THÈSE DE DOCTORAT DE
SORBONNE UNIVERSITÉ

Spécialité
Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par
Arthur Hennequin

Pour obtenir le grade de
DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse:
Performance optimization for the LHCb experiment

présentée le 31 Janvier 2022

devant le jury composé de:

[Directeur de thèse]	Lionel	LACASSAGNE	LIP6	Sorbonne Université
[Rapporteur]	François	IRIGOIN	CRI	Mines ParisTech
[Rapporteur]	Denis	BARTHOU	INRIA	INRIA Bordeaux
[Examinateur]	Stef	GRAILLAT	LIP6	Sorbonne Université
[Examinateur]	Caroline	COLLANGE	INRIA	INRIA Rennes
[Examinateur]	Vladimir	GLIGOROV	LPNHE	Sorbonne Université

Abstract

The LHCb experiment, at CERN, is preparing a major upgrade of its detector and a change from an hardware-based to a fully software-based trigger system. It is now facing the challenge of being able to process incoming events at a rate of 30 million events per second. To cope with this massive data input, the software must be optimized to use the processing power of the filtering farm more efficiently. This thesis focus on the first algorithm of LHCb's High Level Trigger software: the Vertex Locator (VELO) reconstruction algorithm. The VELO is the first detector encountered by particles, directly surrounding the interaction region. Its goal is to find the initial track candidate that are then followed through the other layers of the LHCb detector with a good enough resolution that they could also be used to locate the origin of the collisions. The first step of this algorithm is to prepare the data by grouping pixels of the silicon sensors into hits; this process is called connected component analysis (CCA). This thesis presents multiple new CCA algorithms for both CPU and GPU architectures. The first algorithm, HA4, was developed at the very start of this thesis and improved the state-of-the-art in connected component labeling on GPUs, as well as being the first efficient implementation of connected component analysis on GPUs. The second algorithm is a GPU port of the FLSL SIMD CPU algorithm, inspired by the LSL algorithm. FLSL on GPUs improved upon HA4 by reducing the memory accesses conflicts that are especially presents on new hardware with a lot of cores. Along with FLSL, two other optimisations aimed at further reducing conflicts are presented and evaluated. On CPU, two new algorithms were made for this thesis. The first one is a modification of the classic Rosenfeld algorithm to use SIMD. The second one is a new algorithm, named SparseCCL, which takes advantage of the sparsity of the input images. A new VELO reconstruction algorithm using SIMD is presented, that enable LHCb to process events in real time and improve the quality of the reconstruction. The SIMDWrapper library, developed for the new VELO algorithm, is now part of LHCb's software and is used in other algorithms.

Acknowledgements

During my time as a CERN doctoral student I have often relied on the support and guidance of many people whom I would like to thank in the following:

Lionel Lacassagne for supervising my PhD thesis. I am particularly grateful for his detailed advices and encouragements during my master and PhD years.

François Irigoin and Denis Barthou for accepting to review this thesis manuscript and providing valuable suggestions.

Caroline Collange and Stef Graillat for being part of my jury.

Florian Lemaitre for his precious help and paving the way for efficient use of SIMD in LHCb's software.

Vladimir Gligorov, Ben Couturier and Sebastien Ponce for supervising my work in LHCb.

Gvozden Nešković from the Frankfurt Institute for Advanced Studies for lending me an AMD EPYC "Rome" 7742 system.

The Physics Data Processing group from Nikhef and in particular Tristan Suerink, for lending us the AMD EPYC "Rome" 7702 system used in VELO benchmarks.

E4, especially Marco Cicala, and Andrea Chierici from CNAF for lending me the AMD EPYC "Rome" 7302, 7452 systems as well as the Xeon Platinum 9242 system used in HLT benchmarks.

Eckardt Kehl, Rafael Kazumiti Morizawa, Lutz Weischer, Chris Derson, Frank Fijneman, Pierre Lagier and John Wagner from Fujitsu for giving me access to an A64FX and providing me with useful advices.

Christoph, Sascha, Niklas, Olli, Alex, Rosen, Marco, Victor, Conor, Louis, Andre, Michel, Laurent, Dominik, Claire, Niko, Concezio for welcoming me at CERN and all the useful discussions we had about the LHCb software.

Lastly, I would like to thank my family for their support, John for his help with the English of this manuscript and Clara for her patience during the writing of this manuscript and following me in this adventure.

Contents

Introduction

The Large Hadron Collider Beauty (LHCb) experiment is one of the four main experiments at the world's largest particle accelerator: the Large Hadron Collider (LHC), operated by the European Organization for Nuclear Research (CERN). Inside the LHC, particles follow a circular path of 27 km before being collided at nearly the speed of light. Collisions occur at a rate of 40 million times per second at four distinct locations, inside each of the main experiments: ALICE, ATLAS, CMS, and LHCb. To this date, the most important discovery at the LHC is the discovery in 2012 of the Higgs boson by the ATLAS and CMS collaborations. This major milestone filled a gap in the current best theory of particle physics, the Standard Model. While the Standard Model can predict a good number of experimental results with a very high level of precision, it cannot explain many other observed phenomena in our universe. For instance, the Standard Model does not provide any description of gravitational interactions, nor does it offer a viable dark matter particle possessing all of the required properties deduced from observational cosmology. This incompleteness has led physicists to search for a more general theory beyond the Standard Model.

To this end, the LHCb experiment is currently undergoing a major transformation that will allow it to take in more data, which in turn could help validating new theoretical frameworks to overcome the shortcomings of the Standard Model. This upgrade will replace several crucial parts of the detector and in particular switch from a hardware to a software trigger. The trigger is the detector component that analyses incoming data and decides, in real-time, if it should be saved for further analysis. After the upgrade, the detector is expected to generate about 3 TB/s of data, a volume that cannot be saved to disk, and so will have to be processed in real-time. Switching to a software trigger will allow the decision to be based on a more sophisticated reconstruction of the collision event which helps to better recognize signals of interest.

The goal of this thesis is to find optimizations of the LHCb trigger software that will help it to reach a throughput of 30 MHz[1] within the tight budget allocated to the trigger processing farm. This search starts with the analysis of the strengths and weaknesses of modern hardware. Through the study of the connected component analysis problem, a common part of computer vision processing chains, programming techniques of modern parallel architectures are explored. These findings helped develop new algorithms for the LHCb trigger and guided the general architecture of the software framework. The work presented in this thesis was central to the

[1] million events per seconds, abbreviated MHz as it is analogous to a frequency

successful construction of LHCb's trigger.

Chapter 1 introduces the LHCb experiment, its detector and the challenges faced by the new upgrade. Chapter 2 presents the evolution of central processing unit (CPU) architectures and the challenge of efficiently programming them to fully exploit the parallelism they offer. Chapter 3 explains how graphic processing units can be used beyond their original intended purpose to accelerate high performance computing workloads, and presents the programming techniques that are applied in this thesis. Chapter 4 presents the problems of connected components labeling and analysis, and the algorithms developed in the context of this thesis to efficiently address them on both CPU and GPU architectures. Chapter 5 describes a new SIMD VELO reconstruction algorithm developed for the LHCb experiment real-time trigger. Chapter 6 summarizes the evolution of the LHCb software performance during the last years and its evaluation on different CPU architectures.

Chapter 1

The LHCb experiment

Contents

1.1 Introduction

This first chapter presents the context of this thesis from the physics requirements to the software challenges. After an introduction to the Large Hadron Collider installations, the LHCb experiment's detector and its components are presented. Then, it introduces the challenge of building a new software High Level Trigger for the LHCb upgrade, which this thesis contributed to solve.

1.2 The Large Hadron Collider

The Large Hadron Collider (LHC) is the world's largest particle accelerator. It began operation in 2008 and is located in the same tunnel which previously hosted

the Large Electron-Positron Collider (LEP), dismantled in 2000. It consists of thousands of magnets forming a 27 km ring, located between 50 and 175 m underground. Most of the magnets are superconducting. They wrap two separate beam pipes, in which two particle beams travel in opposite directions at near the speed of light. These pipes are kept at ultrahigh vacuum conditions (10^{-11} mbar) so the particles do not interact with residual particles of air. The particles filling the accelerator are usually protons, but are occasionally replaced with ions for experiments requiring heavier particles.

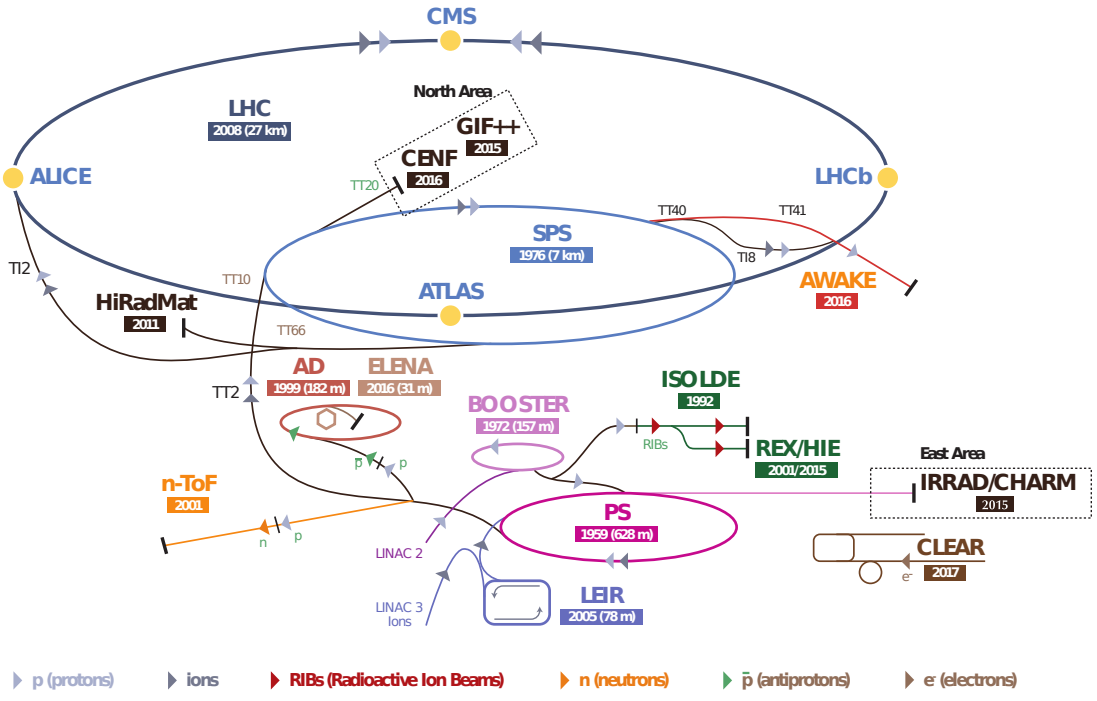


Figure 1.1: The CERN accelerator complex. Featuring the LHC, the four main experiments: ATLAS, ALICE, CMS, LHCb, the acceleration chain: LINAC2, BOOSTER, PS, SPS, and the auxiliary CERN experiments. [127]

The two beams are focused and "collided" at four dedicated interaction points, hosting the detectors of the four major experiments: ATLAS, ALICE, CMS and LHCb, shown on figure 1.1. The two largest experiments, ATLAS (A Toroidal LHC Appartus) and CMS (Compact Muon Solenoid) share a wide physics program, including the search for supersymmetry, dark matter and the famous Higgs Boson, jointly discovered in 2012 by both collaborations. ALICE (A Large Ion Collider Experiment) studies the physics of strongly interacting matter in heavy ion collisions. Lastly, the LHCb experiment (LHC beauty) is a general-purpose spectrometer, instrumented in the forward direction, optimized to investigate the asymmetry between matter and antimatter by studying a type of particle named the "bottom quark", also known as "b quark" or "beauty quark". The LHCb detector will be

presented in details in the next section.

Particles are accelerated and collided in bunches. When two bunches travelling in opposite directions collide, most of the particles cross each other and only a few collisions take place out of the 100 billion particles in each bunch. For each collision, the center of mass energy is the sum of each proton's energy. This is the total energy available for physic experiments. The design center of mass energy of the LHC is 14 TeV. The LHC has so far completed two successful periods of data collection (called Run 1 and Run 2), and is about to start its third (Run 3). During its first years, the LHC operated at 7 TeV, half its design energy; in 2012, the last year of Run1, it was increased to 8 TeV. Run 1 and 2 were separated by a 2 year long shutdown (LS1), during which the LHC machine was upgraded to enable collisions at higher energies. Run 2 operated at 13 TeV from 2015 to 2018. After the current long shutdown (LS2), the LHC will start again at the expected design energy of 14 TeV.

In order to reach their maximal energy, the particles injected in the LHC are gradually accelerated through multiple pre-accelerators. Starting in the LINAC2 linear accelerator, and followed by the BOOSTER circular accelerator, particles are first accelerated to an energy of 1.4 GeV. The beam is then fed into the Proton Synchrotron (PS) where it is accelerated to 25 GeV. Lastly, the bunches are accelerated to 450 GeV by the Super Proton Synchrotron (SPS), before being injected into the LHC in opposite directions. The beams are then accelerated for 20 minutes inside the LHC, until they reach the energy of 6.5 TeV (13 TeV center of mass energy) [39]. Figure 1.1 shows the journey of the protons (or ions) through the stages of the CERN accelerator complex.

Because of the small size of particles, only a fraction of them actually collide during a bunch crossing, also called an event. The probability of particle collision, and therefore the number of interactions, depends on the event rate and the beam's cross section. The design event rate of the LHC is 40 MHz (one event every 25 ns) and was reached during Run 2. The luminosity is the measure of the ability of a particle accelerator to produce the required number of interactions [83]. It is the proportionality factor between the number of events per second $\frac{dR}{dt}$ and the cross section of the beam σ_p.

$$\frac{dR}{dt} = \mathcal{L} \times \sigma_p \tag{1.1}$$

Luminosity is therefore measured in $\mathrm{cm}^{-2}\mathrm{s}^{-1}$ or in $\mathrm{fb}^{-1}\mathrm{s}^{-1}$ (1 femtobarn $= 10^{-39}$ cm^2). While the event rate of the LHC is fixed, the luminosity can be tuned per detector and is set to be increased during Run 3 for LHCb, enabling more interactions per event. Figure 1.2 shows the integrated luminosity recorded by LHCb during Run 1 and Run 2 between 2010 and 2018.

1.3 The LHCb detector

The LHCb detector is optimized to investigate the asymmetry between matter and anti-matter, also known as CP violation, through the high precision study of the decays of beauty and charm hadrons. It is a single arm forward spectrometer: instead of completely wrapping the interaction point like other experiments, it only

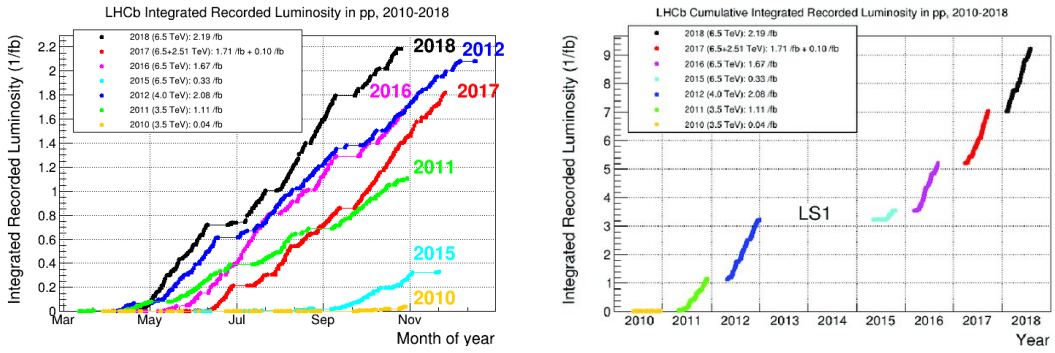


Figure 1.2: Integrated Luminosity recorded by the LHCb experiment. On the right, a breakdown per year. On the left, the cumulative integrated Luminosity over the first 8 years of the detector.

measures the particles going in the forward direction. The reason for this design is the specificity of the $b\bar{b}$ quark pairs, of special interest for LHCb, that are boosted along the beam axis. Its acceptance - the angular range covered by the detector - extends from 10 to 250 mrad vertically and from 10 to 300 mrad horizontally. The acceptance angle θ is often expressed as the pseudorapidity $\eta = -log(tan(\frac{\theta}{2}))$, for LHCb $2 < \eta < 5$. The LHCb coordinate system is cartesian, centered on the interaction point with the z axis parallel to the beam pipe, the y axis vertical, and the x axis horizontal.

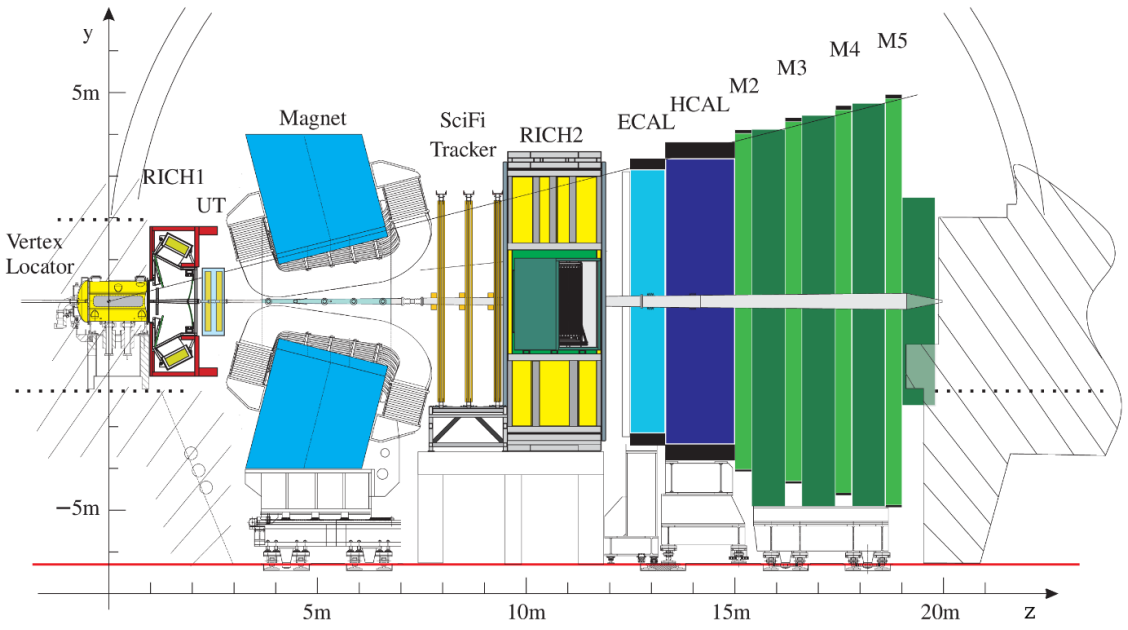


Figure 1.3: LHCb Upgrade Detector.

As part of the second long shutdown upgrade, some sub-detectors have been replaced to cope with the increased luminosity of Run 3 [45]. Figure 1.3 gives an overview of the LHCb upgrade detector. The sub-detectors can be classified into two functional categories: particle tracking and particle identification. The goal of particle tracking is to follow and characterize the trajectory of a charged particle

from its creation to its decay. Particle identification helps to measure properties of the tracked particles and find what type of particle they are.

Particle tracking is achieved using three sub-detectors: the Vertex Locator (VELO), the Upstream Tracker (UT), and the Scintillating Fibre Tracker (SciFi). Particles are created from the collisions occurring in the interaction region surrounded by the VELO, at $z = 0$ on Figure 1.3. The initial points of collision are called primary vertices (PV) and can spawn hundreds of measurable particles. The particles then continue their journey in all directions, but only those traveling in the forward direction ($z > 0$) and in the detector acceptance will be caught by the other detectors. Depending on their type, particles will decay into other kind of particles, sometimes splitting and potentially changing direction. The UT and SciFi placed on each side of the magnet are used to measure the momentum of the charged particles which is proportional to the bending of the trajectory inside the magnet. Due to decays and the limited acceptance of each detector, not all tracks go through every detector. Track types are thus classified into five categories, depending on which detectors the particles interact with. Long Tracks go through the three detectors. VELO tracks are only viewed in the VELO detector; they include the backward tracks representing the particles traveling toward negative z. Upstream tracks are tracks viewed in the VELO and UT but bent out of acceptance by the magnet because their momentum is too low. Downstream tracks are visible only by the UT and SciFi, and are often the product of a decay that occurs outside of the VELO. Finally, T tracks are only visible in the SciFi. Figure 1.4 shows the three tracking detectors and the corresponding track types.

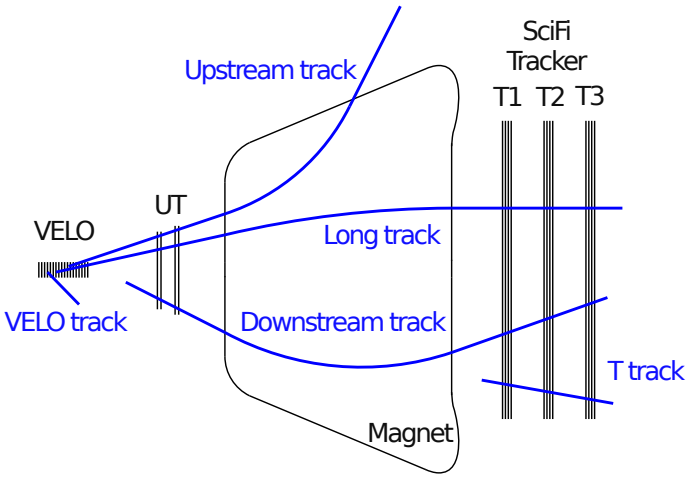


Figure 1.4: Track types for the LHCb Upgrade. Top down view.

