stored in the regular way. Less precise functions are offered by ISPC as `_fast` functions, but for this implementation the higher precision result is preferred. A type casting issue is encountered when doing this conversion in some cases, and explicit type casts are needed when operating over the `int16` types holding the `half` value. For bit comparisons these need to be cast to `int32` to avoid precision problems. This is highlighted in Listings 8.12 and 8.13.

Listing 8.12 depicts an example from Allen on how to use the `half_t` type supported by CUDA language. The function `__float2half()` is used to store a 16-bit value. The value is then casted as a 16-bit

```
const int16 composed_0 = float_to_half(p0Y + numstrips * dp0diY);
int16 composed_1 = float_to_half(numstrips * dp0diX);
const bool sign_0 = composed_0 & 0x8000;
const bool sign_1 = composed_1 & 0x8000;
if (sign_0 ^ sign_1) composed_1 = -composed_1;

const unsigned int32 composed_value =
  ((((unsigned int32) composed_0 << 16) & 0xFFFF0000) |
    (((unsigned int32) composed_1) & 0x0000FFFF));

...

ut_hits.yBegin[hit_index] = floatbits(composed_value);
```

Code 8.13: ISPC "half" type usage and composed value.

`short` integer type, and a mask is used to get the bits for the sign of the integer type. Then a composed value that stores both 16-bit type is stored in a 32-bit `int` type using masks to store the bits for each type at the beginning and the end of the 32-bit type. In the ISPC implementation, the language provides a special function, as in the CUDA language, to covert from a `float` to a `half` type, but this is directly stored in a 16-bit integer type, a `int16` type in ISPC. The composed value is created in a similar way but ISPC needs explicit casting to 32-bit types when joining both 16-bit types to avoid loosing information in the process, which was not needed in CUDA. Finally, in ISPC the included `floatbits()` functions is used to store this value as a float with a bit-to-bit representation, to not loose information. It is different from doing a simple casting `(float) a`, and would correspond to doing `*((float *)&a)` in the C language.

Other changes include manually implementing some functions found in the C++ standard library that are included with CUDA. When calculating the pointers for the data structures and its offsets, the `sizeof()` function needs to be considered for every case, as `sizeof(uniform struct)` is different from `sizeof(varying struct)`. Other C++ specific constructs like lambda functions need to be replaced by normal C functions to work with ISPC, often making the code harder to read and maintain.

Finally an optimization process is applied to vectorize loops following similar principles to those applied with the GPU kernels. All kernels arguments are flagged as `uniform` for the entry function, but data structures that are then processed in parallel by the vector lanes are flagged as `varying` to allow the compiler to efficiently use the vector units. The performance that can be extracted from the ISPC compiler in the form of vectorization, highly depends on the correct designation of the `uniform` and `varying` variables. These allow the programmer to reason about the data structures, the access patterns, and how loops will be vectorized to extract the performance. While all the input argument pointers of the kernels are flagged to be `uniform`, as it is enforced by the ISPC model, inside the kernel some of the data structures these pointers point to are flagged as `varying` for the `foreach()` loops to compute them in parallel.

For the algorithms implemented in ISPC in this chapter a selection of the kernels was found to improve the throughput performance compared to the CPU non-ISPC versions. This matches what is observed with the GPU algorithms, where some kernels are found to be faster when these run on the CPU as sequential versions compared to the parallel one implemented for the GPU. Not all algorithms and workloads are well suited for parallel processing. For this reason a selection of the kernels that improved the performance compared to the non-ISPC version is used to achieve higher throughput.

8.4 EVALUATION

To benchmark the performance of the ISPC algorithm in the adapted Allen framework, a sequence of algorithms that computes all VELO and UT algorithms is prepared. The algorithms include the decoding of the raw input data for both subdetectors, data preparation algorithms, and the tracking algorithms. All kernels run on the CPU, but as explained in the previous section a selection of ISPC algorithms is used, whereas the others are compiled as standard CPU algorithms. The list of algorithms is depicted in Table 8.2. Algorithms that are naturally sequential, such as the prefix sum, were expected to run faster for its sequential implementation. Other algorithms were found to be faster without the ISPC compiler by swapping the algorithms and measuring the throughput performance of both.

Table 8.2: IPSC and non-ISPC algorithms used for the VELO and UT tracking. The ISPC algorithms are identified by the `spmd_` prefix.

non-ISPC algorithms	ISPC algorithms
init_event_list	
global_event_cut	
	spmd_velo_estimate_input_size
prefix_sum_velo_clusters	
	spmd_velo_masked_clustering
velo_calculate_phi_and_sort	
velo_fill_candidates	
	spmd_velo_search_by_triplet
	spmd_velo_weak_tracks_adder
copy_and_prefix_sum_single_block_velo	
copy_velo_track_hit_number	
prefix_sum_velo_track_hit_number	
consolidate_velo_tracks	
	spmd_ut_calculate_number_of_hits
prefix_sum_ut_hits	
	spmd_ut_pre_decode
	spmd_ut_find_permutation
ut_decode_raw_banks_in_order	
	spmd_ut_search_windows
	spmd_compass_ut
copy_and_prefix_sum_single_block_ut	
copy_ut_track_hit_number	
prefix_sum_ut_track_hit_number	
consolidate_ut_tracks	

To run the benchmarks an Intel Xeon(R) E5-2678W-v4 processor is used in the CPU measurements. The framework is compiled using NVIDIA's `nvcc 10.0` compiler; kernels are compiled both with GCC 9.0 and ISPC 1.12 for each ISPC and non-ISPC version. The achieved throughput performance is depicted in Figure 8.3. In this figure, CPU measurements are shown in blue on the left, where the target vector

instruction set is indicated in the X axis. A scalar version without vectorization is used as baseline to compare the scalability of the other implementations. The vectorized benchmarks are run using the instruction sets SSE4, AVX-1 and AVX-2, where the AVX-512 is not tested as the processor does not support it. The fastest configuration of mask size and gang size is used for each instruction set that is tested here. Three GPUs are included in the scalability measurements to compare two SIMD (or SIMT) processors that run the same algorithms. These are depicted in three different shades of green to indicate that these are different processors each.