Partial particle identification can be performed using only the tracking detector, by measuring the particle momentum or its displacement relative to the beam line. But for advanced analysis three other detectors are used. The Ring Imaging Cherenkov (RICH) detector is specialized in the measurement of particle mass. LHCb contains two RICH detectors, placed before and after the magnet (RICH1 and RICH2). The electronic and hadronic calorimeters (ECAL and HCAL) are used to measure the energy of particles, but to do so the particles are absorbed; the calorimeters are therefore placed behind the SciFi Tracker. Lastly, the Muon

stations (M2-M5) detect the remaining particles that pass through the calorimeters and several layers of absorbing material. Each LHCb sub-detector will be described in the following sections.

1.3.1 Vertex Locator

The Vertex Locator (VELO) is the first detector encountered by particles, directly surrounding the interaction region. To be as close as possible to the beam line, the beam pipe is removed and replaced by a 250 μm thin aluminium foil designed to protect the sensors from electromagnetic induction (RF foil). The whole detector is placed in a secondary vacuum. This allow the sensors to be positioned at only 5.1 mm from the interaction region, separated from it by the minimal amount of material that cause scattering.

A sketch of the VELO layout is shown in Figure 1.5. It consists of two halves that can be moved independently to be brought close to the beam line during data collection or away from it while the beam is being focused, to avoid damaging the sensors. Each half comprises 26 L-shaped modules assembled from four silicon pixel sensors. Each sensor consists of 768 $\times$ 256 pixels with a pitch of 55 μm. In total, the 52 modules feature about 41 million pixels.

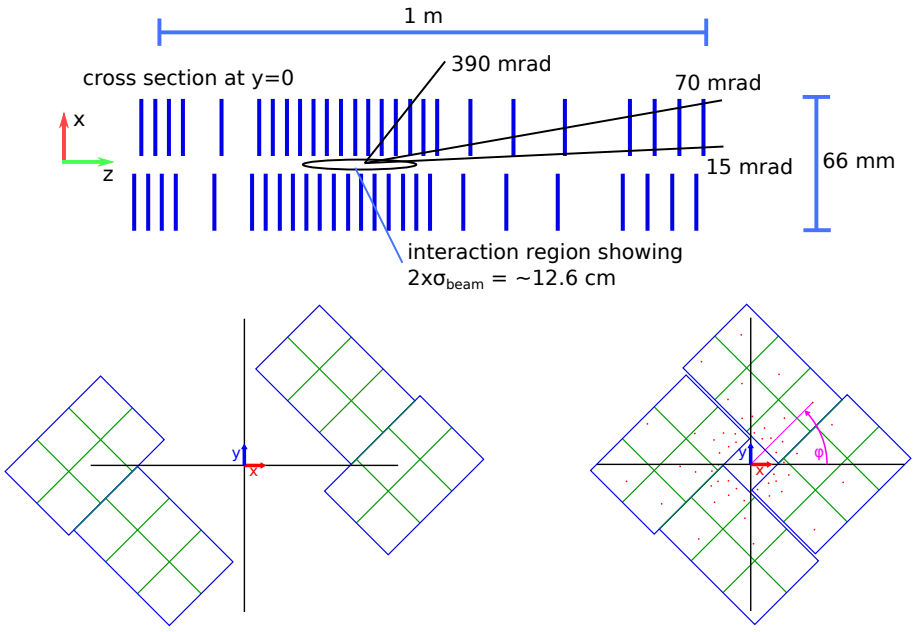


Figure 1.5: Vertex Locator (VELO) Geometry.

The VELO is located outside of the LHCb magnetic field. Therefore, the particles travel in a straight line inside of the detector. We define the track model inside of the VELO as the (x_0, y_0, T_x, T_y) vector. x_0 and y_0 are the x and y coordinates of the intersection with the $z = 0$ plane. T_x and T_y are the slopes of the track, defined as $T_x = \frac{\delta x}{\delta z}$ and $T_y = \frac{\delta y}{\delta z}$.

One of the main goals of the VELO detector is to precisely locate the primary and secondary vertices (PV / SV). PV represent the position of the initial proton-proton interaction, while SV are the positions of decays of beauty or charm hadrons within the VELO detector. PV and SV are essential to measure the lifetime of particles. Each vertex can be described as a 3-dimensional coordinated vector (x_v, y_v, z_v). While the resolution of each track is limited, the resolution of the vertices' position can be refined by combining many tracks.

To evaluate the detector performance, a crucial metric is the impact parameter (IP). The IP_{3D} is defined as the distance between a reconstructed track and a vertex. The IP resolution can be computed from simulation by measuring the IP of a track with respect to its origin vertex. This value is non zero due to the hit resolution and multiple scattering in the RF foil and detector material. The IP resolution is measured as a function of the particle momentum which directly affects the amount of scattering: the higher the momentum, the less the particle is subject to scattering. Figure 1.6 shows the expected IP_x of the VELO upgrade detector compared to the current VELO detector in upgrade conditions (with increased luminosity), as a function of the inverse transverse momentum.

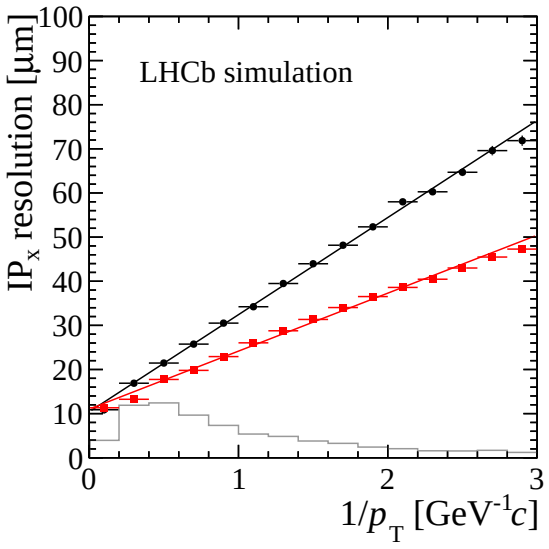


Figure 1.6: IPx resolution of long tracks for the VELO Upgrade (in red) compared to expected performance of the current VELO design in upgrade conditions (in black), as described in TDR [117].

1.3.2 Upstream Tracker

The Upstream Tracker (UT) is the second tracking detector. Its goal is to get an estimate of the charged particle's momentum before it goes through the magnet. To do so, it is placed inside the fringe of the magnetic field where it is weak enough that the particle doesn't deviate too much from a straight line estimate, but strong enough to have a measurable curvature. Inside this detector, the track model used is the 5 parameters LHCb track model (x_0, y_0, T_x, T_y, q/p), with q/p being the ratio between the charge of the particle and its momentum, also called curvature. The resolution of the momentum measured by the UT is only about 15% but this

measurement is essential to reduce the search time in later detectors and limit the amount of false matching of VELO tracks and SciFi hits.

The UT consists of four planes of silicon micro-strip sensors of various sizes and granularities, as shown in Figure 1.7. In the outer region (green), sensors are 10 cm long with a pitch of 190 μm between vertical strips. This design ensures a good resolution in the x direction, crucial for the momentum measurement, while allowing the sensor to cover larger distances in y. The middle and inner-most regions (yellow and red) use a 95 μm pitch to cope with the increased occupancy closer to the beam line. Sensors directly surrounding the beam pipe are also divided vertically, to increase the y resolution by a factor of two.

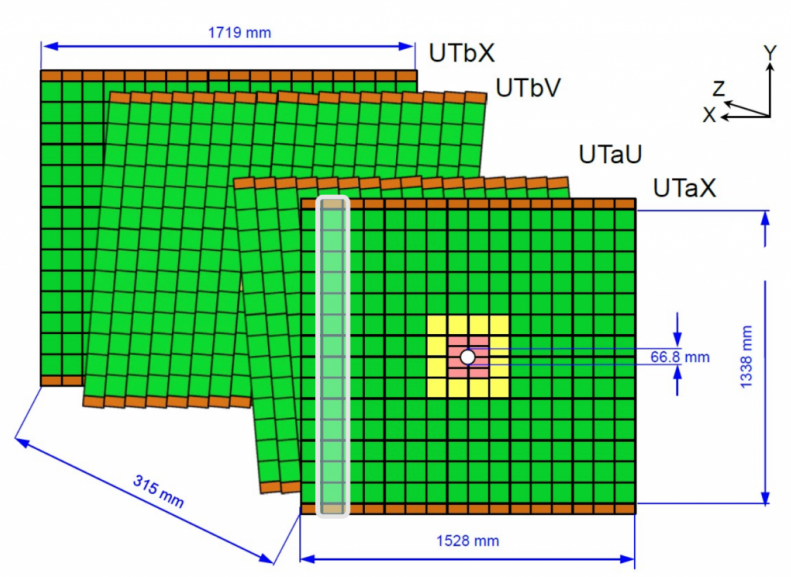


Figure 1.7: Upstream Tracker (UT) detector layout.

The two outermost planes are completely vertical and are called the X layers. The innermost planes are tilted by -5° and +5° and are called stereo or uv layers. These angles give information about the y coordinate of a track without having to increase the y resolution.

1.3.3 Scintillating Fibre Tracker

The Scintillating Fibre (SciFi) Tracker is the largest and last tracking subdetector. It is located after the magnet to measure with a high precision the curvature of the particles that went through it. It consists of 12 layers divided into 3 similar stations, named T1, T2 and T3, each consisting of two outer x layers and two inner stereo layers, similar to the UT. In the SciFi, the x and z components of the magnetic field vector can be neglected, but the y component is strong enough to induce some curvature in the xz plane. The track model inside the SciFi is therefore approximated as a straight line in the yz plane and a third order polynomial in the xz plane [12].

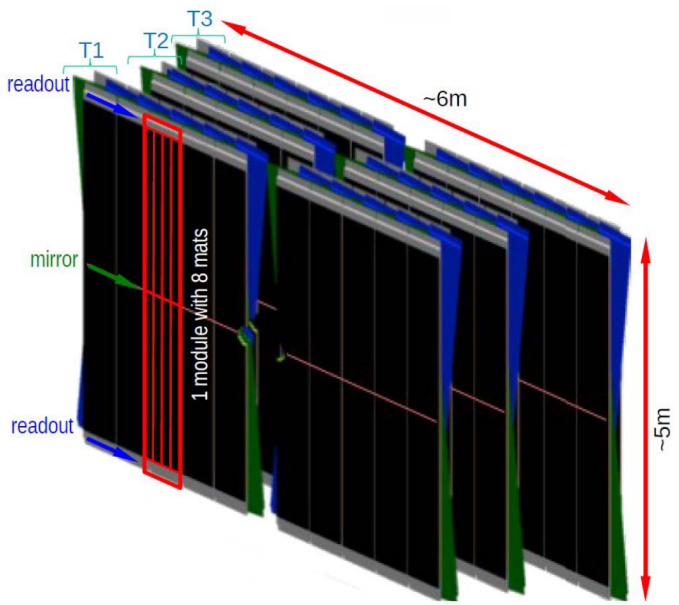


Figure 1.8: Scintillating Fibre (SciFi) Tracker layout.

As shown in figure 1.8, the layers are split into a top and bottom half. When a particle hits a fibre, a photon is created and travels through the fibre by internal reflection, to be read out by a high sensitivity photo-detector at the far end, or be reflected by a double-sided mirror at y=0. Each fibre has a diameter of 250 μm, allowing for a fine-grained measurement of the hit's x coordinates. Figure 1.9 shows an example of a reconstructed track in the SciFi Tracker with simulation data. Red dots are the hits left in the detector by the simulated particle. Each fibre position has been projected to y = 0, therefore the uv layers appear offset from the trajectory. This offset allows the reconstruction algorithm to compute the y intersect.

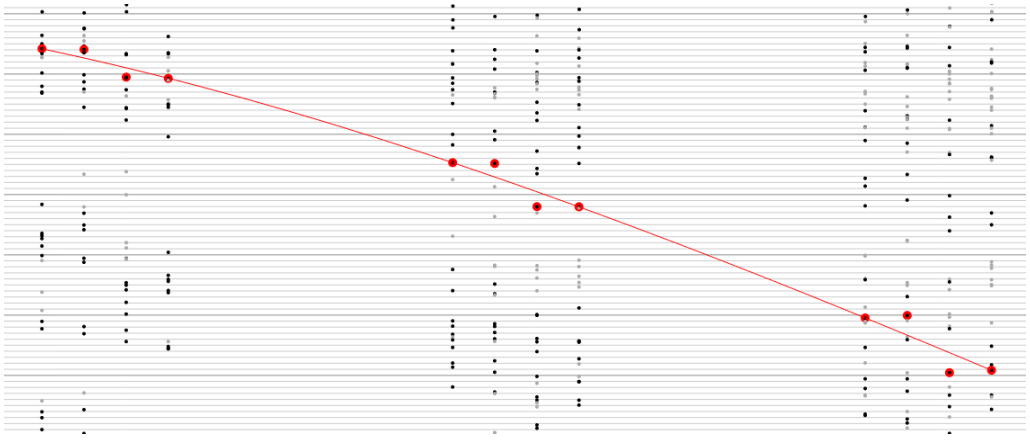


Figure 1.9: Example of a reconstructed track in the SciFi Tracker. Top-down view (xz plane). Data from LHCb Upgrade monte-carlo simulation.

With the other two tracking detectors, the SciFi allows the precise measurement of the momentum of long-lived charged particles with a resolution of 0.5 to 1%.

1.3.4 Ring Imaging Cherenkov Detectors

Ring Imaging Cherenkov (RICH) Detectors are a class of detectors that exploit the emitted Cherenkov light of a particle traversing a material at a speed higher than the speed of light in that material. When a particle traverses such material, photons are emitted in a cone around the particle's trajectory with an opening angle θ_c depending on the mass m and momentum p of the particle and the refractive index of the material n:

$$cos(\theta_c) = \frac{m}{np} \tag{1.2}$$

As the momentum is measured with high precision by the tracking detectors and the refractive index of the material is known, the mass of the particle can be deduced by measuring the opening angle. The choice of the radiator material is crucial to differentiating between particle types of interest within the desired momentum range.

The LHCb detector contains two RICH detectors. RICH1, placed between the Velo and the UT, uses the fluorocarbon gas C_4F_{10} as its radiator material in order to maximize the pion-Kaon (π-K) separation in the 10-40 GeV momentum range. RICH2, placed after the SciFi, employs the fluorocarbon gas CF_4 as its radiator, providing π-K separation up to 100 GeV momenta.

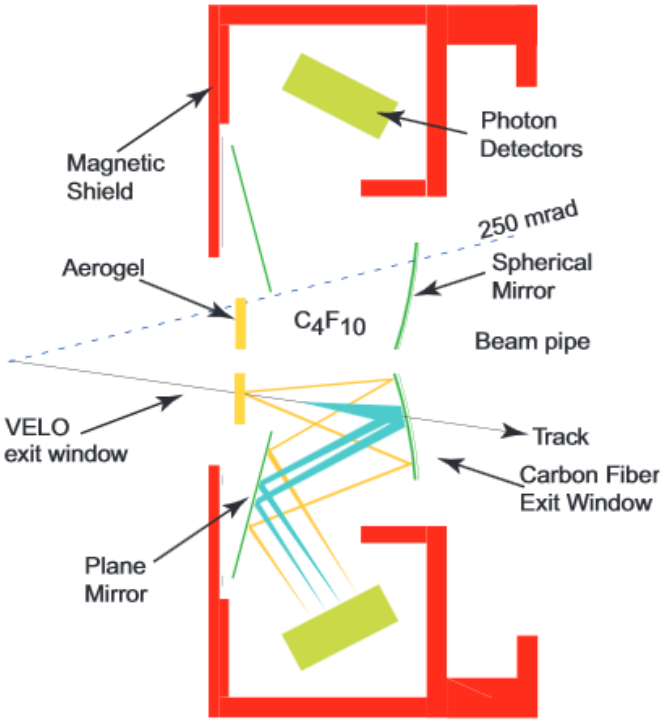


Figure 1.10: Layout of the RICH1 detector.

Figure 1.10 shows the layout of the RICH1 detector. At the end of the radiator material chambers, spherical mirrors bounce the emitted Cherenkov photons but allow the particle to pass through. Photons are then bounced again by planar mirrors

onto the photo-detectors panels. Each photo-detector provides a binary readout. Given a particle trajectory and its momentum, a mass hypothesis is done for each type of particle and then verified using a maximum-likelihood global identification process.

1.3.5 Calorimeters

Calorimeters are designed to measure the energy of particles. An absorbing plate produces a particle shower, which in turn produces photons in a layer of scintillating material. By counting these photons using photo-detectors, the energy of the stopped particles can be deduced. LHCb uses two calorimeters: an electromagnetic calorimeter (ECAL) and a hadronic calorimeter (HCAL). The ECAL employs lead as its absorbing material and measures the energy of photons and electrons, while the HCAL uses iron to measure the energy of hadrons, as shown in Figure 1.11. As this measurement is a destructive process, the calorimeters are placed after the RICH2, as depicted on Figure 1.3.

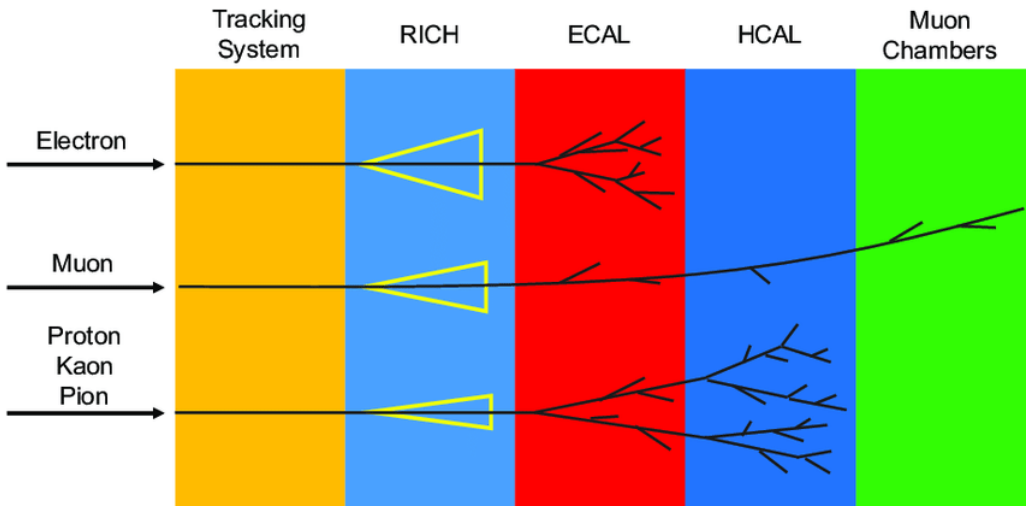


Figure 1.11: Illustration of different particle type responses in the LHCb systems.

1.3.6 Muon stations

The end of the line LHCb subdetector is the Muon stations. Detection of Muons plays an important role in the physics program of LHCb. It enables, for instance, the selection of rare decays like $B_s^0 \rightarrow \mu^+\mu^-$ (the decomposition of a strange B meson, composed of a bottom anti-quark and a strange quark, into an anti-muon and a muon). Muons are long-lived particles that go through every other detector unaffected. To detect them, LHCb employs four Muon stations, separated by 80 cm thick absorbing layers made of iron. Multi-wire proportional chambers (MWPC) are used as charged particle detectors in each station. Because at this stage, no other charged particles aside from Muons will have made it through, detecting them is equivalent to detecting Muons. By counting how many stations the muon can go

through before being completely absorbed, its momentum can be estimated.

1.4 The High Level Trigger

In the previous sections we have seen how events recorded at a rate of 30 MHz by the LHC are observed by the LHCb detector. On average, about 100 KB of data are produced for each event, resulting in a raw data production rate of 3 TB/s. Saving every event's data to long term storage would be impossible: even if the storage media could sustain such a writing speed, it would saturate our resources very quickly. The solution is to filter the events as they come and only save events containing valuable data according to the physics program. This decision is made by the trigger system.

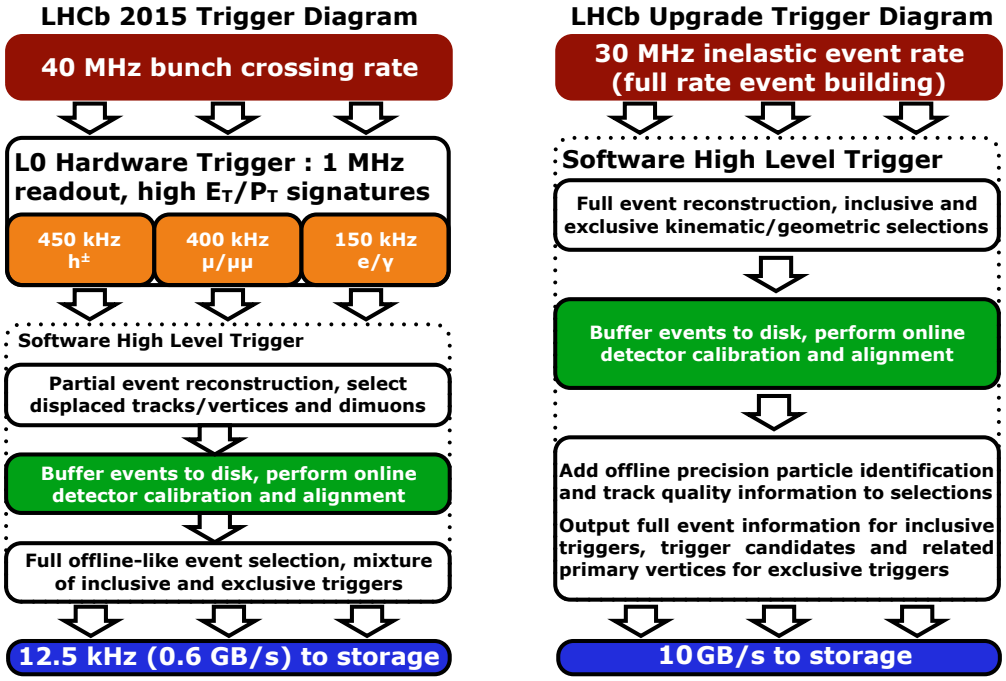


Figure 1.12: LHCb trigger system's diagrams for Run 2 (left) and Run 3 (right).