The same shade of blue is used for the CPU as the processor does not change

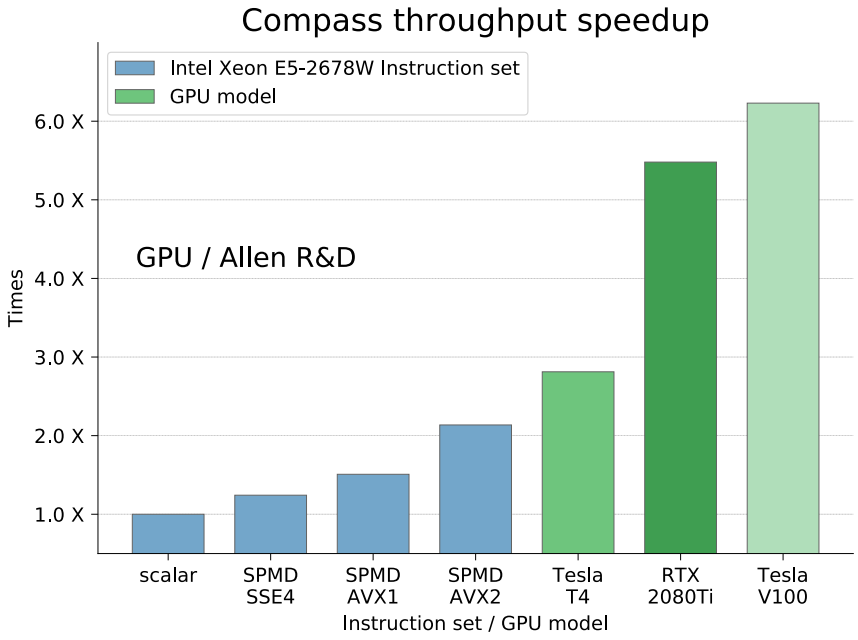


Figure 8.3: Performance comparison

The CPU performance achieves more than 2× speedup with the AVX-2 target instruction set. Some performance losses are expected, compared to the ideal speedup, due to tracks that are filtered early in the algorithms. As the nature of these algorithms is to filter events, some algorithms will discard tracks early in the computation and vector lanes will be unused for the rest of the loop computation of that track. Other losses are expected from branching, where the vector units are impacted negatively, as with GPU processors. Kernels like the candidates finding one present a reduced *gang* usage due to multiple conditional clauses that are needed by the algorithm and vector lanes are wasted. Finally, as not all kernels are vectorized using the ISPC compiler, a significant fraction of the running kernels will not experience any throughput improvement. This limits the maximum speedup that can be achieved with the given configuration of kernels and the given ratio between ISPC and non-ISPC kernels.

For the same benchmark, all the used GPU processors achieve higher throughput compared to the highest throughput CPU AVX-2 throughput. Lower performance GPUs like the Tesla T4 offer a core count of 2560, whereas the E5-2678W-v4 offers 12 cores and 24 threads for computations. Even with AVX-2 256 bit wide vector units that allow to load 8 floats at a time, the number of elements processed in parallel is far from the GPU one. CPUs offer a better instruction level parallelism and higher clock frequencies compared to the GPUs. Higher end GPUs like the Tesla V100 achieve over 6× higher throughput compared to the scalar CPU version of the algorithms.

8.5 SUMMARY

The ISPC compiler offers an alternative to the intermediate libraries, intrinsics or in-line assembly to make an efficient usage of the vector processing units available in CPUs. Programming for its language, a C language variant with extensions, resembles the approach followed by languages like CUDA or OpenCL, where similar principles to achieve good vector units usage apply.

In this chapter all kernels for the VELO and UT subdetectors from the Allen framework are implemented for the ISPC compiler, including compiled versions without using the ISPC compiler and no explicit vectorization. The SPMD model allows to write the algorithm as if these are sequential, and then reason about the parallel loops and regions of code that can be indicated for the compiler to vectorize them, explicitly. This model benefits from data-parallel algorithms with coalesced memory, and data-aware access patterns and data layouts in an efficient manner both from the performance and programmer efficiency point of view. The effort required to write the algorithms resembles those to make GPU kernels, but the limitations of the C language enforce practices that may be more error prone such as function duplication or the absence of member functions. A simpler set of language features is exposed compared to C++, as only the C language features and the needed extensions for explicit vectorization are used. The main advantage is the better readability and maintainability compared to writing intrinsics or intermediate libraries, while offering comparable performance to these solutions. The compiler allows to easily change between gang sizes and mask sizes to adapt to the specifics of the algorithm, which resembles to the kernel launch configuration of CUDA, where different block sizes and groups of threads offer differences in performance. Furthermore, the ISPC compiler integrates performance warnings that are displayed to the programmer to warn about patterns that may degrade performance when using the vector units.

The chain of algorithms tested in this chapter achieve more than a factor 2 compared to the non-vectorized version. The increased throughput is the result of the parallelization of loops inside the kernels in an analogous way as in the GPU version of Allen. The available target instructions sets exhibit different levels of performance improvements, where the widest vector width setting offers the biggest improvement. The same algorithms are compared for both their CPU/ISPC and GPU implementations, showing the performance improvements for the CPU, but highlighting the performance benefit of the GPUs for massively parallel workloads. HEP workloads expose a high degree of parallelism due to the multiple particles and hits that can be processed in parallel. For workloads that need to process high data rates in real-time, vectorization technologies offer an important advantage to better use the available CPU chip.

Part IV

CONCLUSIONS

9

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the presented work of this thesis and presents an analysis and final conclusion. It shows the publications that were produced during the course of the PhD and lead to this thesis. Future directions are discussed as a continuation from this work.

9.1 SUMMARY

This thesis covered new advances and developments in the computing high-energy physics research field. The LHCb experiment at CERN, Switzerland, is undergoing an upgrade for its next data-taking period in 2021. This upgrade will pose a tremendous challenges in terms of computing; a data rate increase of 40× compared to the previous data-taking period forces both the hardware and software to be upgraded, optimized, and improved to cope with a target data throughput of 40 Tb per second. The upgrade change that has the biggest impact on the computing capabilities is the removal of the previously used *hardware trigger* to replace it with a *full software trigger*. This software trigger needs to process the full data rate in real-time to select and discard useful proton-proton collisions. To achieve this goal, parallel hardware architectures and efficient software for them play a central role in accelerating LHCb's computing capabilities. Improvements in hardware and software were taken into account during the planning for the upgrade. However the expectations were not met in terms of computing cost needs and software optimizations, and new approaches were considered including new algorithms, code base modernization and alternative hardware architectures.

This thesis work covers improvements and optimizations of algorithms and software for parallel architectures in the context of LHCb's experiment upgrade. It is structured in three main parts:

- *Part 1. The LHCb experiment*: This part covers the main pieces of the experiment involved in this thesis. The detector and its tracking subdetectors are covered, including the *Data acquisition system* and the *High Level Trigger*. These are the elements that provide the measurements for the particle collisions are run the software to filter them in real-time. The computing infrastructure used at LHCb is introduced, highlighting the software frameworks that run the algorithms to reconstruct particle trajectories: *Gaudi* and *Allen*. Particle track reconstruction and its

particularities at LHCb are introduced to describe the methods, algorithms and metrics used to perform it.

- *Part 2. Parallel computing*: This thesis is focused around parallel computing, including the parallelism found in hardware processors and how software can leverage this parallelism to improve the usage of this chips, and deliver software that runs faster for the given algorithms. Main aspects from this processors that influence software performance are covered, in particular two main hardware accelerators are covered for this thesis: the Intel Xeon Phi and GPUs.

- *Part 3. Particle tracking in high-energy physics*: This part covers the improvements and optimizations performed for various algorithms, using different techniques, frameworks and parallel architectures. Three main contributions are covered here:

  - The studies on the parallelization of the Kalman filter inside each track processing, and with different parallelization schemes. In collaboration with Intel Corporation through CERN Openlab, Intel hardware and software were studied to be candidate technologies for the upgrade. Intel Xeon Phi KNL architecture was used as a target, as it offers a many-core x86 architecture well suited for massively parallel tasks were vector units, high core count and 4-way hyperthreading could provide high throughput. Three levels of parallelization are presented, from finer to coarser grain sized, and different pipeline configurations.

    As LHCb algorithms are expected to have a long life-cycle with years in between data-taking periods, good performance and maintainability are key aspects of LHCb software stack. An implementation that uses generic parallel patterns for the Kalman filter is presented in this thesis, where the patterns offer an abstraction that is optimized for the Intel Xeon Phi to be architecture-aware, and improves readability while minimizing the possibility of incurring in errors. A comparable performance is achieved compared to the baseline while hiding the parallelization details.

  - Different hardware architectures are considered for the upgrade, for instance accelerators such as GPUs present solid alternatives to the x86 architecture. During this thesis contributions are made to LHCb's Allen framework, which offers an alternative software framework that is compact and optimized for parallel data-oriented algorithms. *Compass* is shown as a decoding and tracking algorithm for the Allen framework. The algorithm is designed and implemented to be efficient for massively parallel architectures, in particular for GPUs. A throughput analysis is presented

with results from various scientific and consumer GPUs, achieving a throughput of more than 7× compared to the baseline CPU version.

- Vectorization opportunities are studied within the Allen framework, extending it to support SPMD kernels and algorithms that can be used in an heterogeneous computing environment. Memory efficient and data-oriented principles are extended from the GPUs to CPUs to benefit from vectorization. The Intel Implicit SPMD Program Compiler (ISPC) is used to implement VELO and UT algorithms. By using the language provided by ISPC loops can be explicitly vectorized, and a performance improvement over 2× compared to the scalar CPU version is achieved when mixing vectorized and non-vectorized kernels that are processed with the CPU. A performance study is presented with a comparison against the GPU implementations of Allen.

The hypothesis presented in Section 1.1 was successfully achieved: *The developed algorithms, chosen hardware and applied improvements and optimizations for the LHCb experiment, delivered over the target 40 Tb/s data throughput before the next data-taking period started.* The contributions of this thesis were crucial to reach this goal, in particular to successfully deliver a high-performance GPU framework, Allen, and its algorithms.

The objectives presented in Section 1.2 were met as presented here:

- **O1:** *Various multi-threading possibilities to parallelize the processing of particle tracks have been explored*. During the optimization of the Kalman filter algorithm, different parallelization schemes were analized and discussed. This resulted in three different parallel implementations with different tasks grain sizes, where the goal was to improve the usage of the SMT capabilities of the processor. An implementation of this Kalman filter with generic parallel patterns was delivered, improving the maintainability and readability of the source code while achieving a comparable performance.

- **O2:** *Various implementations of the Kalman filter have been optimized for the Intel Xeon Phi KNL*. The Kalman filter being one of the main time contributors to LHCb HLT computing, was selected as a target to improve with the selected processor. As part of the High Throughput Computing Collaboration with Intel, various Intel technologies were selected to improve computing performance for the upgrade.

- **O3:** *Vectorization opportunities have been explored with different tracking algorithms of LHCb*. As part of the development of the Allen framework and the algorithms for it, vectorized algorithms

have been developed to leverage the data-oriented design of the Allen framework. A SPMD model approach was used to improve the performance and usage of the vector units available in the CPU, with an improvement of over 2× in throughput.

- **O4:** *New algorithms have been implemented for GPU architectures and their performance have been studied*. The development of a GPU framework, Allen, was designed from the beginning to efficiently use GPU accelerators with the goal of delivering a high-throughput compact solution. The Compass algorithm was designed and developed as a data-oriented algorithm that is optimized for GPUs. It delivered an improved performance over 7× the throughput compared to the baseline solution with CPUs.

The use of different high-performance technologies such as higher parallelism, multithreading, vectorization and accelerators in the context of high-energy physics, is a necessary evolution that bring performance and efficiency improvements. To process data rates of up to 40Tb/s in the LHCb experiment, in real-time, an heterogeneous solution provides the benefits of different architectures for different workloads and use cases. The Allen GPU framework project and its algorithms started during the development of this thesis, under the time constraint of the next data-taking period of LHCb, *Run3*, and was successfully designed and implemented by a small team of developers including myself. This project was selected by the LHCb collaboration to be the framework to process the HLT1 for *Run3*.

LHCb solution will be a high-performance heterogeneous computing solution. The Allen framework proved to be compact, efficient and flexible; it is able to compile its algorithms for different architectures, and different compilers and algorithms can be used. This is demonstrated in this thesis with the inclusion of CPU vectorized algorithms with the ISPC compiler. As different parallel architectures and parallelization models are used in scientific computing fields like high-energy physics, abstractions are needed to target different processors or parallelization libraries. Approaches like generic parallel patterns help to reduce the effort needed to implement current and future algorithms that are efficient for parallel architectures. In the context of the software used in the Large Hadron Collider experiments, its software will be maintained and used during several years or decades, while multiple different people reading, using and changing the source code. The importance of maintainable and high-performance solutions is bigger for this kind of projects. This is shown in this thesis with the parallelization of the Kalman filter and the use of generic parallel patterns, providing a solution that is comparable in performance but uses abstractions for the parallelization. This solution is optimized for a particular processor, the Intel Xeon Phi, as part of a joint effort with Intel and CERN Openlab.

LHCb computing for fine grained tasks remains a challenge as the data structures and sizes of the events recorded by the detector are not big in size. The computing challenge is posed by the high data rate generated by the 40 million collisions every second, where multiple tracks per collision need to be reconstructed, in real-time. In future upgrades, experiments like LHCb will be able to record more information and with a higher detail, which will generate bigger events that will open further possibilities to parallelize.

This thesis contributed in prominent fields of high-performance computing to help the LHCb experiment achieve its computing goals. It has shown how accelerators, vectorization, multithreading and generic parallel patterns can be applied to high-energy physics algorithms and scenarios with success.

9.2 DISSEMINATION

A list of publications and research contributions produced during the work of this thesis is presented here:

Publications:

- Plácido Fernández et al. «Parallelizing and Optimizing LHCb-Kalman for Intel Xeon Phi KNL Processors.» In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2018, pp. 741–750. DOI: 10.1109/PDP2018.2018.00121
- Placido Fernandez Declara et al. «A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures.» In: *IEEE Access* 7 (2019), pp. 91612–91626. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927261
- Placido Fernandez Declara and J. Daniel Garcia. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2020. Accepted for publication at CHEP 2019 Proceedings, Adelaide, Australia
- Roel Aaij et al. «Allen: A High-Level Trigger on GPUs for LHCb.» In: *Computing and Software for Big Science* 4.7 (2020). DOI: 10.1007/s41781-020-00039-7

Posters:

- Placido Fernandez Declara. «CompassUT: study of a GPU track reconstruction for LHCb upgrades.» 2019. URL: https://cds.cern.ch/record/2665033. Poster presented at Winter LHCC sessions, CERN, Switzerland
- Placido Fernandez Declara. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2019. URL: https://cds.cern.ch/record/2699802. Poster presented at CHEP 2019, Adelaide, Australia

Talks at international conferences:

- Placido Fernandez Declara. *Fast Kalman Filtering: new approaches for the LHCb upgrade*. Tech. rep. 2018. URL: http://cds.cern.ch/record/2631784

Technical reports:

- LHCb Collaboration. *LHCb Upgrade GPU High Level Trigger Technical Design Report*. Tech. rep. CERN-LHCC-2020-006. LHCB-TDR-021. Geneva: CERN, 2020. URL: https://cds.cern.ch/record/2717938

I am also a co-author in a number of LHCb related papers. The full list can be found at:

- Placido Fernandez Declara. *Google Scholar profile*. https://scholar.google.com/citations?user=Ygkq_7YAAAAJ. 2020

9.3 FUNDING

- CERN Openlab[1], a public-private partnership between CERN and ICT companies. In collaboration with Intel Corporation through the High-Throughput Computing Collaboration (HTCC)[2].
- CERN LHCb experiment[3].
- Madrid Regional Government, CABAHLA-CM (ConvergenciA Big dAta-Hpc: de Los sensores a las Aplicaciones) grant number S2018/TCS-4423.
- University Carlos III of Madrid Strategic Action on Programming Models for Software Improvement (ref. 2013/00196/002).
- Spanish MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD through project grant TIN2016-79637-P TOWARDS UNIFICATION OF HPC AND BIG DATA PARADIGMS.
- EU Project ICT 644235 "REPHRASE: REfactoring Parallel Heterogeneous Resource-Aware Applications".

9.4 FUTURE DIRECTIONS

The work presented in this thesis can be extended in its different contributions, and parts of this thesis are adopted by the LHCb experiment to be used in the coming years. In particular the developments

1 https://home.cern/science/computing/cern-openlab
2 https://openlab-archive-phases-iv-v.web.cern.ch/technical-area/data-acquisition-online
3 http://lhcb-public.web.cern.ch/

in the Allen framework and GPU algorithms will be extended for its use in the next data-taking period from 2021. The Allen framework can implement further optimizations to achieve a higher performance which will allow to integrate more detailed algorithms and data from other subdetectors to help make better filtering decisions. The SPMD implemented algorithms with Allen are the base to integrate heterogeneous capabilities further to the Allen framework. Compilation for other GPU vendors, different CPUs or other accelerators can be achieved with solutions like SYCL or Alpaka, where all architectures would benefit from the data-parallel design of the framework to make an efficient use of the processors.

The author of this thesis has continued working at CERN, Switzerland, in the development of future experiments. In particular the author is working for the Future Circular Collider (FCC) and the Compact Linear Collider (CLIC). This future experiments are developing the tools and frameworks needed for the future challenges in particle physics. The experience and research of the work of this thesis in tracking algorithms and framework development will be applied by the author in these developments.

BIBLIOGRAPHY

[1] R Aaij, S Benson, M De Cian, A Dziurda, C Fitzpatrick, E Govorkova, O Lupton, R Matev, S Neubert, A Pearce, et al. «A comprehensive real-time analysis model at the LHCb experiment.» In: *Journal of Instrumentation* 14.04 (2019), P04006.

[2] Roel Aaij, Johannes Albrecht, M Belous, P Billoir, T Boettcher, A Brea Rodríguez, D vom Bruch, DH Pérez, A Casais Vidal, DC Craik, et al. «Allen: A High-Level Trigger on GPUs for LHCb.» In: *Computing and Software for Big Science* 4.7 (2020). DOI: 10.1007/s41781-020-00039-7.

[3] M Abolins, R Abreu, R Achenbach, M Aharrouche, G Aielli, A Al-Shabibi, I Aleksandrov, E Alexandrov, BM Allbrooke, A Aloisio, et al. «The ATLAS Data Acquisition and High Level Trigger system.» In: *Journal of Instrumentation* 11 (2016).

[4] A Augusto Alves Jr, LM Andrade Filho, AF Barbosa, I Bediaga, G Cernicchiaro, G Guerrer, HP Lima Jr, AA Machado, J Magnin, F Marujo, et al. «The LHCb detector at the LHC.» In: *Journal of instrumentation* 3.08 (2008), S08005.

[5] S Amato, S Topp-Jorgensen, G Carboni, R Schwierz, V Zerkin, S Haider, D George, A Petrolini, Ian C McArthur, L Paoluzi, et al. *LHCb Technical Proposal: A large hadron collider beauty experiment for precision measurements of CP violation and rare decays*. Tech. rep. 1998.

[6] Gene 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*. ACM. 1967, pp. 483–485.

[7] Carl D Anderson. «The positive electron.» In: *Physical Review* 43.6 (1933), p. 491.

[8] R. E. Andreassen, W. M. de Silva, B. T. Meadows, M. D. Sokoloff, and K. A. Tomko. «Implementation of a Thread-Parallel, GPU-Friendly Function Evaluation Library.» In: *IEEE Access* 2 (2014), pp. 160–176. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2014.2306895.

[9] Alexey Pavlovich Badalov et al. «Coprocessor integration for real-time event processing in particle physics detectors.» PhD thesis. Universitat Ramon Llull, 2016.

[10] GAUDI& Barrand, I Belyaev, P Binko, M Cattaneo, R Chytracek, G Corti, M Frank, G Gracia, J Harvey, Eric Van Herwijnen, et al. «GAUDI—A software architecture and framework for building HEP data processing applications.» In: *Computer Physics Communications* 140.1-2 (2001), pp. 45–55.

[11] F. Baruffa, L. Iapichino, N. J. Hammer, and V. Karakasis. «Performance Optimisation of Smoothed Particle Hydrodynamics Algorithms for Multi/Many-Core Architectures.» In: *2017 International Conference on High Performance Computing Simulation (HPCS)*. July 2017, pp. 381–388. DOI: 10.1109/HPCS.2017.64.

[12] Pierre L Bastien and Lawrence A Dunn. «Global transformations in pattern recognition of bubble chamber photographs.» In: *IEEE Transactions on Computers* 100.9 (1971), pp. 995–1001.

[13] G Bauer, B Beccati, U Behrens, K Biery, O Bouffet, J Branson, S Bukowiec, E Cano, H Cheung, M Ciganek, et al. «The data-acquisition system of the CMS experiment at the LHC.» In: *Journal of Physics: Conference Series*. Vol. 331. 2. IOP Publishing. 2011, p. 022021.

[14] I Belyaev, Ph Charpentier, S Easo, P Mato, J Palacios, W Pokorski, F Ranjard, and J Van Tilburg. «Simulation application for the LHCb experiment.» In: *arXiv preprint physics/0306035* (2003).

[15] Pierre Billoir. «Progressive track recognition with a Kalman-like fitting procedure.» In: *Computer Physics Communications* 57.1-3 (1989), pp. 390–394.

[16] Ian Bird. «Computing for the large hadron collider.» In: *Annual Review of Nuclear and Particle Science* 61 (2011), pp. 99–118.

[17] Javier Garcia Blas, Monica Abella, Florin Isaila, Jesus Carretero, and Manuel Desco. «Surfing the optimization space of a multiple-GPU parallel implementation of a X-ray tomography reconstruction algorithm.» In: *Journal of Systems and Software* 95 (2014), pp. 166–175.