During Run 2, the trigger system was made of a first, hardware-based trigger stage called Level 0 (L0) and a second, software-based trigger stage called the High Level Trigger (HLT). The L0 stage employed FPGAs to reduce the event rate to 1 MHz based on the measurement of the transverse momentum in the calorimeters and Muon stations. The remaining events could then be processed by the 2000 CPUs HLT farm at a reasonable rate. However, the foreseen luminosity increase of Run 3 will result in a larger fraction of events containing a variety of valuable signals. This will require a trigger system that can not only discriminate between background and signal, but also classify what kind of signal is present. For this reason, the L0 stage is removed for Run 3 and replaced by a full software HLT. As with Run 2, this HLT is split into two stages: HLT1 and HLT2. The trigger diagrams for Run 2 and Run 3 are shown in Figure 1.12.

1.4.1 Event Building

To replace the L0 trigger, a completely new Data Acquisition system (DAQ) has been installed for Run 3. The challenge of such a system is to be able to forward each event's data at the production rate of the detector to the High Level Trigger. Each LHCb sub-detector has its own front-end electronics, performing basic data packing. Data from each individual subdetector is sent along optical fibres to the surface's event builder farm. The event builder farm consists of about 500 custom DAQ PCIE cards plugged into Commercial Off-The-Shelf (COTS) x86-based server nodes. To sustain the high throughput, data from multiple events are packed together in multi-events packets (MEPs) by the DAQ cards. Before sending the data to the HLT, it must be shuffled to group subdetector's data per events instead of having MEPs of different events for each subdetector. This shuffling is done by exchanging MEPs between the EB server nodes over a high-speed 100 Gbit/s InfiniBand network. This architecture is presented on figure 1.13 and has been carefully validated by the LHCb online team [138].

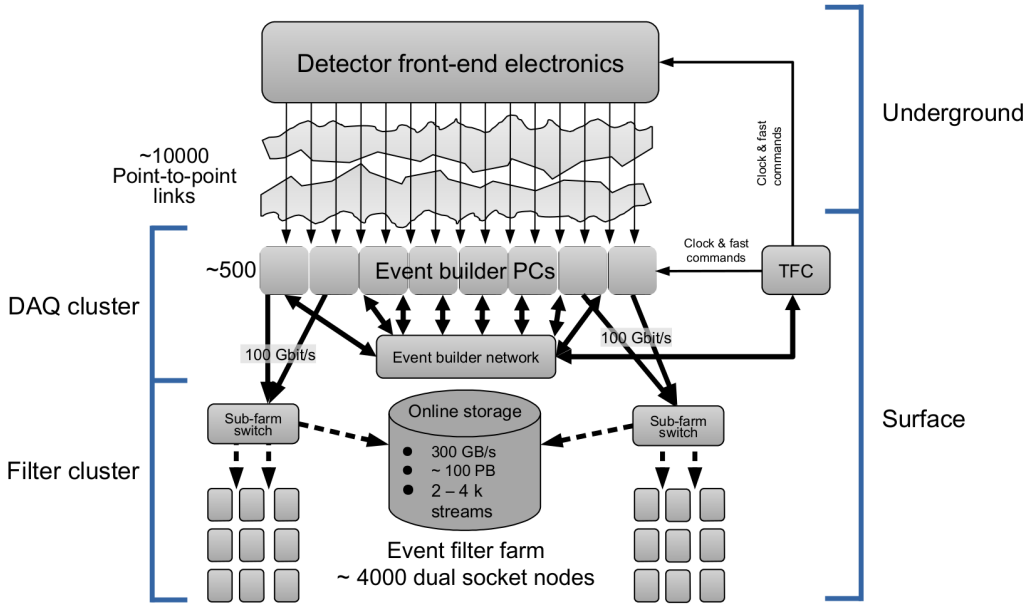


Figure 1.13: Architecture of the LHCb upgrade's readout system.

1.4.2 HLT1

The goal of the first trigger stage HLT1 is to reduce the incoming 30 MHz event rate to about 1 MHz by selecting only events that have a good chance of containing interesting signals. To do so, HLT1 performs a partial event reconstruction, which mainly consists of particle tracking through the VELO, UT and SciFi detectors, a Kalman Filter and Muon identification. Even without doing full particle identification, interesting features of the particles can be extracted from their trajectory only, such as their momentum, charge, or impact parameter.

The upgrade's HLT1 selection strategy is similar to the strategy adopted in Runs 1 and 2. Multiple scenarios have been proposed [13], combining different selection lines. A selection line is a logical expression combining different predicates that aim

at evaluating if a specific signal is present in an event. Selection lines are written and tuned by physicists and then evaluated on simulation and data from previous Runs. The predicates are sometimes referred to as "cuts" when they remove a fraction of the events from the output. Some cuts are motivated solely by computational performances. For instance, the Global Event Cut (GEC) aims to remove the 10% largest events, which take a lot longer to process.

1.4.3 HLT2

Unlike HLT1, the second trigger stage HLT2 makes no compromises on the physics efficiency of the reconstruction. It aims to reconstruct as many tracks as possible to enable full event reconstruction, so that no further processing is needed for offline physics analysis. To compensate for missing hits due to detector inefficiencies and to cover all LHCb track types, many redundant reconstruction paths are followed separately and combined to keep the best tracks. Long tracks are reconstructed by matching VELO and SciFi seed tracks, avoiding small inefficiencies caused by the UT. This process is much more costly than the HLT1 reconstruction but gives better physics efficiencies. In addition to long and upstream tracks, HLT2 also reconstructs downstream tracks from SciFi seeds and hits in the UT. Once all tracks are reconstructed, de-duplicated and filtered, RICH and calorimeter particle identification is performed. With this additional information, HLT2 is able to make a finer-grained decision about keeping a given event. The foreseen available output bandwidth of HLT2 during Run 3 will be between 2 and 10 GB/s. Therefore the amount of events HLT2 is able to save will depend on the average event size. To maximize the number of events which can be used for analysis, the Run 3 HLT2 will store a reduced event format which only includes the reconstruction data needed for offline analysis instead of the full raw data [121, 23, 5].

1.4.4 LHCb Software Framework

LHCb uses many different software packages for its simulation, high level trigger or offline analysis. All of these are built upon the same underlying framework called GAUDI, originally developed by the LHCb collaboration, and now used and maintained by the LHCb and ATLAS collaborations. The modern version of GAUDI is a scheduler capable of executing a functional dataflow algorithm, dispatched on many parallel execution units. The core of the framework is written in C++ but the algorithms can be configured and composed using a high-level API in Python. This configurability is essential to allow the creation of new physics scenarios without the need of software experts. The framework has been in constant development and evolution for 20 years, adapting to the important changes in both hardware and software. Some of the major changes of the hardware landscape will be presented in the next chapter. The following list presents the main software and libraries used by LHCb:

- **Gauss** is a package used to simulate how particles produced by proton-proton collisions interact with the detector. The simulation employs the technique of Monte-Carlo (MC) simulation, and the produced data is often referred to as MC data. It uses third party event generator libraries like PYTHIA [154] or

EVTGEN [106] and the GEANT4 [14] library for handling particle propagation in the detector.

- **Boole** is the software that simulates the effect of sensors and front-end electronics. It processes the output of Gauss to turn it into raw data similar to what is expected from the real detector.
- **LHCb** is a package containing the definitions for LHCb object types and low level libraries.
- **Rec** is a library of algorithms to perform event reconstruction, including tracking and particle identification used for both the HLT and offline analysis.
- **Allen** is the HLT1 GPU implementation which will be used in Run 3. It is written in CUDA and uses the same configuration framework as the CPU HLT1 and HLT2 implementations.
- **Moore** is the configuration framework for the online HLT. It is a collection of Python scripts that helps to define data-flows and selection lines.

The LHCb software framework is open-source and distributed under the GPLv3 license at *https://gitlab.cern.ch/lhcb*. As part of this thesis, my main contributions to the LHCb software framework were the development of new optimized reconstruction algorithms in Rec, as well as some improvements to the Event Model and low level libraries in LHCb. This work will be presented in the next chapters.

1.5 Conclusion

The LHCb Upgrade for Run 3 poses many new challenges. The combination of increased luminosity and the switch from a hardware to a software trigger imposes a high data processing rate of 3 TB/s which has to be achieved with commercial-off-the-shelf hardware. The framework builds on years of trusted physics software which has to be rewritten to fit the modern architectures. The usually short-time scale of software project development that allows for quick iterations has to adapt to the large time scale of a physics project which runs over a decade. At the time of writing, the LHCb Upgrade is about to be installed and the Upgrade II, due in 10 years, is already being planned.

Chapter 2

Parallelism on CPU

Contents

2.1 Introduction

This chapter reviews the evolution of CPU architectures over the last decades and the reasons that have driven them towards more and more parallelism. Modern architectures' key elements like caches or SIMD are presented and their impact on software development are discussed. Finally, a new library designed to simplify the use of SIMD on many different architectures: SIMDWrappers, is presented and compared with other SIMD libraries. This library was developed in the context of this thesis and used to implement some of the algorithms presented in other chapters.

2.2 CPU Architecture

The evolution of CPU performance over the last decades can be described by Moore's law and the two equations 2.1 and 2.2 [51]. Moore's law is an empirical law stating

that the number of transistors in an integrated circuit doubles every N months. The first equation is the time taken by a CPU to execute a program, a direct measure of its performance on a given task:

$$T_{CPU} = \frac{IC}{IPC \times F} \tag{2.1}$$

where IC is the instruction count of the program, IPC is the instruction per CPU clock cycle, and F is the clock frequency.

The second important equation is the power dissipation of CMOS circuits, expressed as the sum of the static power dissipation, the power dissipated when the circuit is idle, and dynamic power dissipation depending on the circuit's activity:

$$P_{CPU} = P_{static} + \alpha \sum C_i \times V_{dd}^2 \times F \tag{2.2}$$

where F is the clock frequency, V_{dd} is the power supply voltage, $\sum C_i$ is the sum of transistor gate and interconnection capacitance, and α is the average percentage of switching capacitance in the circuit.

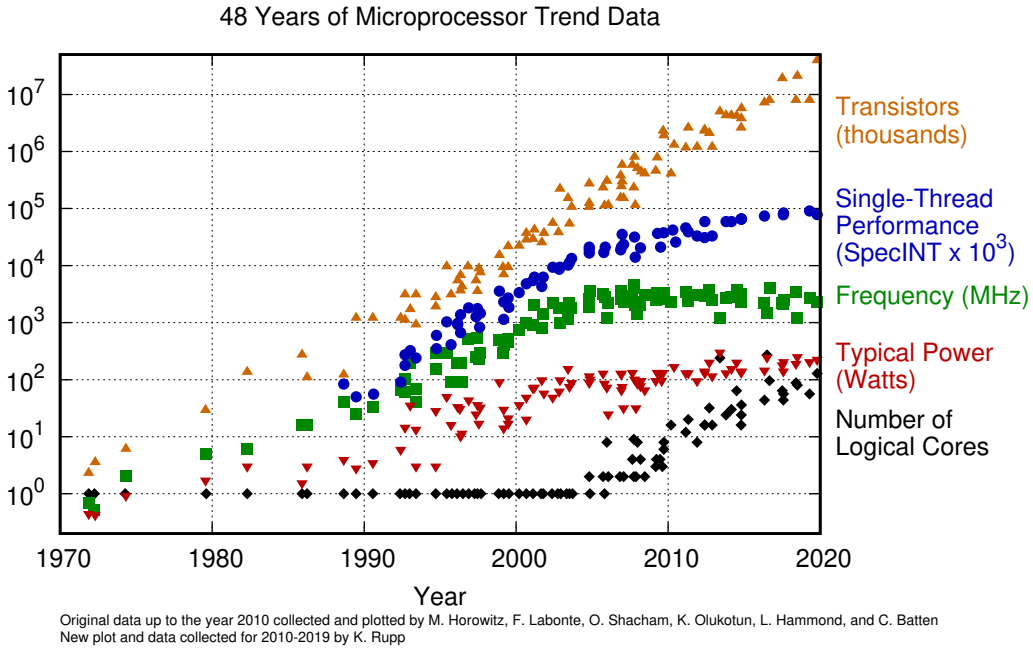


Figure 2.1: 48 years of Microprocessor Trend Data. [149]

As figure 2.1 shows, Moore's law is still valid and the transistor count continues to follow its exponential growth, but frequency has been stagnating for the last two decades. This is commonly referred to as hitting the power wall. In the absence of frequency increase, other means of improving performance had to be found. As pointed out by equation 2.2, the two remaining possibilities are reducing the instruction count and increasing the number of instructions executed per clock cycle (IPC). The former can be achieved with the help of optimizing compilers [94] and by exploiting the rich instruction sets made available by modern hardware, but has its limits. On the other hand, increasing IPC can be done in many ways that revolve

around the same idea: parallelism.

Modern CPUs feature parallelism on all levels. The first way to increase parallelism is to run applications on multiple logical cores. Since the power wall was hit, CPUs have seen their number of cores rising quickly from single core to today's server CPU with hundreds of cores. As the number of transistors continue to rise, many-core architectures become more and more common, even reaching the consumer market. But while running on different cores works well for independent applications, it relies on the duplication of hardware. A CPU core is made of multiple units each dedicated to a single task; when executing a specific instruction, not all units are contributing. Idle units can be used simultaneously for other instructions. This is the basis of instruction level parallelism (ILP). Modern CPUs have pipelines that allow them to decompose the execution of an instruction in multiple steps that can be executed in parallel with other instructions from the same program. This pipeline mechanism contributes to increase the IPC and is done automatically and opportunistically by the CPU. Another, more direct way to increase the IPC at the instruction level is to add a few more compute units, which are small in comparison to the memory subsystems. These compute units can be leveraged to execute the same instruction on multiple data during the same clock cycle. This architecture is called single instruction multiple data (SIMD), and is part of Flynn's taxonomy of computer architectures, as shown in figure 2.2. Unlike the pipeline optimisation, SIMD must be explicitly used by the developer or the compiler. SIMD will be discussed in depth in the next section, but first a few important aspects of CPU architectures will be presented.

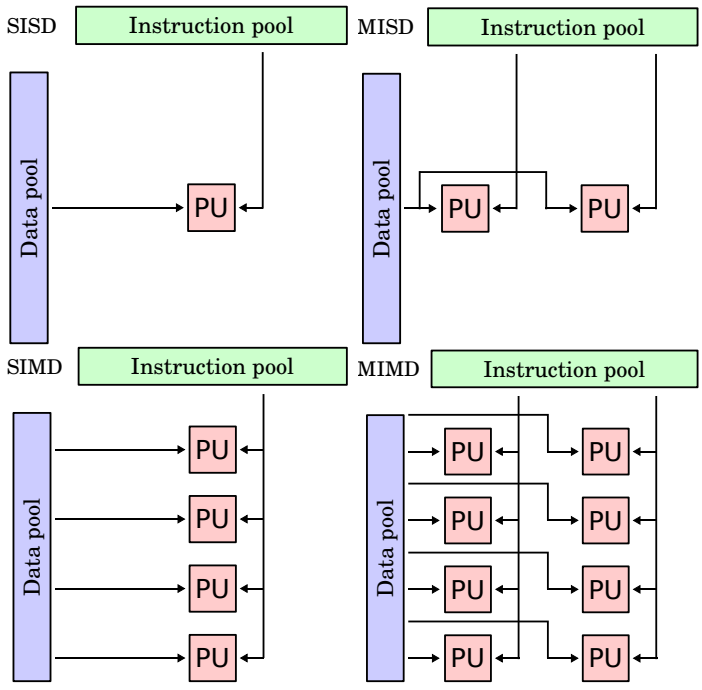


Figure 2.2: Flynn's taxonomy. Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Multiple Data (MIMD).

2.2.1 Multi-core architectures

Multi-core processors are manufactured on the same silicon integrated circuit, called a die. For CPUs in the same package but on different dies, the dedicated term is multi-chip modules. A multi-core processor can be classified as many-core when the core count is particularly high (tens to thousands). In a multi-CPU system, multiple physically distinct CPU packages are communicating with each other to form a single system with even more cores.

All of these techniques have as their common goal to increase Thread Level Parallelism (TLP). Threads are programs that can be executed and managed independently by the operating system scheduler. Threads can be executed concurrently, taking advantage of the multiple cores of the CPU, and can share the same resources, such as peripherals or memory. The sharing of memory allows communication between threads and enables cooperation between them. Multiple threads are grouped in processes, which represent a single application. Processes do not share resources like threads, but can still communicate in other ways, such as using network protocols.

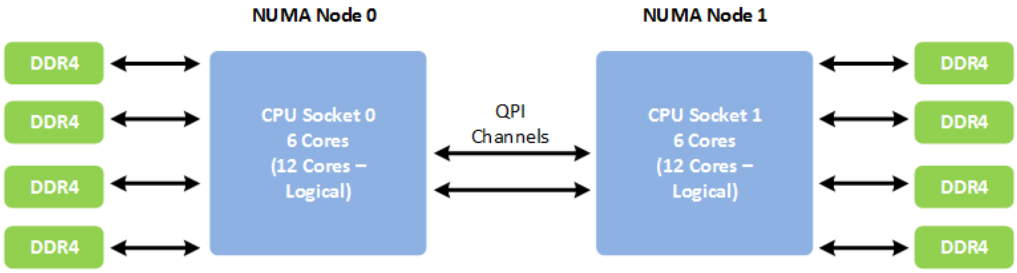


Figure 2.3: Schematic of a dual socket system. Each socket contains a 6 core / 12 thread processor and is linked to DDR4 memory through 4 channels. Sockets can communicate through the QuickPath Interconnect (QPI).

In the early days of multi-processing CPU architectures, all memory accesses from all CPUs of the system were going through a single shared physical bus. This design quickly limited the scalability of such systems where increasing the core count was also increasing the traffic on the memory bus, thus creating a bottleneck. To solve this issue, Non-Uniform Memory Access (NUMA) was invented. In a NUMA architecture, a small number of cores are tied to a single memory bus and form a NUMA node. CPUs can still access the memory from other NUMA nodes, but that access will have to go through multiple memory controllers and will have a higher latency than when accessing the local memory. Figure 2.3 shows an example configuration of a NUMA architecture.

Another way to increase thread level parallelism is Simultaneous Multithreading (SMT). Multithreading consists in running multiple threads on the same CPU core. Using multithreading on a superscalar processor, allows it to pull instructions from different streams to hide latency of other instructions. It is possible, because instructions from multiple threads have no data dependencies. SMT is one of two ways to implement multithreading, the other being temporal multithreading. In

temporal multithreading, only one thread can be executed in a pipeline stage at a time, while in SMT multiple threads can be executed in the same pipeline stage. SMT can be added to a physical CPU core with little change to the hardware: the main additions are the ability to fetch instructions from multiple threads in one cycle, and a larger register file so that each thread has its own set of registers. The number of logical CPUs per physical CPUs is usually two, but some CPUs can have four or even eight concurrent threads per core.

2.2.2 Cache Level Hierarchy

While the shrinking of transistors has allowed CPU performance to improve over the years, memory access speed has not progressed at the same pace. This growing gap between processor and memory performance, shown in figure 2.4, is becoming a bottleneck.

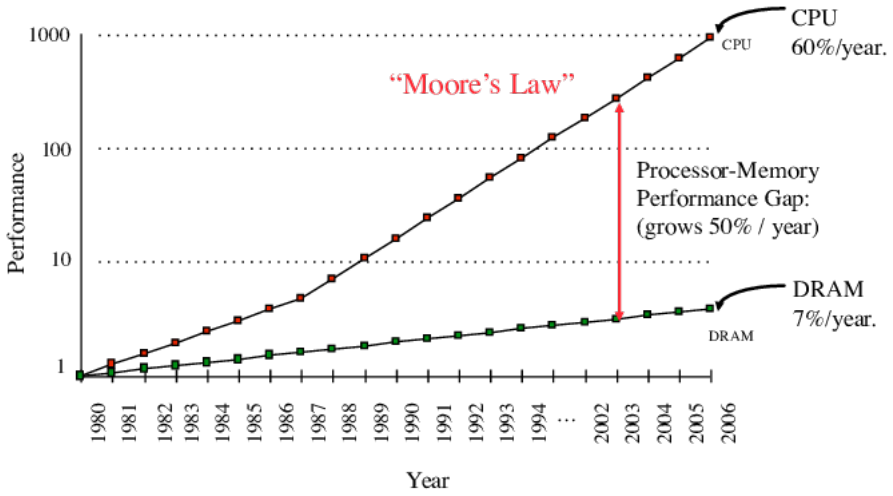


Figure 2.4: Evolution of processor performance and memory access speed over three decades. From [26].

Thankfully, while it is hard to manufacture large amounts of fast memory, smaller capacity memory can be made in similar processes as CPU logic and embedded directly on the CPU die. This embedded, faster memory is used as a cache for the main memory and comes in multiple levels. The Level 1 (L1) cache is usually divided into two functional parts: the instruction cache and the data cache. On modern hardware, each core usually has its own L1 cache that can hold 64 KB. The next levels, L2, L3 and sometimes L4, are increasingly bigger but slower, and are usually shared between multiple cores. Each cache level contains all the data of the lower levels. When the CPU tries to access data not yet in the cache, a cache miss occurs and causes the data to be fetched from the higher cache levels or from the main memory. Caches can implement different policies depending on the architectural choices. Cache policies are based on the principle of locality. Two types of locality are exploited: temporal and spatial localities. Following temporal locality, new data replaces the oldest ones in the cache, which are less likely to be used again. To also make use of spatial locality, data is fetched as a cache line, usually 64 bytes wide, which is the unit of data transfer between the cache levels

and main memory. Figure 2.5 shows the pyramidal organisation of the cache level hierarchy.