[18] Rudolf K Bock, H Grote, and D Notz. *Data analysis techniques for high-energy physics*. Vol. 11. Cambridge University Press, 2000.

[19] E Bos and E Rodrigues. *The LHCb track extrapolator tools*. Tech. rep. 2007.

[20] E Bowen and B Storaci. *VeloUT tracking for the LHCb Upgrade*. Tech. rep. LHCb-PUB-2013-023. CERN-LHCb-PUB-2013-023. LHCb-INT-2013-056. Geneva: CERN, Apr. 2014. URL: http://cds.cern.ch/record/1635665.

[21] Espen Eie Bowen, Barbara Storaci, and Marco Tresch. *VeloTT tracking for LHCb Run II*. Tech. rep. LHCb-PUB-2015-024. CERN-LHCb-PUB-2015-024. LHCb-INT-2014-040. Geneva: CERN, Apr. 2016. URL: http://cds.cern.ch/record/2105078.

[22] Espen Eie Bowen, Ulrich Straumann, Nicola Serra, Olaf Steinkamp, and Barbara Storaci. «Upstream Tracking and the Decay $B^0 \to K^+\pi^-\mu^+\mu^-$ at the LHCb Experiment.» Presented 26 Jan 2017. PhD thesis. University of Zurich, Oct. 2016. URL: http://cds.cern.ch/record/2261918.

[23] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. «hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications.» Anglais. In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Ed. by IEEE. Pisa Italie, Feb. 2010. URL: http://hal.inria.fr/inria-00429889/en/.

[24] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. «Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns.» In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. CGO '16. Barcelona, Spain: ACM, 2016, pp. 194–205. ISBN: 978-1-4503-3778-6. DOI: 10.1145/2854038.2854042. URL: http://doi.acm.org/10.1145/2854038.2854042.

[25] Dorothea vom Bruch. «Online Data Reduction using Track and Vertex Reconstruction on GPUs for the Mu3e Experiment.» In: *EPJ Web of Conferences*. Vol. 150. EDP Sciences. 2017, p. 00013.

[26] Nicola Cadenelli, Zoran Jakšić, Jordà Polo, and David Carrera. «Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads.» In: *Future Generation Computer Systems* 94 (2019), pp. 148–159.

[27] Daniel Hugo Cámpora Pérez. «A high-throughput Kalman filter for modern SIMD architectures.» In: *Euro-Par Workshops*. 2017, in press.

[28] Daniel Hugo Campora Perez. «LHCb Kalman filter cross architecture studies.» In: *J. Phys.: Conf. Ser.* Vol. 898. LHCb-PROC-2017-041. 2017, p. 032052.

[29] Daniel Hugo Cámpora Pérez, Niko Neufeld, and Agustín 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. DOI: 10.1109/IPDPSW.2019.00118. URL: https://doi.org/10.1109/IPDPSW.2019.00118.

[30] Laurent Canetti, Marco Drewes, and Mikhail Shaposhnikov. «Matter and antimatter in the universe.» In: *New Journal of Physics* 14.9 (Sept. 2012), p. 095012. DOI: 10.1088/1367-2630/

14/9/095012. URL: https://doi.org/10.1088%2F1367-2630%2F14%2F9%2F095012.

[31] Mark Cattaneo. *GAUDI-The software architecture and framework for building LHCb data processing applications*. Tech. rep. 2000.

[32] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin A Kwiat, Charles Alexandre Kamhoua, and Chonggang Wang. «GPU-accelerated high-throughput online stream data processing.» In: *IEEE Transactions on Big Data* 4.2 (2018), pp. 191–202.

[33] Mourad Chouaki. *Analysis of Low-Momentum Upstream Tracking for the LHCb upgrade*. Tech. rep. École Polytechnique Fédérale de Lausanne, June 2016. URL: https://lphe.epfl.ch/oschneid/cours/2015-2016/rapports_TP4/TP4b_Mourad_Chouaki_UpstreamTracking_jun2016.pdf.

[34] Mike Clark. «A new× 86 core architecture for the next generation of computing.» In: *Hot Chips Symposium*. 2016, pp. 1–19.

[35] M Clemencic, B Hegner, and C Leggett. «Gaudi evolution for future challenges.» In: *J. Phys. : Conf. Ser.* 898.4 (2017), 042044. 3 p. DOI: 10.1088/1742-6596/898/4/042044. URL: http://cds.cern.ch/record/2297285.

[36] M Clemencic, J Palacios, and N Gilardi. *LHCb Conditions Database*. Tech. rep. LHCb-PROC-2006-004. CERN-LHCb-PROC-2006-004. Geneva: CERN, Feb. 2006. URL: https://cds.cern.ch/record/1442972.

[37] LHCb Collaboration. *LHCb PID Upgrade Technical Design Report*. Tech. rep. CERN-LHCC-2013-022. LHCB-TDR-014. 2013. URL: https://cds.cern.ch/record/1624074.

[38] LHCb Collaboration. *LHCb VELO Upgrade Technical Design Report*. Tech. rep. CERN-LHCC-2013-021. LHCB-TDR-013. Nov. 2013. URL: http://cds.cern.ch/record/1624070.

[39] LHCb Collaboration. «LHCb detector performance.» In: *International Journal of Modern Physics A* 30.07 (2015), p. 1530022.

[40] LHCb Collaboration. *LHCb Upgrade GPU High Level Trigger Technical Design Report*. Tech. rep. CERN-LHCC-2020-006. LHCB-TDR-021. Geneva: CERN, 2020. URL: https://cds.cern.ch/record/2717938.

[41] T Colombo, A Amihalachioaei, K Arnaud, F Alessio, L Brarda, JP Cachemiche, D Cámpora, S Cap, L Cardoso, F Cindolo, 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*. Vol. 1085. 3. IOP Publishing. 2018, p. 032041.

[42] Tommaso Colombo, Paolo Durante, Domenico Galli, Umberto Marconi, Niko Neufeld, Flavio Pisani, Rainer Schwemmer, Sébastien Valat, and Balazs Voneki. «The LHCb DAQ upgrade for LHC Run3.» In: *IEEE Transactions on Nuclear Science* (2019).

[43] Gloria Corti, Marco Cattaneo, Philippe Charpentier, Markus Frank, Patrick Koppenburg, Pere Mato, Florence Ranjard, Stefan Roiser, Ivan Belyaev, and Guy Barrand. «Software for the LHCb experiment.» In: *IEEE transactions on nuclear science* 53.3 (2006), pp. 1323–1328.

[44] Miles D Cranmer, Benjamin R Barsdell, Danny C Price, Jayce Dowell, Hugh Garsden, Veronica Dike, Tarraneh Eftekhari, Alexander M Hegedus, Joseph Malins, Kenneth S Obenberger, et al. «Bifrost: A Python/C++ Framework for High-Throughput Stream Processing in Astronomy.» In: *Journal of Astronomical Instrumentation* 6.04 (2017), p. 1750007.

[45] D. H. Cámpora Pérez, N. Neufeld, A. Riscos Nuñez. «A fast local algorithm for track reconstruction on parallel architectures.» IPDPS workshops, to appear. 2019.