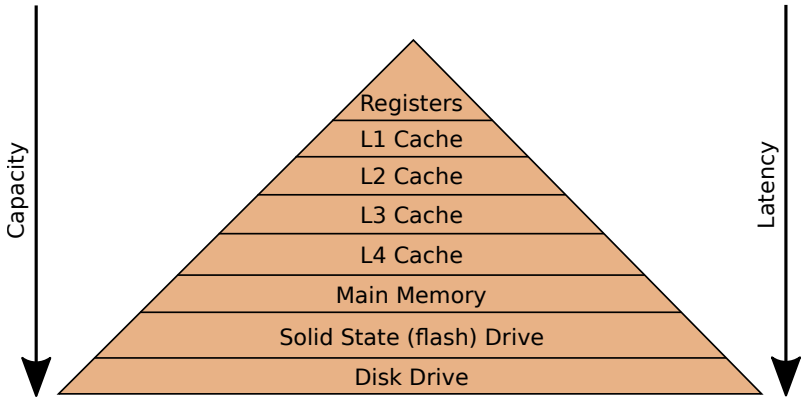


Figure 2.5: Cache Hierarchy.

2.2.3 Data Layout

As a result of the complex memory hierarchy, data layout is of primary importance for performance. Following the spatial locality principle, cache lines of 64 bytes are the unit of transfer between memory and cache. In addition, modern CPUs have SIMD registers capable of holding a full cache line. As a consequence, for a software to be able to reach its maximum throughput, data that must be used in parallel or in a short time span must be placed within the same cache lines.

In High Performance Computing (HPC), it is common to manipulate large collections of objects of the same type. Object oriented languages like C++ offer classes or structures to represent the type of any object: every object can be represented by a group of properties of primitive types and can be nested. It is also possible to construct arrays of these structures. The memory representation will then be a contiguous block of memory containing all objects one after the other. This data layout is called Array Of Structures (AoS). It is human friendly because the data layout matches the language model and respects the spatial locality principle for most scalar applications: if a program uses a property of the first object in an array, it may also need other properties of the same object soon. However, if the software parallelizes over the elements of the array, it would be better to bring the same property of different elements in the same cache line, to use the cache efficiently. This other layout, called Structure Of Arrays (SoA), allows the efficient use of SIMD instructions to exploit parallelism between independent elements of an array, but has the downside of deconstructing the object, making the code less human friendly.

To illustrate the difference between AoS and SoA, let's consider the following example: given an array of points represented by their 3D coordinates (x, y, z) and some weight (w), compute the weighted average. Listing 1 shows the C++ implementation for both AoS and SoA layouts. This loop pattern is trivial enough for the compiler to optimise it using SIMD instructions in both cases, but the code gen-

erated using the SoA layout, being more SIMD friendly, is about 23% more efficient[1].

```
// Array of Structures (AoS):
struct Point {
    float x, y, z, w;
};
Point centerOfMassAoS (const Point* points, const size_t N) {
    Point center{0.f, 0.f, 0.f, 0.f};
    for (size_t i=0 ; i<N ; i++) {
        center.x += points[i].w * points[i].x;
        center.y += points[i].w * points[i].y;
        center.z += points[i].w * points[i].z;
        center.w += points[i].w;
    }
    center.x /= center.w;
    center.y /= center.w;
    center.z /= center.w;
    return center;
}

// Structure of Arrays (SoA):
struct Points {
    float *x, *y, *z, *w; // pointers instead of plain values
};
Point centerOfMassSoA (const Points points, const size_t N) {
    Point center{0.f, 0.f, 0.f, 0.f};
    for (size_t i=0 ; i<N ; i++) {
        center.x += points.w[i] * points.x[i]; // index on each field
        center.y += points.w[i] * points.y[i]; // instead of the object
        center.z += points.w[i] * points.z[i];
        center.w += points.w[i];
    }
    center.x /= center.w;
    center.y /= center.w;
    center.z /= center.w;
    return center;
}
```

Listing 1: Center of mass computation of points in Array of Structures (AoS) and Structure of Arrays (SoA) layouts.

Another issue to consider when using SoA is that different properties of the same object can now be in different cache lines and possibly (depending on the number of elements in the array) far apart. Since the caches are smaller than the main memory, they must implement a replacement policy that decides where to store a given cache line depending on its address. The caches are partitioned into equally sized cache

[1]Measured on an AMD Ryzen 3800X, using g++ 8.4.0 with `-O3` option and 10^7 repetitions

sets. The number of cache lines in a set is called the associativity or the number of ways of the cache. When a cache line is loaded it is added to the set it belongs to, but if the set is already full, it must replace another cache line: this event is called a cache eviction. Traditional cache replacement policies include least-recently used (LRU), pseudo-LRU (PLRU) and first-in first-out (FIFO), but modern CPUs implement more complex policies that are not necessarily documented. When using SoA with a large number of properties, each corresponding to an array with one element per object, it may happen that cache lines of any one property end up in the same set as other properties. If the number of properties exceeds the number of ways in the set, it could lead to systematic cache eviction and a big performance drop. To tackle this effect, a hybrid data layout is often discussed: Array of Structures of Array (AoSoA). In the AoSoA layout, the collection is represented as an array of small SoA structures of fixed size. The size is chosen to break the alignment of properties and to be a multiple of the cache line size. Usually, the minimal size of one cache line is small enough to break the alignment while retaining the benefits from spatial locality. Another reason to use AoSoA is when filling a container while not knowing in advance the number of items it will contain. In this case, it allows the growth of the container by allocating another SoA section without having to move the existing items. Figure 2.6 gives an overview of the three data layouts that were discussed.

In practice, naturally occurring systematic cache eviction is not very common. Modern CPUs have a relatively large numbers of ways and it is rare to write loops accessing more cache lines than what could be held in a set. For instance, the Intel Xeon Gold 6130 has an 8-way set associative L1, 16-way L2 and 11-way L3. When using large 512-bit SIMD registers, cache lines can be stored entirely in registers and processed in one loop iteration, reducing the impact of cache eviction. Finally, for a cache eviction to happen, the cache lines have to map to the same set, an alignment which is easily broken by adding padding to change the size of the SoA structure.

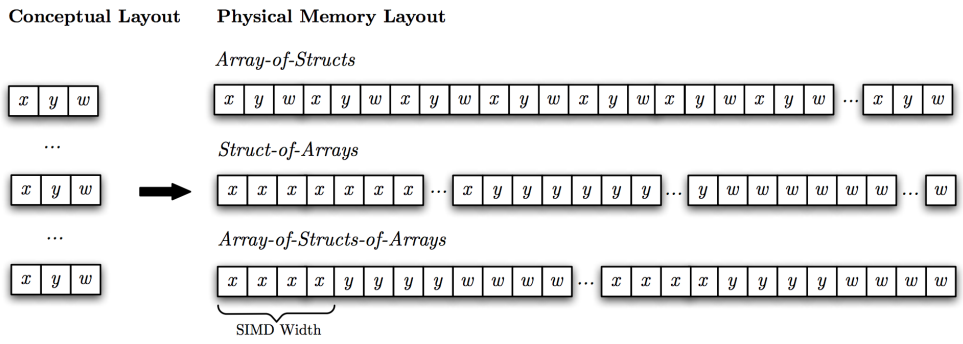


Figure 2.6: Three different data layouts.

2.3 Single Instruction Multiple Data

The Single Instruction Multiple Data (SIMD) paradigm of modern CPUs allow them to increase ILP by having special instructions, which take the same number of clock cycles as scalar instructions but operate on multiple elements at a time. Each

architecture family has its own instruction set that operates on special fixed size SIMD registers, usually 128- or 256-bits wide. The size of SIMD registers tends to increase in modern Instruction Set Architectures (ISA), as of 2021, some ISA support up to 2048 bits registers and hardware implementations exists for up to 512 bits. A typical SIMD ISA can be subdivided into three categories: memory access (loading or storing data from and to memory), arithmetic and logic operations (registers-to-registers mathematical or logical operations) and permute operations (changing the organisation of elements within a single vector register or mixing elements of multiple registers). As the ISA is dependent on the family of CPU used, the developer must be aware of architectural details when using SIMD and often change the structure of the algorithm to utilize it efficiently [61, 101]. There exist many solutions to use SIMD instructions:

- **Assembly language**: the most straightforward way of using SIMD instructions is to write them directly in assembly, either in an assembly program or using inline assembly from C/C++. The downside is that it creates hard to read code and sacrifices portability across architectures.
- **C/C++ compiler intrinsics**: compilers expose higher-level functions that map directly to the corresponding assembly instruction. But it is still hard to read, not portable, and very verbose.
- **SIMD libraries**: wrap the intrinsic to give them a standard name, abstracting architecture to make the program portable. In C++ wrapper libraries are able to use operator overloads and template meta-programming to improve readability.
- **Domain Specific Languages (DSL)** for parallel programming: usually require a different compiler or some extensions. Not only limited to SIMD, DSLs can also be used for other source code optimisation [59, 89, 96] or to target specialized hardware [17, 64]. DSLs are easier to use but lose fine-grained control; it is therefore not always possible to reach the best performance using this solution.
- **Automatic Vectorization** by the compiler provides free performance gains, but not all patterns are eligible for auto-vectorization. It is usually hard to get deterministic results.

For example, the assembly instruction to add the content of two single precision floating point vector registers in the SSE instruction set of the x86 architecture would be:

```
addps   xmm0, xmm1
```

The equivalent C/C++ compiler intrinsic would be:

```
__m128 c = _mm_add_ps(a, b);
```

And using a SIMD library, a DSL or autovectorization would look like this:

```
auto c = a + b;
```

2.3.1 Instruction sets

The term vector processor refers to any CPU that implements an instruction set containing instructions operating on one-dimensional arrays of data, called vectors. These instructions can be implemented in a data parallel way by having multiple instances of the same arithmetic and logic unit (ALU) executing the same instruction (SIMD) or by having the vector data flow through a pipeline of ALUs implementing different instructions (vector).

Early work on vector processing began in 1960 with the "Solomon" project of Westinghouse. The project was quickly abandoned but served as a foundation for the ILLIAC IV project started by the University of Illinois [21]. Released in 1972, the ILLIAC IV was the first vector machine, already using 64 double precision floating point units (FPU). In the ILLIAC IV architecture, every FPU could work on different operands, but were executing the same instruction, making it the first SIMD architecture. At about the same time, two other vector machines were developed: the Advanced Scientific Computer (ASC) of Texas Instruments [46], and the STAR-100 of Control Data Corporation [142]. On those machines, the pipeline approach was used: the vector instruction created a path from memory, through a scalar processing unit, to memory again. Because the vector elements were not processed in parallel, those vector architectures are not considered SIMD.

Improving upon the ASC and STAR-100, the Cray-1 machine [150], released in 1975, added vector registers. Those registers were able to store temporary results of vector operations, allowing the next instruction to start before the current one finished processing the complete vector. It also allowed the avoidance of unnecessary round trips to memory, but imposed a maximum size on the vectors (64 elements of 64 bits). Following Cray's innovation, other companies like Fujitsu, Hitachi and NEC introduced register-based vector machines. This period also saw the birth of massively parallel SIMD machines consisting of thousands of 1-bit independent ALUs. The Distributed Array Processor (DAP) [145] of International Computers Limited (ICL) was the precursor to this approach, in 1979, with 4096 1-bit processing elements. The most famous machine of this type is the Connection Machine [84], released in 1985, and featuring 65536 1-bit processing elements. The interest for vector machines in the supercomputer market faded in the 90s due to multi-core and multi-processor architectures, but reappeared when microprocessors started embedding new SIMD architectures and instruction sets.

The first appearance of SIMD in the microprocessor consumer market, in the mid 90s, accelerated multimedia application like video decoding. Existing architectures were extended to support new SIMD instructions that operated on registers and used as many ALUs as necessary to process every register element in parallel. More flexibility in the choice of individual element size is added with a technique known as SIMD Within A Register (SWAR) [53]. Operations like the addition of smaller data types are easily implemented with minimal hardware modification of the existing larger ALUs. For instance, eight 8-bit adders can be made from a single 64-bit adder by simply adding a switch on the carry from one 8-bit field to the next. The first architecture to get such SIMD extension is the PA-RISC with the 32-bit MAX-1 ISA in 1994, followed by the 64-bit MAX-2 ISA the next year. Other architectures followed

with VIS for SPARC in 1995, MDMX for MIPS-V in 1996, MVI for Alpha in 1996, and MMX for x86 in 1997. All these extensions are aimed at multimedia performance improvements and focus on integer instructions, supporting 8-bit, 16-bit and 32-bit elements within a 64-bit register. Most of them reuse existing registers. For instance, the MMX instruction set reuses the floating point registers of the x87 co-processor, resulting in some slowdown when trying to use simultaneously scalar and SIMD instructions. The first instance of floating point SIMD was the paired-single instruction set of the MIPS-V architecture, released in 1996. As the name suggests, it could execute two operations on single precision floating point numbers in parallel.

Extending MMX, Advanced Micro Devices (AMD) introduced the 3DNow! extension for single precision floating points to the x86 architecture in 1998. The same year, the Apple, IBM and Motorola alliance (AIM) developed the AltiVec instruction set for the "Performance Optimization With Enhanced RISC - Performance Computing" (PowerPC) architecture. Featuring 128-bit registers, it was able to process four single precision floating-point elements in parallel and was used in multimedia processing software. It is one of the most flexible SIMD instruction sets. In 1999, Intel introduced SSE, a 128-bit ISA for x86. Adding eight 128-bit registers, it didn't rely on the x87 co-processor like MMX did. But it was only able to process floating-points, and so MMX was still used for integers. In 2001, ARM released the Vector Floating-Point (VFP) instruction set. This architecture defined 16 registers of double precision floating-point, which could be also used as 32 single precision floating-point registers. Consecutive registers could be combined to form a vector register with a maximum size of 256 bits. Like the Cray machines, each element was processed sequentially in a pipeline. The next year, ARM introduced its own 32-bit SWAR instruction set to optimize multimedia applications. This ISA didn't have an official name but is referred to as "ARM multimedia extension".

In 2001, a new version of SSE, SSE2 added support for double precision floating-points, 8-, 16- and 32-bit integers and partial support for 64-bit integers to the x86 architecture. SSE2, like SSE, can also emulate scalar instructions by processing only the first element of the vector register. This allowed the x86 compilers to stop using the x87 co-processor for floating-point operations and use instead the SIMD FPU, which is compliant with the IEEE 754 [4] standard. Multiple SSE versions followed, with minor improvements, until 2008: SSE3 (2004), SSSE3 (2006), SSE4.1 (2007), SSE4a (2007) and SSE4.2 (2008).

In 2008, Intel and AMD specified the AVX instruction set, but the first machines to implement it were released in 2011. The main feature of AVX was its 256-bit width. It didn't introduce new registers but re-scaled the existing SSE registers and introduced a new instruction encoding, VEX, that supported instructions with three operands, reducing the pressure on registers. Improved integer supports, especially 64-bit integer, was brought by the next version of the extension: AVX2. It also introduced the Fused Multiply-Add (FMA) instruction family, which allows it to perform one addition and one multiplication on three operands in a single operation.

In 2008, ARM defined a new 128-bit SIMD instruction set: Neon. It uses 16 128-bit registers shared with VFP that can also be interpreted as 32 64-bit regis-

Architecture	Year	Type	Width
ILLIAC IV [21]	1972	SIMD	64 × 64
CDC STAR-100 [46]	1974	vector	N / A
TI ASC [142]	1974	vector	N / A
Cray-1 [150]	1975	vector	64 × 64
ICL DAP [145]	1979	SIMD	4096 × 1
Goodyear MPP	1983	SIMD	16384 × 1
Cray-2	1985	vector	64 × 64
NEC SX-2	1985	vector	256 × 64
Connection Machine 1 [84]	1985	SIMD	65536 × 64
CDC ETA10	1987	vector	?
NEC SX-3	1990	vector	256 × 64
Connection Machine 2	1991	MIMD	4 × 64

Architecture	Extension	Year	Type	Width	I8	I16	I32	I64	F16	F32	F64
PA-RISC	MAX-1	1994	SIMD	32	✓	✓					
PA-RISC	MAX-2	1995	SIMD	64	✓	✓	✓				
SPARC	VIS	1995	SIMD	64	✓	✓	✓				
MIPS-V	MDMX	1996	SIMD	64	✓	✓	✓				
MIPS-V	paired-single	1996	SIMD	64						✓	
Alpha	MVI	1996	SIMD	64	✓	✓	✓				
x86	MMX	1997	SIMD	64	✓	✓	✓				
x86	3DNow!	1998	SIMD	64						✓	
PowerPC	Altivec	1998	SIMD	128	✓	✓	✓			✓	
x86	SSE	1999	SIMD	128						✓	
ARM	VFP	2001	pipeline	32-256						✓	✓
x86	SSE2	2001	SIMD	128	✓	✓	✓	∼		✓	✓
ARM	media extension	2002	SIMD	32	✓	✓					
x86	SSE3	2004	SIMD	128	✓	✓	✓	∼		✓	✓
x86	SSSE3	2006	SIMD	128	✓	✓	✓	∼		✓	✓
x86	SSE4.1	2007	SIMD	128	✓	✓	✓	∼		✓	✓
x86	SSE4.2	2008	SIMD	128	✓	✓	✓	✓		✓	✓
ARM	Neon	2008	SIMD	128	✓	✓	✓	∼		✓	
PowerPC	VSX	2009	SIMD	128	✓	✓	✓	✓		✓	✓
x86	AVX	2011	SIMD	256	✓	✓	✓			✓	✓
PowerPC	QPX	2012	SIMD	256							✓
x86	AVX2	2013	SIMD	256	✓	✓	✓	✓		✓	✓
ARM	Neon (64 bits)	2013	SIMD	128	✓	✓	✓	✓	✓	✓	✓
x86	AVX512	2017	SIMD	128 - 512	✓	✓	✓	✓		✓	✓
ARM	SVE	2017	SIMD	128 - 2048	✓	✓	✓	✓	✓	✓	✓
NEC	TSUBASA	2018	vector & SIMD	256 × 64	?	?	?	?		✓	✓
ARM	MVE (Helium)	2019	SIMD	128	✓	✓	✓		✓	✓	
ARM	SVE2	2020	SIMD	128 - 2048	✓	✓	✓	✓	✓	✓	✓

Table 2.1: Evolution of CPU SIMD and Vector architectures.

ters. Neon supports 8-, 16- and 32-bit integers and single precision floating point. In 2009, the VSX extension for the PowerPC architecture added support for 64-bit integers and double precision floating points to AltiVec. It also added instructions

to precisely compute division and square root. In 2012, IBM created the 256-bit QPX extension aimed at scientific applications, which supports only double precision floating points.

In 2013, Intel proposed a 512-bit extension to AVX2: AVX-512. Not only did it re-scale the registers to add more parallelism, it also introduced new instructions and made some architectural changes. The number of registers was doubled from 16 to 32. Every instruction could be masked to ignore individual elements. Eight new registers dedicated to storing masks were added. New instructions like conflict-detection or compress-store were added in order to support more irregular algorithms that were not efficiently vectorizable before that.

In 2017, ARM and Fujistu announced the Scalable Vector Extension (SVE) instruction set aimed principally at High Performance Computing (HPC) and Machine Learning (ML) applications. This instruction set is vector length agnostic: the width of the registers is not fixed by the architecture and can vary between 512 and 2048 depending on the implementation. A code compiled using SVE could be executed on any architecture supporting SVE, independently on the implementation's width. Like AVX-512, SVE instructions can be masked. As of 2021, only one CPU implements SVE, the A64FX from Fujitsu made for the Post-K supercomputer, which features 512-bit SVE. In 2020, ARM announced the SVE2 extension, designed to enable applications beyond HPC and ML. In 2018, NEC released a vector machine in the form of PCIE accelerator cards, the SX-AURORA TSUBASA. This machine uses a hybrid SIMD / pipeline architecture. Its vector registers have a width of 16 384 bits cut in slices of 32 elements. Each element slice is processed in parallel following the SIMD paradigm, but the slices themselves are processed sequentially, taking advantage of the pipeline architecture.

Table 2.1 summarizes the timeline of vector architectures and SIMD instruction sets.

2.3.2 SIMD speedup and frequency scaling

As clock frequencies of modern processors are expected to stay near their current levels, or even to reduce, the primary method to improve the computation power of a chip is to increase either the number of processing units (cores) or the intrinsic parallelism of a core (SIMD). The speedup that can be achieved for a particular application depends on the amount of code that can be vectorized. Amdahl's law [16] gives a theoretical bound for the speedup:

$$speedup(c) = \frac{1}{1 - \tau + \frac{\tau}{c}}$$

where c is the SIMD cardinality, and τ is the fraction of vectorized code.