[46] Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati. «Data stream processing via code annotations.» In: *The Journal of Supercomputing* (June 2016). ISSN: 1573-0484. DOI: 10.1007/s11227-016-1793-9. URL: https://doi.org/10.1007/s11227-016-1793-9.

[47] Michel De Cian et al. *Status of HLT1 sequence and path towards 30 MHz*. Tech. rep. LHCb-PUB-2018-003. CERN-LHCb-PUB-2018-003. Geneva: CERN, Mar. 2018. URL: http://cds.cern.ch/record/2309972.

[48] Bruce Denby. «The use of neural networks in high-energy physics.» In: *Neural Computation* 5.4 (1993), pp. 505–549.

[49] Youcef Djenouri, Ahcene Bendjoudi, Zineb Habbas, Malika Mehdi, and Djamel Djenouri. «Reducing thread divergence in GPU-based bees swarm optimization applied to association rule mining.» In: *Concurrency and Computation: Practice and Experience* 29.9 (2017), e3836.

[50] Richard O Duda and Peter E Hart. *Use of the Hough transformation to detect lines and curves in pictures*. Tech. rep. Sri International Menlo Park Ca Artificial Intelligence Center, 1971.

[51] Richard O Duda and Peter E Hart. «Use of the Hough transformation to detect lines and curves in pictures.» In: *Communications of the ACM* 15.1 (1972), pp. 11–15.

[52] Luke Durant, Olivier Giroux, Mark Harris, and Nick Stam. *Inside Volta: The World's Most Advanced Data Center GPU*. https://devblogs.nvidia.com/inside-volta/. Published: 2017-05-10.

[53] Rutger M Van der Eijk. «Track reconstruction in the LHCb experiment.» PhD thesis. NIKHEF, Amsterdam, 2002.

[54] Pierre Estérie, Joel Falcou, Mathias Gaunard, and Jean-Thierry Lapresté. «Boost. simd: generic programming for portable simdization.» In: *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*. ACM. 2014, pp. 1–8.

[55] Placido Fernandez Declara. *Fast Kalman Filtering: new approaches for the LHCb upgrade*. Tech. rep. 2018. URL: http://cds.cern.ch/record/2631784.

[56] Placido Fernandez Declara. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2019. URL: https://cds.cern.ch/record/2699802. Poster presented at CHEP 2019, Adelaide, Australia.

[57] Placido Fernandez Declara. «CompassUT: study of a GPU track reconstruction for LHCb upgrades.» 2019. URL: https://cds.cern.ch/record/2665033. Poster presented at Winter LHCC sessions, CERN, Switzerland.

[58] Placido Fernandez Declara. *Google Scholar profile*. https://scholar.google.com/citations?user=Ygkq_7YAAAAJ. 2020.

[59] Placido Fernandez Declara and J. Daniel Garcia. «Compass SPMD: a SPMD vectorized tracking algorithm.» 2020. Accepted for publication at CHEP 2019 Proceedings, Adelaide, Australia.

[60] Placido Fernandez Declara, Daniel Hugo Perez Campora, Javier Garcia-Blas, Dorothea Vom Bruch, J Daniel Garcia, and Niko Neufeld. «A parallel-computing algorithm for high-energy physics particle tracking and decoding using GPU architectures.» In: *IEEE Access* 7 (2019), pp. 91612–91626. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927261.

[61] Plácido Fernández, David del Rio Astorga, Manuel F Dolz, Javier Fernández, Omar Awile, and J Daniel García. «Parallelizing and Optimizing LHCb-Kalman for Intel Xeon Phi KNL Processors.» In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2018, pp. 741–750. DOI: 10.1109/PDP2018.2018.00121.

[62] Michael Flynn. «Flynn's Taxonomy.» In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 689–697. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2. URL: https://doi.org/10.1007/978-0-387-09766-4_2.

[63] Agner Fog. «C++ vector class library.» In: *URL: http://www.agner.org/optimize/vectorclass.pdf* (2013).

[64] Daniel Funke, Thomas Hauth, Vincenzo Innocente, Günter Quast, P Sanders, and D Schieferdecker. «Parallel track reconstruction in CMS using the cellular automaton approach.» In: *Journal of Physics: Conference Series*. Vol. 513. IOP Publishing. 2014, p. 052010.

[65] Mary K Gaillard, Paul D Grannis, and Frank J Sciulli. «The standard model of particle physics.» In: *Reviews of Modern Physics* 71.2 (1999), S96.

[66] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[67] *Gaudi source code*. https://gitlab.cern.ch/gaudi/Gaudi. Accessed: 2020-06-07.

[68] Donald A Glaser. «Some effects of ionizing radiation on the formation of bubbles in liquids.» In: *Physical Review* 87.4 (1952), p. 665.

[69] Matt Godbolt, Rubén Rincón, Patrick Quist, Austin Morton, Jared Wyles, Chedy Najjar, Simon Brand, and Filipe Cabecinhas. *Compiler explorer*. 2017.

[70] Brice Goglin. «Exposing the Locality of Heterogeneous Memory Architectures to HPC Applications.» In: *1st ACM International Symposium on Memory Systems (MEMSYS16)*. Washington, DC, United States: ACM, Oct. 2016. DOI: 10.1145/2989081.2989115. URL: https://hal.inria.fr/hal-01330194.

[71] Paul D Groves. «Principles of GNSS, inertial, and multisensor integrated navigation systems, [Book review].» In: *IEEE Aerospace and Electronic Systems Magazine* 30.2 (2015), pp. 26–27.

[72] John L Gustafson. «Reevaluating Amdahl's law.» In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[73] VKAVV Halyo, Patrick LeGresley, P Lujan, V Karpusenko, and A Vladimirov. «First evaluation of the CPU, GPGPU and MIC architectures for real time particle tracking based on Hough transform at the LHC.» In: *Journal of Instrumentation* 9.04 (2014), P04005.

[74] Wookhyun Han, Hoon Sung Chwa, Hwidong Bae, Hyosu Kim, and Insik Shin. «GPU-SAM: Leveraging multi-GPU split-and-merge execution for system-wide real-time support.» In: *Journal of Systems and Software* 117 (2016), pp. 1–14.

[75] Mark Harris, Shubhabrata Sengupta, and John D Owens. «Parallel prefix sum (scan) with CUDA.» In: *GPU gems* 3.39 (2007), pp. 851–876.

[76] B Hegner, P Mato, and D Piparo. «Evolving LHC data processing frameworks for efficient exploitation of new CPU architectures.» In: *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2012 IEEE*. IEEE. 2012, p. 513.

[77] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[78] Bernhard Hofmann-Wellenhof, Herbert Lichtenegger, and Elmar Wasle. *GNSS–global navigation satellite systems: GPS, GLONASS, Galileo, and more*. Springer Science & Business Media, 2007.

[79] Cheng-Liang Hsieh, Lucas Vespa, and Ning Weng. «A high-throughput DPI engine on GPU via algorithm/implementation co-optimization.» In: *Journal of Parallel and Distributed Computing* 88 (2016), pp. 46–56.