To reduce power consumption and help thermal stability, Intel CPUs use dynamic frequency scaling to limit the frequency of cores running SIMD instructions. There are three levels of frequency as shown in table 2.2. The frequency is reduced, per core, if the process encounters a sufficiently high density of instruction of the

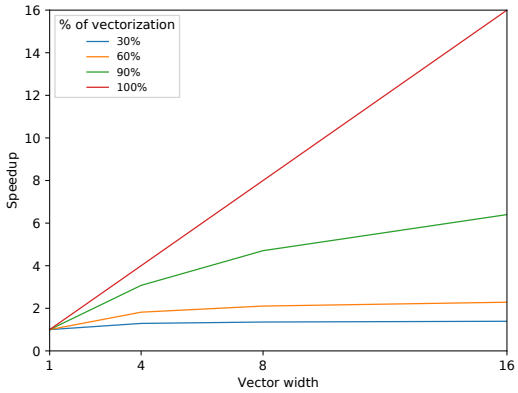


Figure 2.7: Amdahl's law applied to SIMD vector width.

corresponding type. The frequency reduction consists of multiple steps. When the CPU detects that no heavy instructions are used anymore, it waits approximately 2 ms before reverting the changes: in the mean time, scalar code runs at the lowered frequency [54]. Figure 2.8 illustrates the change of frequency induced by the usage of SIMD instructions. If the application interleaves scalar and AVX code, it will likely run at the AVX-induced lower frequency. We can modify Amdahl's law to account for this frequency scaling:

$$speedup(c) = \frac{1}{1 - \tau + \frac{\tau}{c}} \times \frac{freq(c)}{freq(1)}$$

$freq(c)$ is the maximum frequency for a vector of cardinality c. Figure 2.9 shows the theoretical speedups with frequency correction, for light and heavy instructions, for two different Intel CPUs. As we can see in Figure 2.10, for wide vectors a large amount of vectorized code is needed to keep increasing performance. To counterbalance the effects of frequency scaling, vendors added more specific instructions that can carry out complex operations in fewer cycles.

	Base	Turbo (1 core)	Turbo (10 cores)
Non-AVX / Light AVX2	2.2	3.0	2.5
Heavy AVX2 / Light AVX-512	1.8	2.9	2.2
Heavy AVX-512	1.1	1.8	1.4

Table 2.2: Maximum frequency (GHz) for an Intel Xeon Silver 4114 [1]

2.4 The SIMDWrappers library

2.4.1 Design objectives

Developing new algorithms that efficiently use SIMD architecture can only be done efficiently if the specific instructions are available. It is hard for a compiler to provide vectorization support for irregular algorithms. Domain Specific Languages (DSLs) such as Halide [143] or SPMD [136] do not contain all patterns necessary for our

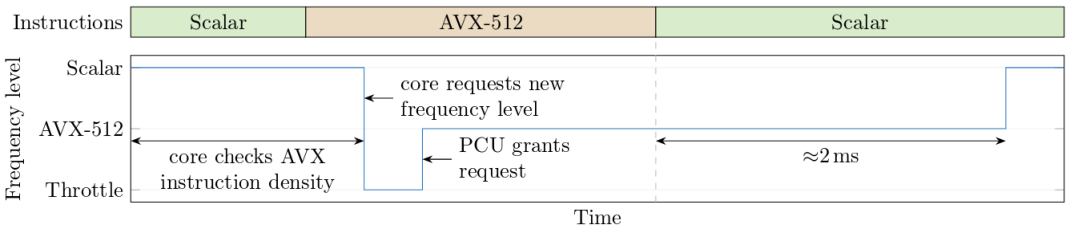


Figure 2.8: Frequency levels when an Intel Skylake-SP core temporarily executes 512-bit FMA instructions. After AVX-512 usage has been detected, the core executes at reduced performance while requesting a new power license level. Once the request has been granted, the core switches to the new frequency. (Image source: [54])

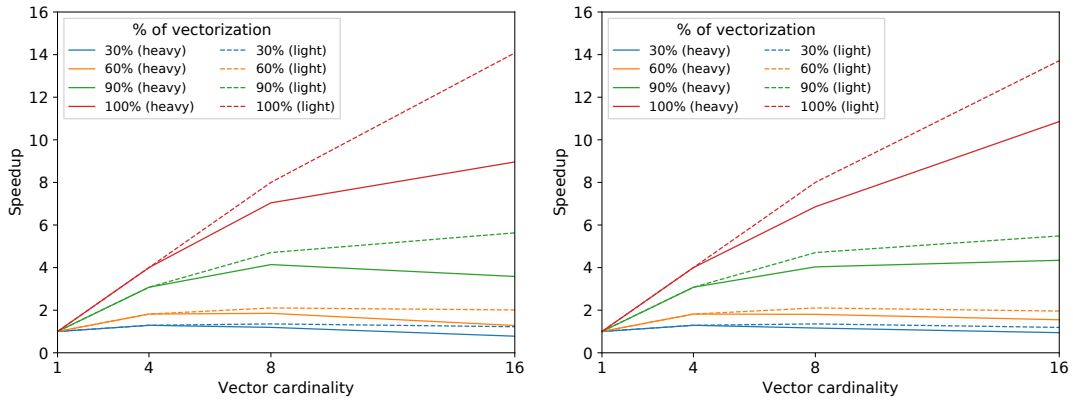


Figure 2.9: Amdahl's law applied to SIMD vector width with frequency correction, for an Intel Xeon Silver 4114 (left) and an Intel Xeon Gold 6130 (right).

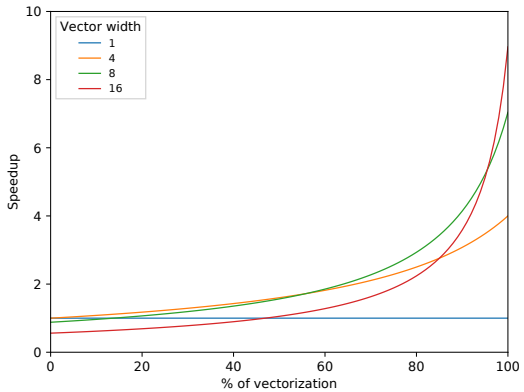


Figure 2.10: Speedup expected from % of vectorization for heavy instructions.

problems and would also introduce significant additional complexity in the context of the LHCb codebase, which is almost entirely written in C++ and Python. A less invasive option is to use SIMD libraries, like VC [100], UME::SIMD [92] or VecCore [15], that wrap the compiler intrinsics to provide a higher abstraction level to the developer.

While these libraries work well for implementing most algorithms, they do not currently fully implement the latest SIMD ISA extensions. I therefore developed

a set of vector length agnostic C++ template codes that can be instantiated for different SIMD backends in order to evaluate the impact of SIMD width on the performance and allow our algorithms to be ported on a wide range of architectures. In order to evaluate the impact of vector size on AVX-induced frequency scaling, I implemented two backends that use AVX-512 instruction variants for 256- and 128-bit wide vector registers: AVX256 and AVX128[2]. For simplicity, and because it matched LHCb's use case, the implementation is limited to 32-bit elements for integer and floating point types. The SIMD backend is determined by the developer, and is resolved at compile time following the fallback scheme depicted in Figure 2.11. This allows SIMD backends to be mixed and provides easy debugging and testing capabilities while ensuring portability. The main focus of the library was to implement backends for all major x86 SIMD instruction sets, because it is the architecture used by LHCb for its trigger application. In order to evaluate ARM CPUs, Neon and SVE backends were later implemented.

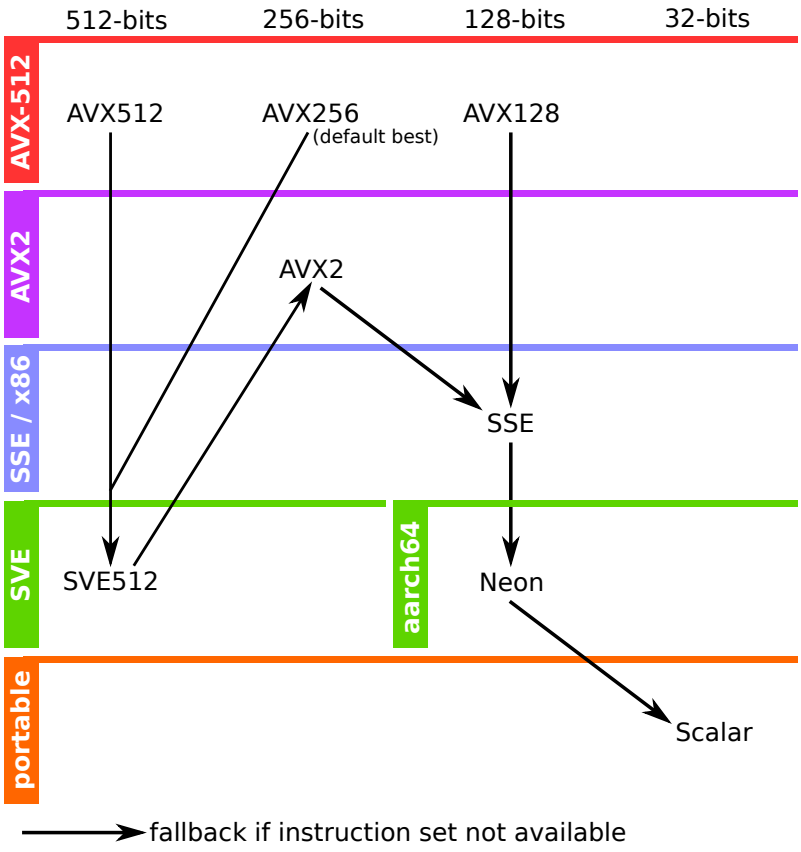


Figure 2.11: Available SIMDWrappers' backends and their fallback strategy. The user can ask for a specific backend. At compile time, if the backend is not available on the target architecture the compiler will follow the arrow until a valid backend is found.

Alongside the templated SIMD types for integers, floating points and masks, the library also provides mathematical functions that operate on those types, such as approximations of trigonometric and logarithm functions. High-level mathematical functions are not part of the SIMD instruction sets and the standard maths

[2]An approach I also used in a previous article [80]

library (libm) only defines them for scalar uses. While several SIMD libm have been developed [107, 137], they are aimed at auto-vectorization. Templated mathematical objects and linear algebra operators were developed with SIMDWrappers in mind. Based on the idea that SIMD and data layout are tightly coupled, the SOACollections library was developed to define SOA types and interface them with SIMDWrappers. In SOACollections, collections of objects are defined as the set of properties, or fields, that compose them. Fields can be integers, single precision floating points or any structures composed from them. Arrays can be defined if their size is known at compile time. The user interface of the library was designed to abstract away the SOA aspect of the data and to be as close as possible to the AOS layout to which users are accustomed. For example, a simplified track object can be defined as:

```
namespace TrackTags {
    struct position : SOACollections::vec3_field {};
    struct direction : SOACollections::vec3_field {};

    template<typename T>
    using track_t = SOACollections<T, position, direction>;
}
struct Tracks : TrackTags::track_t<Tracks> {
    // optional proxy specialisation and container level methods
};
```

In this example, a track is defined as a 3D point and a direction, describing the 3D trajectory as a straight line. The `vec3_field` type is defined by the library and defines a field that could be manipulated as a 3D vector object by the SIMDWrapper library. Collections are meant to be accessed using proxies to mimic the scalar case. When a collection is defined, the library defines automatically an associated proxy type that can be templated with the SIMD backend used to access it. A proxy object therefore represents a chunk of the collection that matches the size of the SIMD vector. Proxies can be used to append an object to the collection:

```
// Define which SIMD backend to use
using simd = SIMDWrapper::InstructionSet::Scalar;
// Append and return a proxy object, resize the collection if needed
auto track_proxy = tracks.emplace_back<simd>();
// Use the proxy to set the fields, if SIMD is used multiple tracks
// could be initialized at once
track_proxy.field<TrackTags::position>()
           .set(Vec3<float>{1.f, 2.f, 3.f});
track_proxy.field<TrackTags::direction>()
           .set(Vec3<float>{0.f, 0.f, 1.f});
```

The proxies can also be used to iterate over tracks. The following example shows a simplified version of the Impact Parameter (IP) filter algorithm. This algorithm takes a collection of VELO tracks and a collection of reconstructed primary vertices. The IP of a track is defined as the distance to the closest vertex. The algorithm goal is to keep tracks that have an IP greater than a threshold.

```cpp
using simd = SIMDWrapper::InstructionSet::Best;
using F = simd::float_v;
// iterate over the tracks collection using the best SIMD
// backend available
for (auto const& track : tracks.simd<simd>()) {
    // true if the corresponding mask element is in the collection,
    // false if the index is greater than the size of the collection
    auto loop_mask = track.loop_mask();

    // get the origin and direction of the track state
    Vec3<F> B = track.get<TrackTags::position>();
    Vec3<F> u = track.get<TrackTags::direction>();

    // Check all vertices to find the closest one
    F min_distance = 10e3;
    for (auto const& pv : vertices.scalar()) {
        // pv is representing a single vertex but the track proxy
        // can represent multiple tracks, the pv position need
        // to be adapted to match the number of tracks by using
        // a broadcasting cast:
        Vec3<F> A = Vec3<F>(pv.x(), pv.y(), pv.z());
        auto distance = (B - A).cross( u ).mag2();
        min_distance = min( min_distance, distance );
    }

    // compute the IP and make the selection
    auto trackIP = sqrt(min_distance) / u.mag();
    auto mask = ip_cut_value < trackIP;

    // copy only the tracks that passed the selection test
    tracks_out.copy_back<simd>(tracks, track.offset(), mask && loop_mask);
}
```

Filtering collections is a recurrent pattern in the HLT reconstruction algorithm. The `copy_back` function added by the SOACollections library help to implement this pattern efficiently using the `compressstoreu` instruction when available or falling back to an efficient emulation of it.

The SIMDWrappers library was globaly adopted by the LHCb collaboration. In particular, the SOA track representation became the standard for implementing the LHCb event model and the library allowed the vectorization of the selection process [120]. Selections are defined in a Python configuration, that generates C++ code compiled just-in-time. SIMDWrappers allowed to add SIMD without too many modifications to the existing code generator. Figure 2.12 shows how a speedup of up to 65% was achieved in the particle combination algorithms used by the selections. The execution times for the "2-body loose" and "3-body" algorithms are lower for backends which utilise vector instruction sets. The "2-body tight" algorithm has a tight selection on its input objects and subsequently cannot fully fill the vector instruction registers during execution.

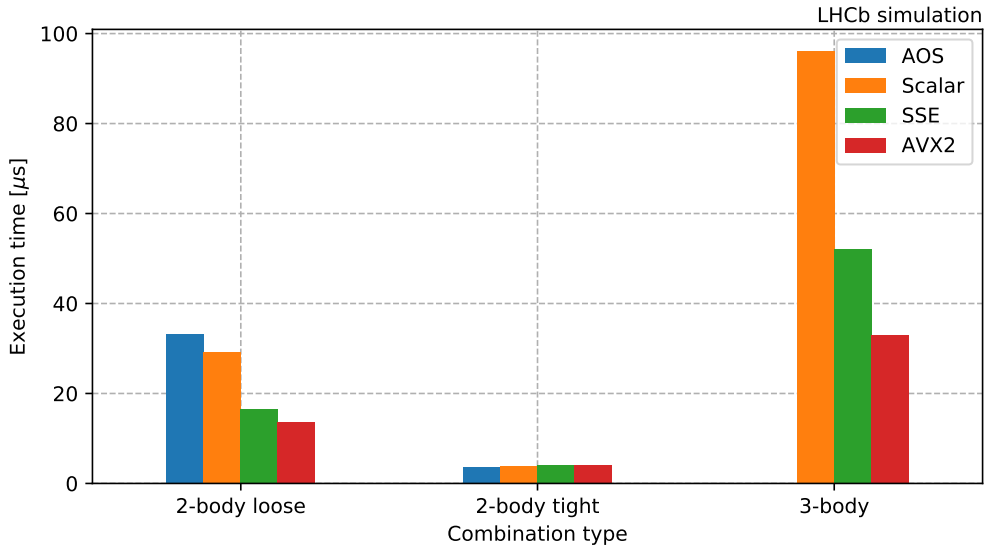


Figure 2.12: The timings of three different particle combination algorithms as performed by four different execution backends [120].

2.4.2 Comparison with other SIMD libraries

This section reviews some of the other SIMD libraries and explains the features unique to SIMDWrappers. The libraries listed are, to my knowledge, the main libraries still maintained or in active development. Only open-source libraries are discussed here.

SIMD libraries can be classified according to the instruction sets they support and which data type they expose. Table 2.3 summarize the backends and data types of each library. SIMDWrappers is the only library that provides distinct, interchangeable backends for AVX-512, allowing the systematic study of the impact of AVX-induced frequency scaling on Intel architectures. The 512-bit SVE backend was developed and tested on the Fujitsu A64FX. The choice of restricting the vector types to 32-bit integers and floating-points was made to simplify the user interface. Having every element be the same size allows SIMD code similar to scalar code to be written, because every vector register contains the same number of elements. This choice may restrict the algorithms one can implement with the library but provides a simple interface with the SOACollection library.

Some libraries like Boost.SIMD or simdpp use C++ expression templates. This technique forces the compiler to rewrite arithmetic expressions into dedicated SIMD instructions. A typical example is the Fused Multiply and Accumulate (FMA) instructions that could compute expressions such as $a*b+c$ in one cycle. The drawback of this technique is the complexity it adds to the library and the large cryptic errors the compiler can produce. Fortunately, modern compilers already reliably rewrite expressions into FMA instructions. For this reason, SIMDWrappers don't use expression templates and instead rely on the compiler to infer FMA from addition and multiplication SIMD instructions.

Name	SSE	AVX2	AVX-512			Neon	SVE	AltiVec	Integer				Float		Math
	128	256	128	256	512	128	512	128	8	16	32	64	32	64	Func.
MIPP [38]	✓	✓			✓	✓			✓	✓	✓	✓	✓	✓	✓
VCL	✓	✓			✓				✓	✓	✓	✓	✓	✓	✓
simdpp	✓	✓			✓	✓		✓	✓	✓	✓	✓	✓	✓	
T-SIMD	✓	✓				✓			✓	✓	✓		✓		
Vc [100]	✓	✓								✓	✓	✓	✓	✓	✓
xsimd	✓	✓									✓	✓	✓	✓	✓
Boost.SIMD	✓								✓	✓	✓	✓	✓	✓	✓
Highway	✓	✓			✓	✓	∼	✓	✓	✓	✓	✓	✓	✓	
SIMDWrappers	✓	✓	✓	✓	✓	✓	✓				✓		✓		✓

Table 2.3: Comparison of SIMD libraries.

2.4.3 Instruction emulation

The main goal of any SIMD library is to abstract the architecture-specific instructions. One way to achieve it is by restricting the available instructions to the minimal set of instructions common to all architectures. However some instructions can be very useful and we may want to use them if they are available, even if it means paying a small performance penalty on architectures where the instruction is not available. A good example of such instruction is the `compressstoreu` instruction available in the AVX-512 instruction set. This instruction allows, based on a given mask, some elements of a register to be packed to its left and stored in memory. It is used by the `copy_back` function of SOACollections. Figure 2.13 shows an example of a compression operation on AVX-512 and its emulation on AVX2 hardware. As this instruction is not available in older instruction sets, it can be emulated using a lookup table and a permute instruction followed by a regular store [2, 111]. Because this instruction is used a lot, usually multiple times in a row with the same mask, the compiler is able to optimize the code to perform the lookup once and the core can pipeline the independent permute and store instructions, making it efficient.

2.5 Conclusion

Modern architectures are becoming more and more parallel. Taking advantage of all the computing power available requires a software development paradigm shift. Using multiple threads requires careful data management to avoid communication bottlenecks. Exploiting SIMD capabilities of modern CPUs efficiently requires the developer to change the data layout and algorithm. To help in this process, new DSLs and SIMD libraries are created. This chapter introduces SIMDWrappers, a new library integrated in the LHCb framework, that allows to simplify the development of SIMD algorithms.

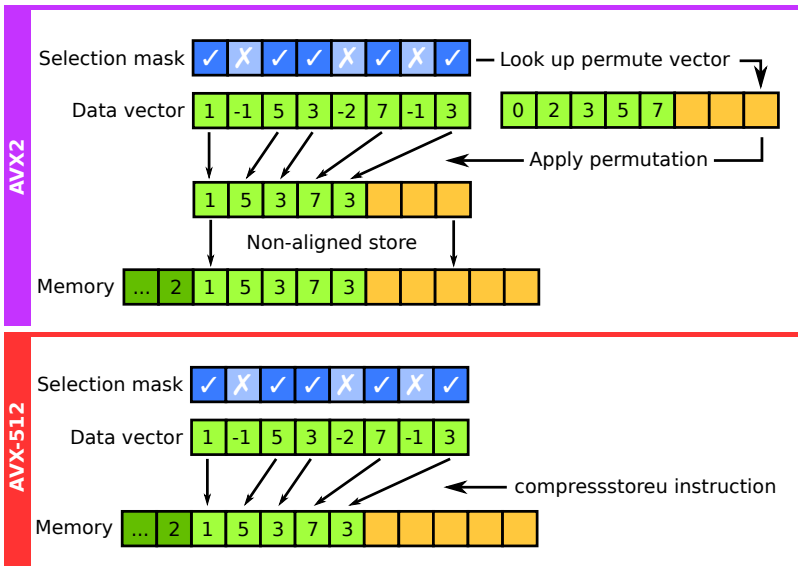


Figure 2.13: Emulation of AVX-512's compressstoreu instruction on AVX2 capable architecture.

Chapter 3

Parallelism on GPU

Contents

3.1 Introduction

Similarly to the previous chapter, this chapter reviews the evolution of GPU architectures and their increasing use in High Performance Computing (HPC). The CUDA programming model is presented with some elements of GPU architectures. Finally, three common optimization patterns are presented: grid-stride loops, shared memory caching and warp-level programming.

3.2 From arcade video games to HPC

Specialized graphics circuits appeared as early as 1970 with arcade system boards. In early video game hardware, the memory for frame buffers was expensive and dedicated video chips were required to offload the data composition tasks from the CPU. In 1980, NEC introduced the first implementation of a PC graphics display processor on an integrated circuit, the NEC μPD7220, which was capable of drawing lines, circles, arcs and character graphics to a bit-mapped display. This chip enabled the design of low-cost, high-performance video graphics cards, laying out the foundation of the PC graphics market. In 1988, the first dedicated polygonal 3D graphics boards were introduced in arcades with the Namco System 21 and Taito Air System. In the 1990s the PC market continued to grow with an increasing demand for 2D graphical user interface (GUI) accelerators like the S3 86C911 from S3 Graphics. As real-time 3D graphics continued to develop in the arcade, computer

and home console games, the demand for hardware-accelerated 3D graphics grew.