[80] Xinyi Hu and Yaqun Zhao. «Gridding Algorithm in ARL Based on GPU Parallelization.» In: *Proceedings of the 6th ACM/ACIS International Conference on Applied Computing and Information Technology*. ACM. 2018, pp. 13–18.

[81] Mikhail Hushchyn, A Baranov, S Roiser, K Arzymatov, and A Ustyuzhanin. «IOP: The LHCb Grid simulation: Proof of concept.» In: *J. Phys.: Conf. Ser.* Vol. 898. 2017, p. 052020.

[82] *Intel TBB Documentation - Controlling Chunking*. 2016. URL: https://software.intel.com/en-us/node/506060 (visited on 10/09/2010).

[83] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[84] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. «Dissecting the nvidia volta gpu architecture via microbenchmarking.» In: *arXiv preprint arXiv:1804.06826* (2018).

[85] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. «High-Level Programming Abstractions for Distributed Graph Processing.» In: *IEEE Transactions on Knowledge and Data Engineering* (2017).

[86] R. E. Kalman. «A New Approach to Linear Filtering and Prediction Problems.» In: *Journal of Basic Engineering* 82.1 (Mar. 1960), pp. 35–45. ISSN: 0098-2202. DOI: 10.1115/1.3662552. URL: http://dx.doi.org/10.1115/1.3662552 (visited on 09/05/2017).

[87] Donald E. Knuth. «Computer Programming as an Art.» In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

[88] LHCb Collaboration. *LHCb Tracker Upgrade Technical Design Report*. Tech. rep. CERN-LHCC-2014-001. LHCB-TDR-015. CERN, Feb. 2014. URL: http://cds.cern.ch/record/1647400.

[89] LHCb Collaboration. «Measurement of the track reconstruction efficiency at LHCb.» In: *Journal of Instrumentation* 10.02 (2015), P02007.

[90] CERN (Meyrin) LHCb Collaboration. *Computing Model of the Upgrade LHCb experiment*. Tech. rep. CERN-LHCC-2018-014. LHCB-TDR-018. Geneva: CERN, May 2018. URL: http://cds.cern.ch/record/2319756.

[91] *LHCb Trigger and Online Upgrade Technical Design Report*. Tech. rep. CERN-LHCC-2014-016. LHCB-TDR-016. May 2014. URL: https://cds.cern.ch/record/1701361.

[92] Massimo Lamanna. «The LHC computing grid project at CERN.» In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 534.1-2 (2004), pp. 1–6.

[93] Huanxin Lin, Cho-Li Wang, and Hongyuan Liu. «On-GPU thread-data remapping for branch divergence reduction.» In: *ACM Transactions on Architecture and Code Optimization (TACO)* 15.3 (2018), p. 39.

[94] Tautvydas Maceina, Paolo Bettini, Gabriele Manduchi, and Mauro Passarotto. «Fast and efficient algorithms for computational electromagnetics on GPU architecture.» In: *IEEE Transactions on Nuclear Science* 64.7 (2017), pp. 1983–1987.

[95] P. Madhavan, P. Young, and S. Chang. «Performance of Scientific Simulations on QCT Developer Cloud: A Case Study of Molecular Dynamic and Quantum Chemistry Simulations.» In: *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*. June 2017, pp. 18–21. DOI: 10.1109/CSCloud.2017.54.

[96] Alessio Magro. «A Real-Time, GPU-Based, Non-Imaging Back-End for Radio Telescopes.» In: *arXiv preprint arXiv:1401.8258* (2014).

[97] Rainer Mankel. «Pattern recognition and event reconstruction in particle physics experiments.» In: *Reports on Progress in Physics* 67.4 (2004), p. 553.

[98] Fabienne CERN Marcastel. *WLCG*. Tech. rep. 2013.

[99] Timothy G Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[100] A. Mazouz, S. A. A. Touati, and D. Barthou. «Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of SPEC OMP applications on intel architectures.» In: *2011 International Conference on High Performance Computing Simulation*. 2011, pp. 273–279. DOI: 10.1109/HPCSim.2011.5999834.

[101] Abdelhafid Mazouz, Denis Barthou, et al. «Dynamic thread pinning for phase-based OpenMP programs.» In: *European Conference on Parallel Processing*. Springer Berlin Heidelberg, 2013, pp. 53–64. ISBN: 978-3-642-40047-6. DOI: 10.1007/978-3-642-40047-6_8.

[102] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439.

[103] Andrea Merli. «Search for CP violation in the angular distribution of $\Lambda_b^0 \to p\pi^-\pi^+\pi^-$ baryon decays and a proposal for the search of heavy baryon EDM with bent crystal at LHCb.» Presented 23 May 2019. May 2019. URL: http://cds.cern.ch/record/2687820.

[104] C. Misale. «Accelerating Bowtie2 with a lock-less concurrency approach and memory affinity.» In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Feb. 2014, pp. 578–585. DOI: 10.1109/PDP.2014.50.

[105] MissMJ, Cush. *Standard Model of Elementary Particles*. [Online; accessed July 4, 2020]. 2020. URL: https://commons.wikimedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg.

[106] Esma Mobs. *The CERN accelerator complex-August 2018*. Tech. rep. 2018.

[107] Gordon E Moore et al. *Cramming more components onto integrated circuits*. 1965.

[108] Janine Muller. «The LHCb SciFi Tracker: studies on scintillating fibres and development of quality assurance procedures for the SciFi serial production.» PhD thesis. Dortmund U.[, 2018.

[109] NVIDIA Corporation. *CUDA Toolkit Documentation*. NVIDIA Corporation, 2019. URL: https://docs.nvidia.com/cuda/.

[110] Tesla NVIDIA. *NVIDIA Tesla V100 GPU Architecture*. 2017.

[111] J. Nieto, D. Sanz, P. Guillén, S. Esquembri, G.de Arcas, M. Ruiz, J. Vega, and R. Castro. «High performance image acquisition and processing architecture for fast plant system controllers based on FPGA and GPU.» In: *Fusion Engineering and Design* 112 (2016), pp. 957 –960. ISSN: 0920-3796. DOI: https://doi.org/10.1016/j.fusengdes.2016.04.004. URL: http://www.sciencedirect.com/science/article/pii/S0920379616302848.

[112] Andreas Nowatzyk, Fong Pong, and Ashley Saulsbury. «Missing the memory wall: The case for processor/memory integration.» In: *23rd Annual International Symposium on Computer Architecture (ISCA'96)*. IEEE. 1996, pp. 90–90.

[113] Stuart Keble Paterson. «LHCb distributed data analysis on the computing grid.» PhD thesis. University of Glasgow, 2006.

[114] Shrikant Pawar, Aditya Stanam, and Ying 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.

[115] Daniel Hugo Cámpora Pérez, Omar Awile, and Cédric Potterat. «A high-throughput Kalman filter for modern SIMD architectures.» In: *European Conference on Parallel Processing*. Springer. 2017, pp. 378–389.

[116] Matt Pharr and William R Mark. «ispc: A SPMD compiler for high-performance CPU programming.» In: *2012 Innovative Parallel Computing (InPar)*. IEEE. 2012, pp. 1–13.

[117] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma. «Analyzing the Impact of CPU Pinning and Partial CPU Loads on Performance and Energy Efficiency.» In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pp. 1–10. DOI: 10.1109/CCGrid.2015.164.

[118] Miguel Ramos Pernas. *LHCb trigger in Run 3*. Tech. rep. 2020.

[119] Andre Recnik, Kevin Bandura, Nolan Denman, Adam D Hincks, Gary Hinshaw, Peter Klages, Ue-Li Pen, and Keith Vanderlinde. «An efficient real-time data pipeline for the CHIME Pathfinder radio telescope X-engine.» In: *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*. IEEE. 2015, pp. 57–61.

[120] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. «A Generic Parallel Pattern Interface for Stream and Data Processing.» In: *Concurrency and Computation: Practice and Experience* (2017), n/a–n/a. ISSN: 1532-0634. DOI: 10.1002/cpe.4175. URL: http://dx.doi.org/10.1002/cpe.4175.

[121] David del Rio Astorga, Manuel F Dolz, Javier Fernández, and J Daniel García. «A generic parallel pattern interface for stream and data processing.» In: *Concurrency and Computation: Practice and Experience* 29.24 (2017), e4175.

[122] Daniele Rogora, Michele Papalini, Koorosh Khazaei, Alessandro Margara, Antonio Carzaniga, and Gianpaolo Cugola. «High-Throughput Subset Matching on Commodity GPU-Based Systems.» In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: ACM, 2017, pp. 513–526. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064190. URL: http://doi.acm.org/10.1145/3064176.3064190.

[123] David Rohr, Sergey Gorbunov, Volker Lindenstruth, ALICE Collaboration, et al. «GPU-accelerated track reconstruction in the ALICE High Level Trigger.» In: *Journal of Physics: Conference Series*. Vol. 898. IOP Publishing. 2017, p. 032030.

[124] S Roiser and C Bozzi and. «The LHCb Software and Computing Upgrade towards LHC Run 3.» In: *Journal of Physics: Conference Series* 1085 (Sept. 2018), p. 032049. DOI: 10.1088/1742-6596/1085/3/032049. URL: https://doi.org/10.1088%2F1742-6596%2F1085%2F3%2F032049.

[125] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Neurocomputing: Foundations of research.» In: *ch. Learning Representations by Back-propagating Errors* (1988), pp. 696–699.

[126] Rupp, Karl. *FLOPs per Cycle for CPUs, GPUs and Xeon Phis*. [Online; accessed June 15, 2020]. 2016. URL: https://www.karlrupp.net/2016/08/flops-per-cycle-for-cpus-gpus-and-xeon-phis/.

[127] Rupp, Karl. *42 Years of Microprocessor Trend Data*. [Online; accessed June 14, 2020]. 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[128] Siddharth Samsi, Brian Helfer, Jeremy Kepner, Albert Reuther, and Darrell O Ricke. «A linear algebra approach to fast DNA mixture analysis using GPUs.» In: *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE. 2017, pp. 1–6.

[129] Manuel Tobias Schiller. «Track reconstruction and prompt K^0_S production at the LHCb experiment.» Dissertation, Heidelberg University., 2011.

[130] Mary C Seiler and Fritz A Seiler. «Numerical recipes in C: the art of scientific computing.» In: *Risk Analysis* 9.3 (1989), pp. 415–416.

[131] Priya Sen and Vikas Singhal. «Event selection for MUCH of CBM experiment using GPU computing.» In: *India Conference (INDICON), 2015 Annual IEEE*. IEEE. 2015, pp. 1–5.

[132] Yossi Shiloach and Uzi Vishkin. «An O (n2log n) parallel max-flow algorithm.» In: *Journal of Algorithms* 3.2 (1982), pp. 128–146.

[133] Herb Sutter. «The free lunch is over: A fundamental turn toward concurrency in software.» In: *Dr. Dobb's journal* 30.3 (2005), pp. 202–210.

[134] CERN The LHCb Collaboration. *Upgrade Software and Computing*. Tech. rep. CERN-LHCC-2018-007. LHCB-TDR-017. Geneva: CERN, Mar. 2018. URL: http://cds.cern.ch/record/2310827.

[135] Dean M Tullsen, Susan J Eggers, and Henry M Levy. «Simultaneous multithreading: Maximizing on-chip parallelism.» In: *ACM SIGARCH computer architecture news*. Vol. 23. 2. ACM. 1995, pp. 392–403.

[136] J Van Tilburg and M Merk. «Track simulation and reconstruction in LHCb.» PhD thesis. VRIJE UNIVERSITEIT, 2005. URL: http://cds.cern.ch/record/885750.

[137] Matthias Vogelgesang, Lorenzo Rota, Luis Eduardo Ardila Perez, Michele Caselle, Suren Chilingaryan, and Andreas Kopmann. «High-throughput data acquisition and processing for real-time x-ray imaging.» In: *Developments in X-Ray Tomography X*. Vol. 9967. International Society for Optics and Photonics. 2016, p. 996715.

[138] JONG-HOON WON, Dominik Dötterböck, and Bernd Eissfeller. «Performance Comparison of Different Forms of Kalman Filter Approaches for a Vector-Based GNSS Signal Tracking Loop.» In: *Navigation* 57.3 (2010), pp. 185–199.

[139] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. «Gunrock: A High-performance Graph Processing Library on the GPU.» In: *SIGPLAN Not.* 51.8 (Feb. 2016), 11:1–11:12. ISSN: 0362-1340. DOI: 10.1145/3016078.2851145. URL: http://doi.acm.org/10.1145/3016078.2851145.

[140] Samuel Williams, Andrew Waterman, and David Patterson. «Roofline: an insightful visual performance model for multicore architectures.» In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[141] Richard Wilton, Tamas Budavari, Ben Langmead, Sarah J Wheelan, Steven L Salzberg, and Alexander S Szalay. «Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space.» In: *PeerJ* 3 (2015), e808.

[142] Richard Wilton, Alexander S Szalay, Xin Li, and Andrew P Feinberg. «Arioc: GPU-accelerated alignment of short bisulfite-treated reads.» In: *Bioinformatics* 34.15 (Mar. 2018), pp. 2673–2675. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty167. URL: http://oup.prod.sis.lan/bioinformatics/article-pdf/34/15/2673/25230719/bty167.pdf.

[143] Jingzhou Zhao, Zhen An Liu, Wenxuan Gong, Pengcheng Cao, Wolfgang Kuehn, and Thomas Gessler. «New version of high performance Compute Node for PANDA Streaming DAQ system.» In: *arXiv preprint arXiv:1806.09128* (2018).

[144] Alexander Zlokapa, Abhishek Anand, Jean-Roch Vlimant, Javier M Duarte, Joshua Job, Daniel Lidar, and Maria Spiropulu. «Charged particle tracking with quantum annealing-inspired optimization.» In: *arXiv preprint arXiv:1908.04475* (2019).

[145] LHCb experiment. *Allen framework source code - Gitlab*. 2020. URL: https://gitlab.cern.ch/lhcb/Allen.