The term Graphics Processing Unit (GPU) appeared in 1994 with the 32-bit Sony GPU of the PlayStation video game console. The first transformation and lighting accelerator for home video game consoles was the Nintendo 64's Reality Coprocessor, released in 1996, followed the next year by the Fujitsu Pinolite, a 3D geometry processor for PC that could be used to accelerate any kind of 3D processing workload; and the 3Dpro/2MP of Mitsubishi, a fully-featured GPU capable of transform and lighting, which was used by ATi in their FireGL 4000 graphics card. At this time, performance 3D graphics were only possible with discrete boards dedicated to 3D function accelerations and lacking 2D GUI support, such as the PowerVR and the 3dfx Voodoo. With the improvements of manufacturing technologies, chips integrating video, 2D GUI and 3D functions started to appear. First failed attempts at low cost 3D graphics chips were previous generation 2D accelerators with 3D features extensions, such as the S3 ViRGE, ATi Rage, and Matrox Mystique. But the first to really succeed were Rendition's Vérité chipsets. 1999 marked the original release of the NVIDIA GeForce 256, which brought hardware accelerated transform and lighting to the consumer market.

In 2000, NVIDIA introduced the first GPU chip capable of programmable shading: the GeForce 3. Instead of following a fixed pipeline, each pixel could now be processed by a short program, called a shader, that could perform custom texturing and lighting computations. Similarly, each vertex could be processed by a shader that could control the way it was projected onscreen. The introduction of the ATi Radeon 9700, in 2002, added more control flow and more complex floating point mathematical operations to the pixel and vertex shaders, which soon became as flexible as CPUs. The NVIDIA GeForce 8 series unified vertex and pixel shaders provided them with the same capabilities. This unification is known as the unified shader model. As GPUs became capable of more general computation, they started to be used for other workloads than graphics. This subfield of research was called General Purpose Computing on GPU (GPGPU), and started by hacking texture memory as a way to provide data and pixel shaders to perform arbitrary computations. In 2006, NVIDIA announced the Compute Unified Device Architecture (CUDA) [130], the first successful programming model for GPU computing. Other programming models like OpenCL [156] or SYCL [95] followed, but to this day CUDA remains the most popular way to do general purpose computing on NVIDIA GPUs. CUDA will be presented in the next section.

In 2010, NVIDIA partnered with Audi to include GPUs in car dashboards, improving the navigation and entertainment systems. These embedded GPUs are part of the Tegra family System on a Chip (SoC) which includes an ARM CPU, an NVIDIA GPU and a memory controller in one package. Acquired in 2006 by AMD, ATI products were rebranded with the release of AMD's Radeon HD 6000 Series cards in 2010 and 2011. In 2012, the NVIDIA Kepler architecture introduced the GPU boost feature, enabling GPUs to adjust their frequency depending on the load and the power available. In the following years, both AMD and NVIDIA continued to improve their GPUs by using more and more efficient manufacturing processes. In 2017, NVIDIA released the Volta architecture, dedicated to the datacenter mar-

ket and aiming at optimising machine learning tasks. It included additional CUDA cores, HBM2 memory and new tensor cores specially designed to accelerate tensor multiplication operations for deep learning. The next year, the Turing architecture was announced, with the addition of ray tracing cores that could be used by the rendering pipelines to compute realistic reflections and shading effects. It also added tensor cores to consumer GPUs, as they can be useful for denoising ray traced renders using deep learning.

3.3 CUDA programming model

Introduced in 2006, the Compute Unified Device Architecture (CUDA) is a parallel programming model designed to develop general purpose applications on GPUs that scales transparently with the number of processor cores. The C++ version of CUDA is presented as a language extension and allows the programmer to use a single source for both the host code running on the CPU and the device code running on the GPU. The device code is organised in kernels. Each kernel is an entry point function that will be executed by a thread on the GPU. Multiple threads will execute the same kernel in parallel. This execution model is called Single Instruction Multiple Threads (SIMT). Each thread has access to its own set of registers and an allocated local memory. Threads executing the same kernel are organised into blocks, which themselves are organised into grids. These two levels correspond to different communication abilities and shared memory. Threads within the same blocks can communicate through the use of shared memory, while threads within the same grid only share the view of the global memory. Like the cache hierarchy on a CPU, memory levels on a GPU vary in speed and capacity. The register file is the fastest kind of memory on the GPU, but is also small and must be partitioned between all threads. Shared memory is partitioned to match the number of blocks, exposing a larger capacity than registers, but is also slower. Global memory is usually outside the GPU chip and is both the largest memory available and the slowest. Local memory is implemented using global memory, but can be configured to be cached through two cache levels to improve access speed. Figure 3.1 illustrate the thread organisation and memory hierarchy in CUDA.

To understand CUDA's organisation, it is useful to present the underlying GPU architecture. The NVIDIA GPU architecture is an array of multi-threaded Streaming Multiprocessors (SMs). When a kernel is launched, the blocks of the grid are distributed to multiprocessors with available execution capacity. The threads of a block execute concurrently on the same SM, and an SM can execute multiple blocks. The SM manages threads by groups of 32, called warps. Each thread composing a warp starts at the same program address, but as the Volta architecture introduces independent thread scheduling, all threads have their own program counter, and are thus free to branch and execute independently. A warp executes only one common instruction at each cycle. To achieve full efficiency, all 32 threads must agree on their execution path. If some threads diverge due to a data-dependent conditional branch, the warp executes each path taken by disabling threads from other paths. Since Volta, different paths are interleaved at the instruction level to allow fine-grained synchronization of threads following different paths, as shown in Figure 3.2. Threads from different warps execute independently regardless of the path they are

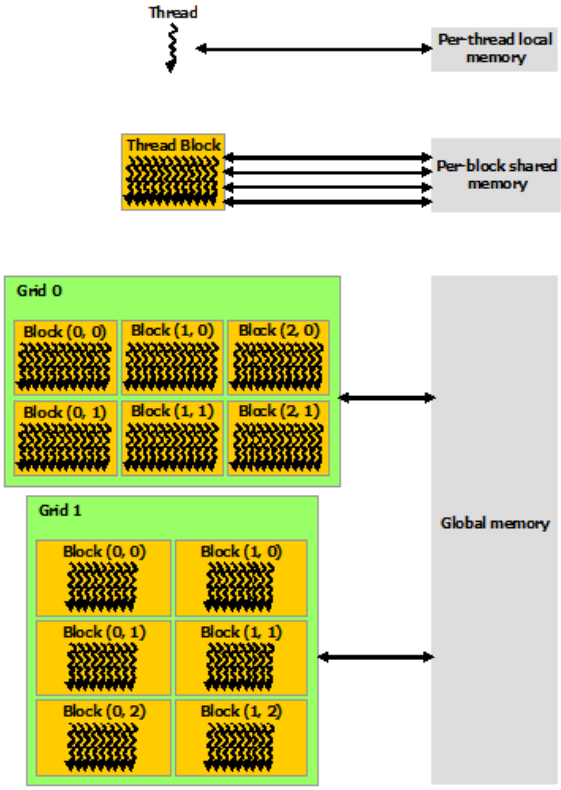


Figure 3.1: CUDA Thread and Memory Hierarchy [130].

taking. The execution context of each warp processed by an SM is stored on-chip during the entire lifetime of a warp. This has the benefit of offering zero-cost context switching, but necessitates the partitioning the register file among the warps. The number of blocks and warps that can reside on a single SM is therefore bounded by the amount of registers and shared memory used by the kernel and by the SM capacity.

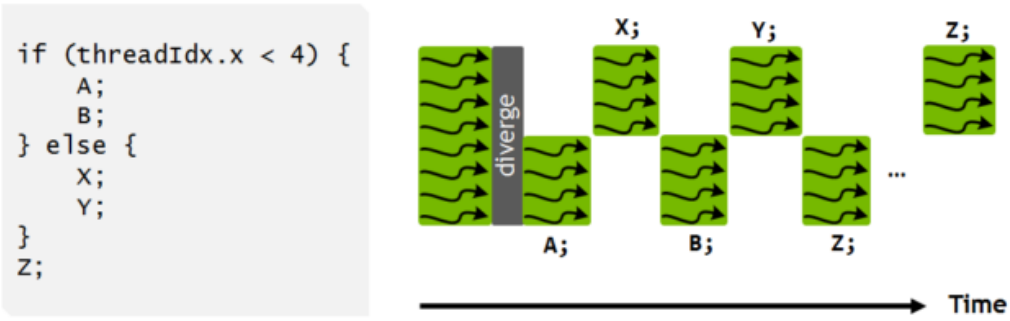


Figure 3.2: Interleaved execution of the two paths resulting from a divergent branch.

The CUDA programming language is an extension of C++. Kernels are defined as C++ functions prefixed with the `__global__` keyword. Inside a kernel, globally defined variables can be accessed, like the index of the thread executing the kernel `threadIdx`, the block dimension, or number of threads in the block, `blockDim` and the block index `blockIdx`. These three variables are represented as

3-components vectors, simplifying their use for writing kernels operating on vectors, matrices or tensors. Listing 2 shows a possible CUDA implementation of the single precision A × X Plus Y (saxpy), a standard function of the Basic Linear Algebra Subroutines (BLAS) library. The saxpy function takes as input two vectors of size n, X and Y, and a scalar A, then computes for each element the expression $A \times X_i + Y_i$, and finally stores the result in Y. The implementation assigns one thread to compute each element of the vector. This requires having more threads than the number of elements to process, ideally the same number, in order to avoid wasting threads. In the device code, threads start by computing the index of the elements they have to process (line 3). They then need to test that this index is valid, i.e. smaller than the number of elements in the vector (line 4). Finally they can perform the computation (line 5). In the host code, the kernel is launched using the CUDA-specific syntax `kernel<<<numberOfBlocksInTheGrid, numberOfThreadsInEachBlock>>>(arguments, ...)`. In the example, the `saxpy` kernel is launched with 4 096 blocks of 256 threads each, for a total of 1 048 576 threads (line 20). The constants `n` and `A` can be transmitted directly to the device during the kernel call, but the X and Y vectors must be valid pointers in the device's global memory, allocated using CUDA-specific API calls (lines 15 and 16).

```
1  // Kernel definition (device code)
2  __global__ void saxpy (int n, float A, float *X, float *Y) {
3      int i = blockIdx.x * blockDim.x + threadIdx.x;
4      if (i < n) {
5          Y[i] = A * X[i] + Y[i];
6      }
7  }
8
9  // Host code:
10 int main () {
11     // ...
12     int N = 1000000;
13
14     // Fill the device memory by copying data from the host
15     cudaMemcpy(deviceX, hostX, N, cudaMemcpyHostToDevice);
16     cudaMemcpy(deviceY, hostY, N, cudaMemcpyHostToDevice);
17
18     // Kernel invocation with 4096 blocks of 256 threads each
19     // totaling to 1 048 576 launched threads
20     saxpy<<<4096, 256>>>(N, 2.f, deviceX, deviceY);
21
22     // Transfer the result back to host memory
23     cudaMemcpy(hostY, deviceY, N, cudaMemcpyDeviceToHost);
24     // ...
25 }
```

Listing 2: saxpy implementation in CUDA, adapted from [68]

3.4 Grid-stride loops

As presented in the previous section, kernels like saxpy are executed efficiently when the number of threads used matches the size of the data. If fewer threads are used the kernel, as written in the example, will fail to produce the correct result for the last elements of the vector. If more threads are used, they will not take the branch and waste compute power by not contributing to the result. Such a kernel is called monolithic, because it assumes a single large grid of threads to process the entire array in one pass. But it is not always possible to know in advance what the size of the data will be, and the number of elements might exceed the maximum allowed number of threads in a grid. An alternative way of writing the same kernel is to reintroduce a loop inside the kernel:

```
1  __global__ void saxpy (int n, float A, float *X, float *Y) {
2      int threadIndex = blockIdx.x * blockDim.x + threadIdx.x;
3      int stride = blockDim.x * gridDim.x;
4      for (int i = threadIndex; i < n; i += stride) {
5          Y[i] = A * X[i] + Y[i];
6      }
7  }
```

This pattern was described and named "Grid-stride loop" by Mark Harris in a 2013 NVIDIA devblog post [65]. Instead of assuming that the grid is large enough to assign one thread per element, this kernel loops over the data one grid-size at a time. Each iteration processes `blockDim.x * gridDim.x` elements, as many as there are threads in the grid. Each thread of the block processes contiguous elements to keep the maximum memory coalescing of the monolithic version. If called with a large enough grid, the grid-stride loop kernel adds no overhead to the monolithic version, as the for loop branch is replacing the if branch, but it has several advantages.

The first and most important benefit of this pattern is its scalability and thread reuse. Creating or destroying a thread and finding an SM to launch it on has a small overhead that is amortised when the thread is reused for multiple loop iterations. Furthermore, data can be persisted in registers from one iteration to another, allowing optimizations which are only possible in sequential programs. An example of such optimization will be given in Section 4.3. By limiting the number of blocks, the occupancy and the performance can be tuned, as having a smaller number of threads and blocks allows more registers and shared memory for them.

The second benefit of this technique is the ability to launch a kernel with only one thread for debugging purposes, which makes the execution fully serial instead of parallel, and easier to visualize.

The last benefit is the portability of the code due to its similarity with more familiar sequential programs. Some libraries, like Hemi [69] or LHCb's Allen [7], take advantage of this feature to compile functions either as a CUDA kernel for the GPU or as a sequential loop that can be executed on the host CPU.

3.5 Shared memory optimisations

When threads of a warp are accessing global memory, the GPU tries to group the individual accesses into a minimum number of transactions. As the transactions occur off-chip they are slower than on-chip accesses. In order to maximize the bandwidth, each transaction must utilize the full width of the memory bus, which can only happen when the requested addresses are contiguous in memory. When the addresses are too far apart, the GPU can no longer group them and the bandwidth falls quickly. This effect can be shown in the first plot of figure 3.3, where the bandwidth is measured as a function of the distance between each addresses, on a fairly recent RTX 2070. When the distance between elements is 1, the bandwidth is maximal, but degrades exponentially when the distance increases, until a distance of 32 when the worst case of one access per transaction is reached. Unfortunately, strided accesses to global memory are necessary for many applications, such as accessing elements of a multidimensional array.

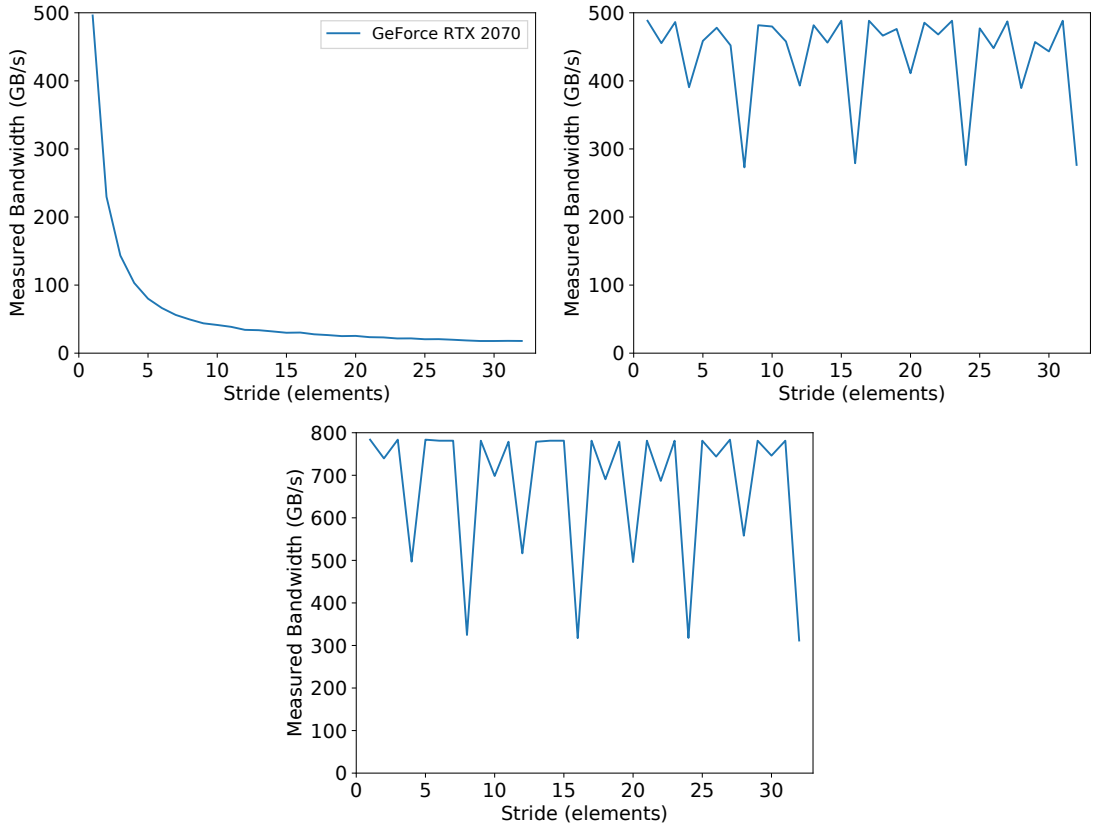


Figure 3.3: Impact of strided global memory accesses on effective bandwidth (top-left), impact of bank conflicts on shared memory accesses, after an initial copy from global memory to shared memory (top-right), and without the initial copy (bottom).

To avoid this issue, a common technique consists in first copying the data to the shared memory using coalesced accesses and then doing the strided access in shared memory, where bandwidth is higher and latency much lower [66]. Shared memory is allocated per block, so all threads in the same block have access to the same shared memory. Threads can access data in shared memory loaded from global

memory by other threads within the same block. Shared memory is the primary means of communication within a block, and can be used as a user managed cache or to enable cooperative parallel algorithms. In CUDA, a variable can be allocated in shared memory by prefixing its declaration by the `__shared__` keyword.

As always when dealing with communication between concurrent threads, care must be taken to avoid race conditions. A block can contain threads of different warps that execute at the same time. There is therefore no guarantee that two different threads executing the same kernel in parallel reach the same instruction simultaneously. If one thread expects to read some data stored by another executing the same kernel, they must first be synchronized. Otherwise a race condition can occur, which leads to undefined behavior and incorrect results. CUDA provides a synchronization barrier primitive: `__syncthread()` which makes the threads wait until all threads of the block have reached the barrier. By synchronizing threads between a store and a load access, the shared memory can be accessed safely.

Figure 3.3 shows the impact of shared memory used to cache a strided access of global memory. The bandwidth no longer reaches the worst case exponentially, but some spikes can be noticed for strides that are powers of two and multiples of powers of two. These spikes are due to the hardware implementation of shared memory. In order to achieve high memory bandwidth for concurrent accesses, even when they are far apart, shared memory is divided into equally sized memory banks that can be accessed simultaneously. The shared memory addresses are mapped onto memory banks so that successive 32-bit words are assigned to successive banks. Each bank can deliver one 32-bit word each cycle. If multiple threads are requesting the same address, they are all accessing the same element of a single bank and can be served in one cycle, the data is broadcast. But if multiple threads request multiple addresses from the same bank, a bank conflict occurs and the accesses are serialized. Modern GPUs have 32 banks to match the 32 threads of their warps. The maximum bandwidth of shared memory is reached when all threads are accessing different banks or the same elements within each bank. Listing 3 shows the kernels that were used to generate the plots of Figure 3.3.

3.6 Warp-level programming

Warps are an essential part of the CUDA execution model as they enable it to reach high-performances, yet they are often implicitly used. Many kernels could take advantage of explicit warp-level programming to reach even higher performances. Non-trivial parallel programs use collective communication operations, such as parallel reductions and scans. Warp primitives are the basic building blocks of collective communications within a warp.

All synchronized data-exchange primitives are suffixed with `_sync` and take a bit-mask as their first argument. Each bit of the mask corresponds to one of the 32 threads in the warp and the mask defines which thread should participate in the exchange. Only the threads that are participating are synchronized, other threads can still diverge.

```
// Kernel for measuring bandwidth of global memory accesses
// as a function of the stride
__global__ void strideGlobal (float *a, int stride) {
    int i = (blockDim.x * blockIdx.x + threadIdx.x) * stride;
    a[i] = a[i] + 1;
}

// Kernel for measuring the impact of bank conflicts on shared
// memory accesses as a function of the stride
__global__ void strideShared (float *a, int stride) {
    __shared__ float s[256];
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    s[threadIdx.x] = a[i];
    __syncthreads();
    i = (threadIdx.x * stride) % 256;
    s[i] = s[i] + 1;
}
```

Listing 3: Benchmark kernels to analyse the effect of memory accesses stride on bandwidth, adapted from [67]

```
int __all_sync(unsigned mask, int predicate);
```

Returns true if and only if the predicate argument is true for all threads participating in the exchange.

```
int __any_sync(unsigned mask, int predicate);
```

Returns true if and only if the predicate argument is true for any thread participating in the exchange.

```
unsigned __ballot_sync(unsigned mask, int predicate);
```

Returns a bit-mask where the Nth bit is set if the predicate is true for the Nth bit participating in the exchange.

```
unsigned int __match_any_sync(unsigned mask, T value);
```

Returns a bit-mask representing the set of threads that have the same value of `value`. Threads with different values receive a different bit-mask.

```
unsigned int __match_all_sync(unsigned mask, T value, int *pred);
```

Returns `mask` if all threads in `mask` have the same value or 0 otherwise. Predicate `pred` is set to true if all threads in `mask` have the same value of `value` or false otherwise.

The next set of warp primitives is the family of shuffle functions. They are designed for direct variable exchange between thread registers within a warp and are templated to work on any numeric primitive type: `int`, `unsigned`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double`, `half`, `half2`. Threads may only read variables from other threads participating in the exchange, if the source thread is inactive, the result is undefined.

```
T __shfl_sync(unsigned mask, T var, int srcLane, int width=32);
```

Returns the variable `var` from the thread specified by `srcLane`.

```
T __shfl_up_sync(unsigned mask, T var, unsigned delta, int width=32);
```

If the current thread is in lane `X`, returns the variable `var` from the thread in lane `X − delta`.

```
T __shfl_down_sync(unsigned mask, T var, unsigned delta, int width=32);
```

If the current thread is in lane `X`, returns the variable `var` from the thread in lane `X + delta`.

```
T __shfl_xor_sync(unsigned mask, T var, int laneMask, int width=32);
```

If the current thread is in lane `X`, returns the variable `var` from the thread in lane `X ⊕ laneMask`.

Using shuffle functions, parallel tree-reduction can be implemented efficiently [122]. Here is an example of such an algorithm when all 32 threads are participating:

```
constexpr unsigned FULL_MASK = 0xffffffff;
for (int offset = 16; offset > 0; offset >>= 1) {
    val += __shfl_down_sync(FULL_MASK, val, offset);
}
```

At the end of the loop, the `val` variable of the first thread contains the reduced sum of the `val` of all threads. The algorithm executes efficiently in 5 iterations. Figure 3.4 illustrates the execution of this algorithm.

With the Ampere architecture, new primitives for parallel reductions were added to CUDA: `__reduce_add_sync`, `__reduce_min_sync`, `__reduce_max_sync`, `__reduce_and_sync`, `__reduce_or_sync` and `__reduce_xor_sync`.

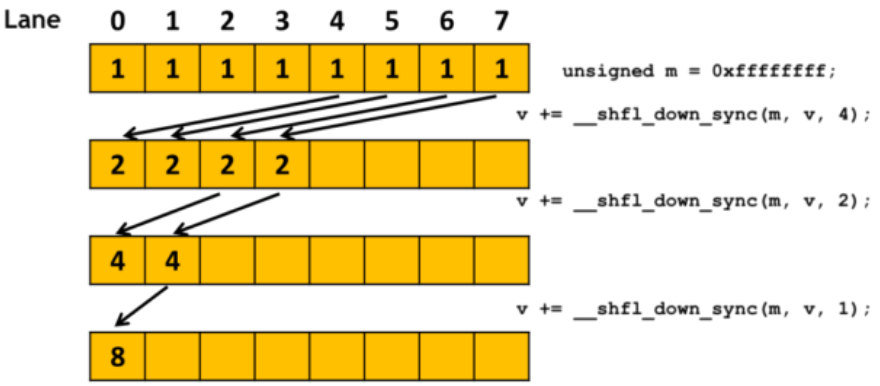


Figure 3.4: Illustration of the last operations of warp-level parallel tree-reduction [122].

3.7 Conclusion

GPUs are getting bigger and bigger. With more than 10000 cores, new problems arise. To continue to exploit this massive parallelism, algorithms must find new ways to synchronize their threads to avoid contention and use the memory bandwidth efficiently to continuously feed the compute cores. Techniques from SIMD programming can be applied to GPUs' warps using warp-level intrinsics. The techniques presented in this chapter will be applied in the next one.

Chapter 4

Connected Component Analysis

Contents

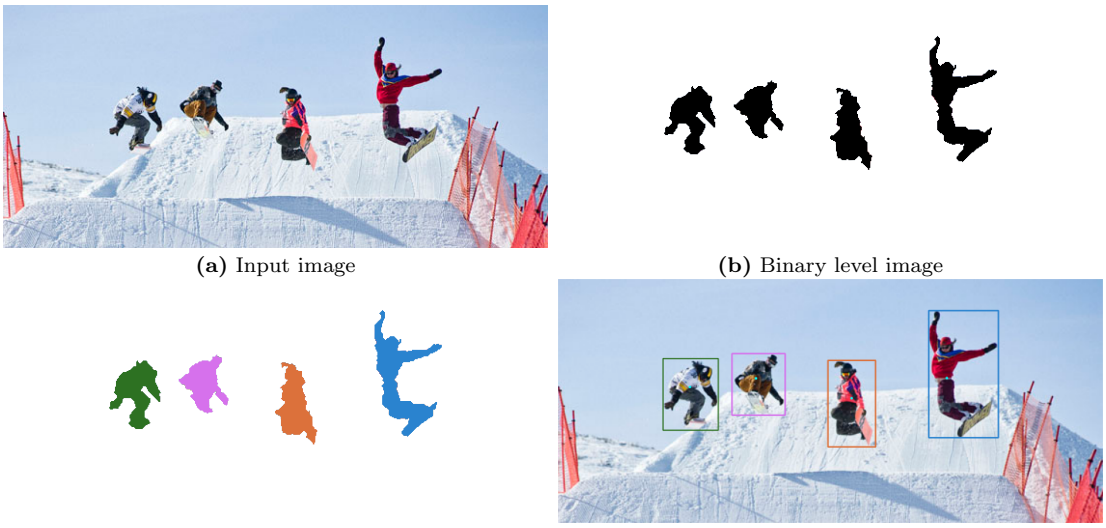
4.1 Introduction

This chapter presents the Connected Component Analysis problem and multiple algorithms, developed in the context of this thesis, to address it efficiently. The first algorithm, HA4, was developed at the very start of this thesis and improved the state-of-the-art in connected component labeling on GPUs, as well as being the first efficient implementation of connected component analysis on GPUs. The second algorithm is a GPU port of the FLSL SIMD CPU algorithm, inspired by the LSL algorithm. FLSL on GPUs improved upon HA4 by reducing the memory accesses conflicts that are especially presents on new hardware with a lot of cores. Along with FLSL, two other optimisations aimed at further reducing conflicts are presented and evaluated. On CPU, two new algorithms were made for this thesis. The first one is a modification of the classic Rosenfeld algorithm, which is presented at the start of this chapter, to use SIMD. The second one is a new algorithm, named SparseCCL, which takes advantage of the sparsity of the input images. Through these algorithms, an evaluation of CPU and GPU architectures is made and optimisation techniques taking advantage of the architecture are presented.

4.2 Connected Component Labeling and Analysis

Connected Component Labeling (CCL) is a crucial part of Computer Vision and is as old as the field [148, 161, 63]. A connected component is defined as a set of pixels for which it exists a relation of connectedness, i.e. for all pair of pixels of the connected component, there is a path within the connected component that link the two pixels [41]. CCL can operate on any graph topology, but the algorithms presented in this chapter focuses on 2D images with square pixels, connected in a square mesh of 4 or 8 neighbours, that will be referred as 4- and 8-connected. Many applications using CCL require the computing of some features for each connected component like its bounding box, its surface or its centroid. This can be used directly by the application or just used to filter out small connected components. This evolution of CCL algorithms is called *Connected Component Analysis* (CCA). CCA is used by many medical applications [43, 129, 10, 123, 97], surveillance [90, 151, 125, 104, 105, 50], autonomous driving [163, 52] and other Computer Vision applications [62, 159]. Figure 4.1 shows a typical computer vision pipeline featuring CCL and CCA.

CCL on CPUs has been heavily studied and optimized [60, 73, 25, 111, 103]. Early GPU CCL algorithms were iteratives [168, 20, 88]. The first direct CCL algorithm for GPUs was introduced by Komura [99] and improved by Playne [139] by



(a) Input image

(b) Binary level image

(c) Connected Component Labeling (CCL)

(d) Connected Component Analysis (CCA)

Figure 4.1: Example of a typical computer vision processing chain: starting with an input image (a) that is then turned into binary, for instance using a motion detection algorithm (b), then a CCL algorithm extracts connected components (c) and CCA is performed to extract features, like the bounding rectangles (d).

limiting the number of unions performed.

On the other hand, parallelization of CCA is much harder as it must perform many parallel reductions of many distinct pixel sets. It is a voting algorithm [160] just like histogram computation or Hough transform [86]. This issue arises from the serialization of memory accesses and is amplified by the high number of cores of the GPUs. Therefore, while there are many hardware algorithms [98, 167, 157, 155], there are only a few algorithms for multi-core CPUs [128, 32] and GPUs [146, 76]. The next sections will describe the main classes of CCL algorithms.

4.2.1 One component at a time

In this first class of algorithm, connected components are processed one at a time. The image is scanned one time and, for every foreground pixel encountered, a traversal of the connected component is done to label all the pixels. This algorithm and its variants are often called *flood fill* or sometimes *seed fill*. The traversal can be done using a stack in depth-first order, or a queue in breadth-first order. Implementations of algorithms of this class are found in [162] and [9]. This algorithm can be optimized by only adding, on top of the stack, the branching pixels – ie. the pixels that have more than one non-visited neighbour – and directly processing the others. Doing so, we avoid a store and a load for these pixels. If the image is sparse and if the algorithm has a list of pixel coordinates, it can directly start at known pixel positions, avoiding the read of many background pixels. However, this does not prevent the test of every pixel on the contour of the connected component and it adds the cost of removing pixels from the list. An implementation of this type of algorithm, specialized for the LHCb experiment, is described in [24]. Contour Tracing algorithms [40, 152] can also be classified as one component at a time. They

are smart algorithms which can find the contour of a connected component without visiting every pixels, but they are not cache aware and therefore inefficient.

4.2.2 Multi-pass iterative algorithms

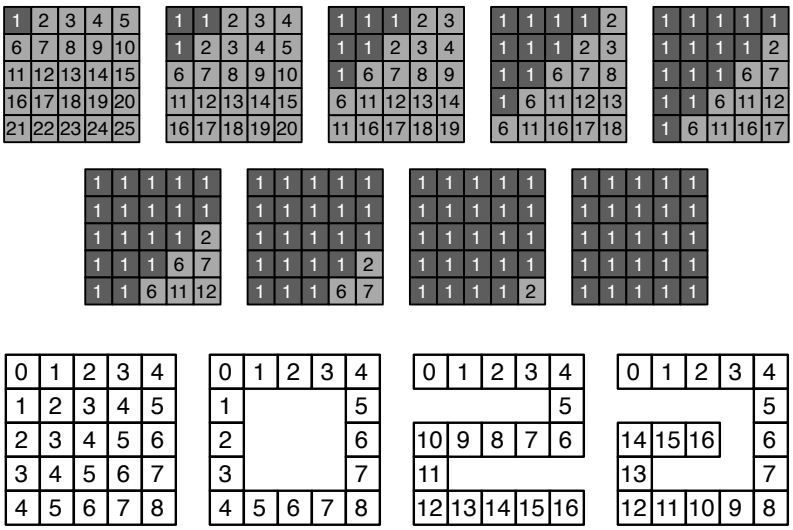


Figure 4.2: The number of iteration depends on the data structure: 9 iterations for a 5×5 square (top), 16 iterations for a zig-zag or a spiral (bottom).

Multi-pass iterative algorithms were introduced by Haralick [63]. Each pixel is initialized with a unique temporary label; this label is then propagated to the pixel's neighbors using local minimum or maximum propagation. The propagation step is repeated until the image of labels reaches stabilization, i.e. there is no more change within the image. This algorithm was particularly fitted for implementations on parallel architectures [103, 50], due to its high regularity, before the appearance of fast scatter and gather operations needed for direct two-pass algorithms. The number of iterations, and thus the processing time, of iterative algorithms depends on the longest path in the image, called the maximum geodesic distance. For an $n \times n$ spiral, the number of iterations is equal to $\frac{n^2}{2}$. Figure 4.2 gives examples of some connected components structures.

4.2.3 Direct two-pass algorithms

The two-pass CCL algorithms are split into three steps and perform two image scans (like the pioneering algorithm of Rosenfeld [148]). The first scan (or first labeling) assigns a temporary label to each connected component and some label equivalences are built if needed, using a union-find equivalence table T [132]. The second step solves the equivalence table by computing the transitive closure of the graphs associated with the label equivalences. The third step performs a second scan (or second labeling) that replaces the temporary label of each connected component with its

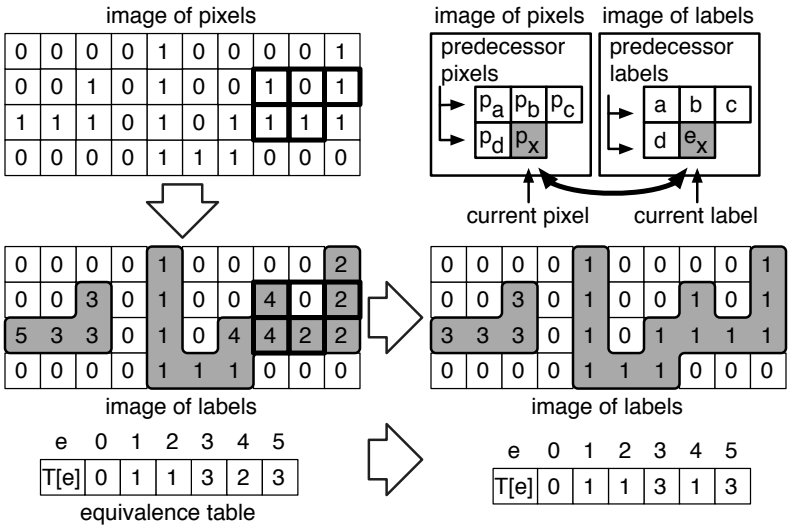


Figure 4.3: Example of 8-connected CCL with Rosenfeld algorithm: binary image (top), image of temporary labels (bottom left), image of final labels (bottom right) after the transitive closure of the equivalence table.

final label, by doing a simple lookup: $I(i,j) \leftarrow T[I(i,j)]$.

Figure 4.3 defines some notations and gives an example of a classic Rosenfeld algorithm execution. The current pixel is noted p_x and its label e_x. The neighbor pixels are noted p_a, p_b, p_c, p_d, and their associated labels a, b, c, d. T is the equivalence table, e is a label and r its root. The first scan of Rosenfeld is described in algorithm 1, the transitive closure in Algorithm 2, while the classical union-find algorithms are provided in Algorithms 3 and 4. In the example of Figure 4.3, the rightmost CC requires three labels 1, 2 and 4. When the mask is in the position seen in the figure in bold outline, the equivalence between 2 and 4 is detected and stored in the equivalence table T. At the end of first scan, the equivalence table is complete and applied to the image.

Decision Tree (DT) based algorithms for CCL have been proved to be very efficient in enhancing scalar implementations [166]. The DT reduces the number of Union and Find function calls to the strict minimum, i.e. when there is an equivalence between two different labels. There are only two patterns generating an equivalence between labels: stairs and concavities, which are depicted in figure 4.4. A DT is more efficient than path-compression as it reduces the number of memory accesses. Considering the classical implementation of the Rosenfeld algorithm (Algorithm 1), there are four calls to `find` and one to `union`. The calls to find in the union can be omitted because the input labels are already equivalence trees' roots. When using a DT (Algorithm 5), there are at most one call to `union` and two calls to `find` as we have at most one equivalence between labels. Figure 4.5 shows the decision tree for the Rosenfeld algorithm.

Algorithm 1: Rosenfeld algorithm – first labeling (step 1)

Input: a, b, c, d, four labels, p_x, the current pixel in (i, j)

if $p_x \neq 0$ **then**
  $a \leftarrow E[i-1][j-1], \quad b \leftarrow E[i-1][j]$
  $c \leftarrow E[i-1][j+1], \quad d \leftarrow E[i][j-1]$
  if $(a = b = c = d = 0)$ **then**
    $ne \leftarrow ne + 1, \quad e_x \leftarrow ne$
  else
    $r_a \leftarrow find(T, a), \quad r_b \leftarrow find(T, b)$
    $r_c \leftarrow find(T, c), \quad r_d \leftarrow find(T, d)$
    $e_x \leftarrow min^+(r_a, r_b, r_c, r_d)$
    if $(r_a \neq 0$ **and** $r_a \neq e_x)$ **then** $union(T, e_x, r_a)$
    if $(r_b \neq 0$ **and** $r_b \neq e_x)$ **then** $union(T, e_x, r_b)$
    if $(r_c \neq 0$ **and** $r_c \neq e_x)$ **then** $union(T, e_x, r_c)$
    if $(r_d \neq 0$ **and** $r_d \neq e_x)$ **then** $union(T, e_x, r_d)$
else
  $e_x \leftarrow 0$

Algorithm 2: Sequential solve of equivalences (step 2)

for $e \in [1 : n]$ **do**
  $T[e] \leftarrow T[T[e]]$

Algorithm 3: find(T, e)

Input: e a label, T an equivalence table
Result: r, the root of e

$r \leftarrow e$
while $T[r] \neq r$ **do**
  $r \leftarrow T[r]$
return r

Algorithm 4: union(T, e_1, e_2)

Input: e_1, e_2 two labels, T an equivalence table
Result: e, the least common ancestor of the e's

if $e_1 < e_2$ **then**
  $e \leftarrow e_1$, $T[e_2] \leftarrow e$
else
  $e \leftarrow e_2$, $T[e_1] \leftarrow e$
return e

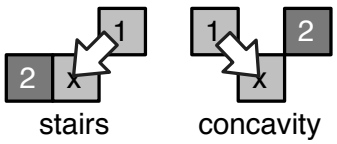


Figure 4.4: 8-connected Basic patterns generating an equivalence: stairs & concavity.

4.2.4 Mask topology: blocks and segments

All efficient modern CCL algorithms are two-pass algorithms. They differ on the optimisations strategies applied to minimize the number of temporary labels and

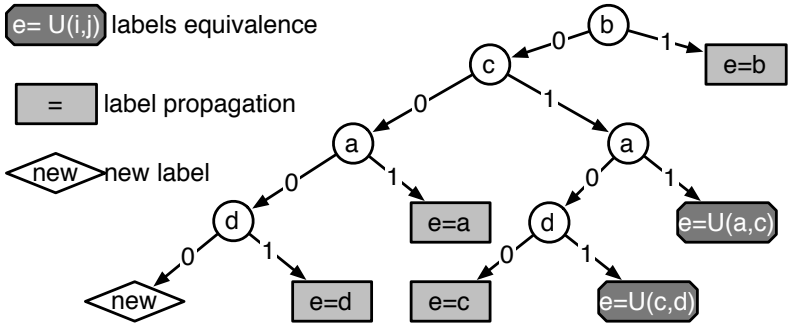


Figure 4.5: 8-connected Decision Tree for a 4-pixel mask. Labels equivalence (call to Union) in dark gray.

Algorithm 5: Rosenfeld with DT – optimized first labeling (step1)

```
   Input: a, b, c, d, four labels, p_x, the current pixel in (i, j)
 1 if p_x ≠ 0 then
 2     b ← E[i − 1][j]
 3     if (b ≠ 0) then
 4         e_x ← b
 5     else
 6         c ← E[i − 1][j + 1]
 7         if (c ≠ 0) then
 8             a ← E[i − 1][j − 1]
 9             if (a ≠ 0) then
10                 e_x ← U(a, c)
11             else
12                 d ← E[i][j − 1]
13                 if (a ≠ 0) then
14                     e_x ← U(c, d)
15                 else
16                     e_x ← c
17         else
18             a ← E[i − 1][j − 1]
19             if (a ≠ 0) then
20                 e_x ← a
21             else
22                 d ← E[i][j − 1]
23                 if (a ≠ 0) then
24                     e_x ← d
25                 else
26                     n_e ← n_e + 1
27                     e_x ← n_e
28 else
29     e_x ← 0
```

the way they manage equivalences. The mask topology is the access pattern around the pixels that are being processed. For the Rosenfeld algorithm presented in the previous section, one pixel is processed at a time and need to access the label of 4 of its neighbours. A mask topology is characterized by its load/store ratio, which is 4:1 for the Rosenfeld mask. Algorithms like RCM [82] and HCS_2 [72] reduce the number of loads that are necessary to compute a label by using alternative mask topologies with a ratio of 3:1 and 5:2, which are closer to 1.

As presented in the previous section, the execution time of a two-pass CCL al-

gorithm is correlated to the number of patterns that lead to the creation of a new temporary label. To reduce this number, and therefore improve the algorithm, a wider mask can be used. Algorithms like HCS_2 and Grana [57, 58] are block-based: they process 2 and 4 pixels from a 6-pixel and 16-pixel neighborhood. Like Rosenfeld algorithm's decision tree that decides whether or not to create a new label based on the presence of specific stair and concavity pattern, Grana mask can detect concavities that are small enough to fit in the mask and avoid temporary label creation by resolving them immediately.

While block-based algorithms can prevent some label creation from concavities, the only way to prevent those which arise from stairs is to use a run-based (also called segment-based) algorithm like HCS [71] and LSL [102, 32]. In a run-based algorithms, horizontal adjacency between pixels is detected before starting to assign labels to the runs. HCS is a "half" run-based algorithm which labels runs but manage the equivalences pixel by pixel. LSL is a "full" run-based algorithm as it manages equivalences between runs directly.

Figure 4.6 summarizes the different mask topologies presented. These algorithms were the subjects of multiple reviews [28, 29, 74] and are implemented in the open-source YACCLAB benchmarking framework [56, 25].

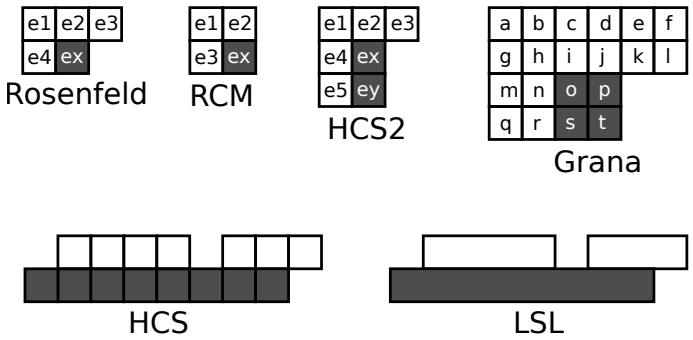


Figure 4.6: Mask topologies of Rosenfeld, RCM, HCS_2, Grana, HCS and LSL: input labels are in white boxes, output labels are in grey boxes.

4.3 HA4: Hybrid pixel/segment CCL for GPU

This section presents HA4, a new Hardware Accelerated 4-connected CCL / CCA algorithm, developed in 2018 [76] in the context of this thesis. It is based on a hybrid pixel / segment approach and relies on CUDA low-level intrinsic functions to be efficient. Unlike most existing GPU CCL algorithms, the image is not split into tiles but into horizontal multi-line strips, each strip being processed by a unique block and each line of the image by a unique warp. Each image segment is itself split into sub-segments of max length the size of a warp. The low-level intrinsics let each thread efficiently test if it is the start of a segment: only the start of a segment performs memory access to manage equivalences or features computations. The longer the segment the more it saves accesses.

The algorithm can be divided into three successive kernels that are presented in the following sub-sections:

- Strip labeling: we independently label horizontal strips of the image.
- Border merging: we check for label equivalences at the borders between strips.
- CCL / CCA: we perform a transitive closure of each pixel or compute some features for each label.

4.3.1 Strip labeling

The first step of the algorithm is to produce a partially labeled image. The input image I is divided into horizontal strips and each one is attributed to a block. In order to support any image width without having to increase the block size, we use the grid-stride loop design pattern [65]. Instead of assuming the block is large enough to process the entire strip, the kernel loops over the data one block size at a time. Because the same kernel processes the pixels of one strip, it can reuse past information about the continuity of the pixels, removing the need for the vertical border merging kernel. The loop also helps to amortise the threads creation and destruction by reusing them. The block width was set to the number of threads in a warp, which is 32 on current hardware, and the block height to 4, as it was found that this block size provides high occupancy and good performance.

Because each warp of the block processes consecutive pixels that are on the same line, it can use some warp-level primitives to optimize computations and memory accesses. A segment is defined as a consecutive set of non-zero pixels. By construction, a warp can contain up to 16 different segments. The start and the end of the segment are defined as its leftmost and rightmost pixels. Each thread of the warp is associated to one pixel of the image and can share the value of its corresponding pixel to all the other threads in the warp by using a `__ballot_sync` instruction. This instruction builds a 32-bit bitmask where the i^{th} bit is set if some predicate for the i^{th} thread of the warp is true. Here, the predicate is simply the boolean value of the thread's pixel.

Once the bitmask is known by all the threads, each thread can retrieve some information about its segment. Two distance operators are used: `start_distance` and `end_distance` (described in algorithm 6). For the start of the segment, `start_distance` is always equal to zero, while `end_distance` is always equal to the number of pixels in the segment. For each thread, `start_distance` gives the distance to the start of the segment. Figure 4.7 shows an example of both operators. The `__clz` (*Count Leading Zeros*) intrinsic returns the number of consecutive zeros starting from the most significant bit and going down inside a 32-bit register. The `__ffs` (*Find First Set*) intrinsic returns the position of the first bit set to one, starting from the least significant bit and going up inside a 32-bit register.

Since CUDA 9, all warp level primitives take a mask parameter that determines which threads are participating in the operation. This allows the threads to diverge and only synchronize if needed. The image width is assumed to be a multiple of the

warp size, and the mask is set to `ALL` = `0xFFFFFFFF`.

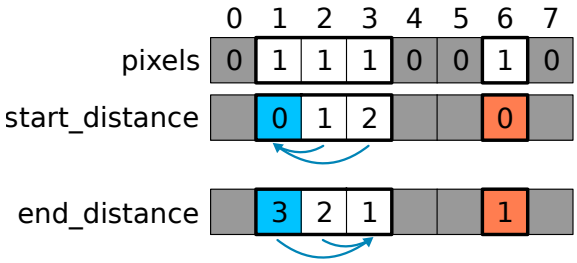


Figure 4.7: Distance operators on a 8-bit bitmask. Only set pixels are considered.

Algorithm 6: Distance operators for 32-bit bitmasks

```
operator start_distance(pixels, tx)
    return __clz(∼(pixels << (32−tx)))
operator end_distance(pixels, tx)
    return __ffs(∼(pixels >> (tx+1)))
```

For each block, the threads load their corresponding pixel from global memory, then build the bitmask and perform a segment start detection. The labels of the start pixels are initialized to their linear address $L[k_{y,x}] = k_{y,x}$. The other pixels are not initialized in order to reduce the amount of memory stores. For each line, the last segment start is tracked. If the first thread of the warp has a set pixel, the algorithm checks if it belongs to a longer segment and initializes it to its start address. After this first line labeling, the threads of the block are synchronized and the bitmask of pixels is retrieved from the warp above. This allows us to merge the lines within the strip. Each thread checks if its corresponding pixel in the current line or the line above is a segment start and, if it is, performs a union-find merge as described in algorithm 7. This merge function was first described by Playne and Hawick in [139] and is based on Komura's reduce function [99]. It works by finding the root of the two equivalence trees to which the labels belong to and writing the minimum root index to the root with the maximum index.

Algorithm 7: parallel merge(L, $label_1$, $label_2$)

```
while label1 ≠ label2 and label1 ≠ L[label1] do
    label1 ← L[label1]
while label1 ≠ label2 and label2 ≠ L[label2] do
    label2 ← L[label2]
while label1 ≠ label2 do
    if label1 < label2 then swap(label1, label2)
    label3 ← atomicMin(L[label1], label2)
    if label1 = label3 then label1 ← label2
    else label1 ← label3
```

The strip labeling is done in global memory. Because of the few memory stores performed, going to shared memory first for the label image L, like in previous

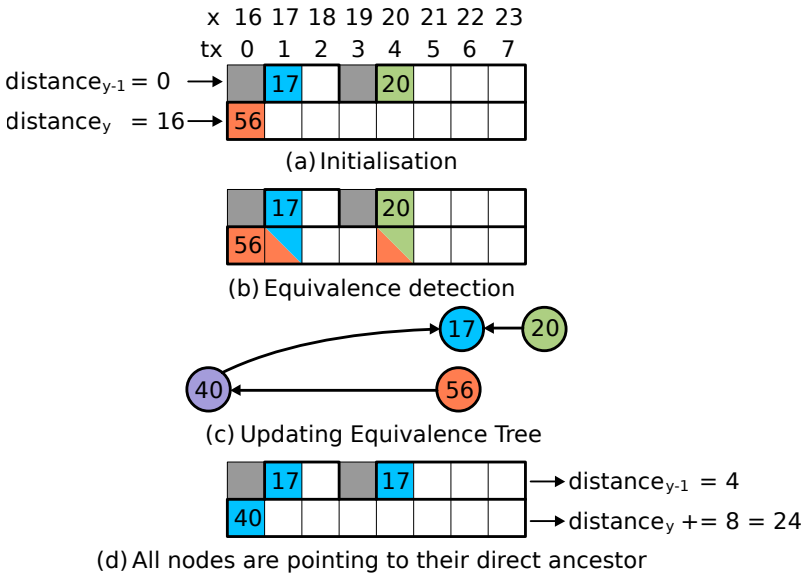


Figure 4.8: Example of a block labeling (image width = 40, block width = 8). **(a)** shows the initialization of the start pixels to their linear address. In **(b)** each thread detects the equivalences between segments of the two lines. The equivalence of node 56 to node 40 is detected because $\text{distance}_y \neq 0$ and 56−16=40. **(c)** shows the updated equivalence tree after the call of the merge function. Finally, **(d)** shows the final values of the start pixels and the updated values for the distances.

works [30][139], would be inefficient. Instead, shared memory is used to exchange bitmasks between warps.

As shared memory is organized in 32 banks, two threads willing to access different memory cells inside the same bank would result in a bank conflict, causing access serialization. For this reason, the algorithm exchanges the bitmasks instead of the pixels. This way, only the first thread of each warp would do a memory store, in a different bank for each line, and then in the next step, all threads from the same line would load from the same cell inside the same bank, resulting in a broadcast of the data. The entire strip-labeling kernel is described in algorithm 8 and an example is provided with figure 4.8.

4.3.2 Border Merging

Previous algorithms suffered from the non-coalesced access of the vertical border merging. Thanks to the strip division, only the horizontal borders have to be merged. As in the strip labeling, merge operations are performed only on the start of each segments, limiting the number of expensive global memory accesses and atomic operations.

The border merging described in algorithm 9 produces an equivalence forest of all the segment starts inside the L array. From this forest, it can be decided to finalize the labeling as described in subsection 4.3.3 or to compute some features as described in subsection 4.3.4.

Algorithm 8: HA4_Strip_Labeling(I, L, width)

```
1  declare shared array shared_pixels of size BLOCK_H
2  line_base ← y × width + tx
3  distance_y ← 0, distance_{y−1} ← 0
4  for i ← 0 to width by warp_size do
5      k_{y,x} ← line_base + i
6      p_{y,x} ← I[k_{y,x}]
7      pixels_y ← __ballot_sync(ALL, p_{y,x})
8      s_dist_y ← start_distance(pixels_y, tx)
9      if p_{y,x} and s_dist_y = 0 then
10         L[k_{y,x}] ← k_{y,x} (− distance_y if tx = 0)
11     __syncthreads()
12     if tx = 0 then shared_pixels[ty] ← pixels_y
13     __syncthreads()
14     pixels_{y−1} ← shared_pixels[ty-1] if ty > 0 else 0
15     p_{y−1,x} ← get bit tx of pixels_{y−1}
16     s_dist_{y−1} ← start_distance(pixels_{y−1}, tx)
17     if tx = 0 then
18         s_dist_y ← distance_y
19         s_dist_{y−1} ← distance_{y−1}
20     if p_{y,x} and p_{y−1,x} and (s_dist_y = 0 or s_dist_{y−1} = 0) then
21         label_1 ← k_{y,x} − s_dist_y
22         label_2 ← k_{y,x} − width − s_dist_{y−1}
23         merge(L, label_1, label_2)
24     d ← start_distance(pixels_{y−1}, 32)
25     distance_{y−1} ← d (+ distance_{y−1} if d = 32)
26     d ← start_distance(pixels_y, 32)
27     distance_y ← d (+ distance_y if d = 32)
```

Algorithm 9: HA4_Strip_Merge(I, L, width)

```
1  if y > 0 then
2      k_{y,x} ← y × width + x
3      k_{y−1,x} ← k_{y,x} − width
4      p_{y,x} ← I[k_{y,x}]
5      p_{y−1,x} ← I[k_{y−1,x}]
6      pixels_y ← __ballot_sync(ALL, p_{y,x})
7      pixels_{y−1} ← __ballot_sync(ALL, p_{y−1,x})
8      if p_{y,x} and p_{y−1,x} then
9          s_dist_y ← start_distance(pixels_y, tx)
10         s_dist_{y−1} ← start_distance(pixels_{y−1}, tx)
11         if s_dist_y = 0 or s_dist_{y−1} = 0 then
12             merge(L, k_{y,x} − s_dist_y, k_{y−1,x} − s_dist_{y−1})
```

4.3.3 CCL - Final labeling

A relabeling kernel is implemented to compare the CCL version of HA4 with previous works [30][139].

To avoid unnecessary memory accesses, each segment delegates the task of finding

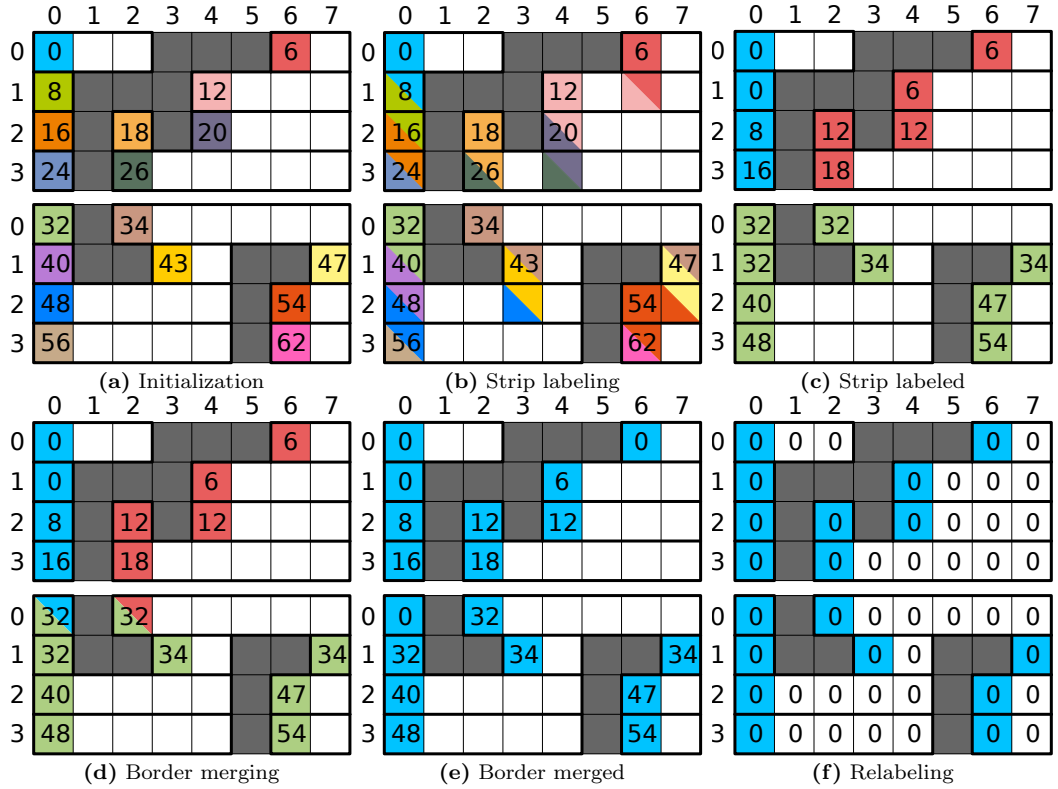


Figure 4.9: Example of the HA4 algorithm on an 8×8 image divided into two strips of height 4. In **(a)**, each segment start is initialized with its linear address. In **(b)**, local equivalences are resolved for each strip. In **(d)**, we merge the equivalence trees of the two strips. Finally, in **(f)**, each segment start finds the root of its tree and shares it with the other threads of the segment for relabeling.

the equivalence tree's root to the thread corresponding to its start. Once the start thread has found its true label, it propagates it to the other threads of the segment using a `__shfl_sync` instruction. After receiving the label, each thread updates the label image L. Algorithm 10 describes this kernel. Like in previous kernels, blocks are launched with their width equal to the warp size. Figure 4.9 shows the execution of the complete algorithm on a small image.

Algorithm 10: HA4_Relabeling(I, L, width)

```
1  k_{y,x} ← y × width + x
2  p_{y,x} ← I[k_{y,x}]
3  pixels ← __ballot_sync(ALL, p_{y,x})
4  s_dist ← start_distance(pixels, tx)
5  label ← 0
6  if p_{y,x} and s_dist = 0 then
7  |   label ← L[k_{y,x}]
8  |   while label ≠ L[label] do  label ← L[label]
9  label ← __shfl_sync(ALL, label, tx − s_dist)
10 if p_{y,x} then  L[k_{y,x}] ← label
```

4.3.4 CCA and Feature Computation

The CCA algorithm presented in this section uses the same warp and segments idea as in previous kernels. Maximum performance is reached when it is used in combination with the strip labeling and border merging kernels presented in previous subsections, but this kernel can be used after any algorithm that produces a label equivalence image. As previously, the core idea is that only the thread associated with the first pixel of each segment searches for the roots of its equivalence tree and updates the features with atomic operations. With the distance operators defined in subsection 4.3.1, the start thread can compute all the features for the segment from the pixel bitmask only. Algorithm 11 shows how to compute the most frequently used features: the number of pixels S, the sum of x coordinates S_x, the sum of y coordinates S_y and the bounding rectangle MIN_x, MIN_y, MAX_x and MAX_y. For a given segment starting at x_0 and ending at x_1, $S = x_1 - x_0 + 1$, $S_x = \phi(x_1) - \phi(x_0 - 1)$, and $S_y = y \times S$, with ϕ the sum of the first n integers: $\phi(n) = n(n+1)/2$. This algorithm is modular as we can remove unwanted features. It can also be noticed that the MIN_y feature is already encoded in the label and can be retrieved as $min_y = \lfloor label/width \rfloor$.

Algorithm 11: HA4_Features(I, L, *features*, width)

```
1  k_{y,x} ← y × width + x
2  p_{y,x} ← I[k_{y,x}]
3  pixels ← __ballot_sync(ALL, p_{y,x})
4  s_dist ← start_distance(pixels, tx)
5  count ← end_distance(pixels, tx)
6  sum_x ← ((2 × x + count − 1) × count) / 2
7  sum_y ← y × count
8  max_x ← x + count − 1
9  if p_{y,x} and s_dist = 0 then
10     label ← L[k_{y,x}]
11     while label ≠ L[label] do  label ← L[label]
12     atomicAdd(S[label], count)
13     atomicAdd(S_x[label], sum_x), atomicAdd(S_y[label], sum_y)
14     atomicMin(MIN_x[label], x), atomicMin(MIN_y[label], y)
15     atomicMax(MAX_x[label], max_x), atomicMax(MAX_y[label], y)
```

4.3.5 Processing two pixels per thread

At this point, the work done by the threads was successfully reduced. In fact, for the worst case scenario, when for every two pixels there is one white and one black pixel, only half of the threads are working. This means that in every situation, there could not be two consecutive threads in the same warp doing useful work at the same time. Therefore, the kernels can be modified to process two pixels per thread, as shown in figure 4.10.

In this new version, each warp of 32 threads is processing 64 pixels, so the horizontal thread index needs to be updated inside the kernels: $tx \leftarrow tx \times 2$ and `BLOCK_W` $\leftarrow$ `BLOCK_W` $\times 2$. The `uint64_t` type is used to store bitmasks and almost all the primitives used for 32-bit bitmasks are replaced by their 64-bit equivalent.

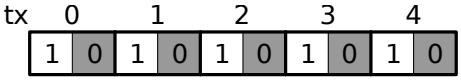


Figure 4.10: One thread can process two consecutive pixels.

Each thread loads the $p_{y,x}$ and $p_{y,x+32}$ pixel. As the `__ballot_sync` instruction can only create 32-bit bitmasks, two bitmasks have to be recombined into one 64-bit bitmask after the transfer. Two strategies were tested.

The first strategy is based on a fast bit interleaving operator as described in algorithm 12. Alternatively, the `unpack` operator can be optimized by the use of the `__byte_perm` instruction that could replace lines 2 and 3. It could be further optimized by defining an `unpack_high` operator that would produce the unpacked bitmask already shifted by one, for the odd part. In the border merging kernel, where two bitmasks need to be built, the interleaving work is distributed to even and odd threads and the 64-bit bitmasks are exchanged with a `__shfl_sync` instruction.

Algorithm 12: 32 to 64 bits unpack and interleave operators

```
1 Operator unpack(b)
2     b ← (b | (b << 16)) & 0x0000FFFF0000FFFF
3     b ← (b | (b <<  8)) & 0x00FF00FF00FF00FF
4     b ← (b | (b <<  4)) & 0x0F0F0F0F0F0F0F0F
5     b ← (b | (b <<  2)) & 0x3333333333333333
6     b ← (b | (b <<  1)) & 0x5555555555555555
7     return b
1 Operator interleave(even, odd)
2     return unpack(even) | (unpack(odd) << 1)
```

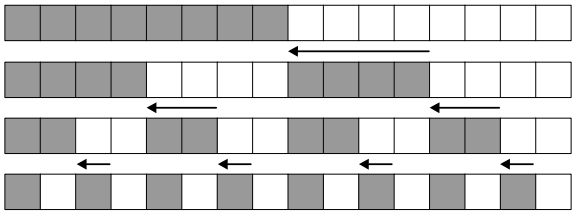


Figure 4.11: Logarithmic in-place unpack of 8-bit data into a 16-bit register. Data (in blank) is shifted recursively to make space (in grey) for the next shift.

The second strategy is more straightforward to implement but requires two load instructions instead of one. In most kernels, the second strategy is more efficient because of the cost of the interleaving operation. But in the border merging kernel the first strategy is slightly faster as the interleave cost is distributed among threads.

The distance operators also have to be slightly changed, as well as the features computation to take into account which of the two pixels processed by the current thread is the real root of the segment. Algorithm 13 describes the modified opera-

tors for 64-bit bitmasks.

Algorithm 13: Distance operators for 64-bit bitmasks

```
1 operator start_distance64(pixels, tx)
2     b ← get bit tx of ∼pixels
3     txb ← tx + b
4     return __clzll(∼(pixels << (64−txb)))
1 operator end_distance64(pixels, tx)
2     b ← get bit tx of ∼pixels
3     txb ← tx + b
4     return __ffsll(∼(pixels >> (txb+1)))
```

4.3.6 Experimental Evaluation

The state-of-the-art Playne [139] and Cabaret [30, 31] were implemented from their respective papers and compared to CCL / CCA HA4 on an embedded Jetson TX2 card. The GPU has 256 Pascal CUDA cores set to 1.3 GHz using the MAX_N performance setting. All codes were compiled with the CUDA 9.0. For reproducible results, MT19937 [124] was used to generate images of varying density ($d \in [0\% - 100\%]$) and granularity ($g \in \{1 - 16\}$) like in [30]. The granularity is the size of a macro-pixel side, it controls how clustered the pixels are and thus the minimum size of a connected component.

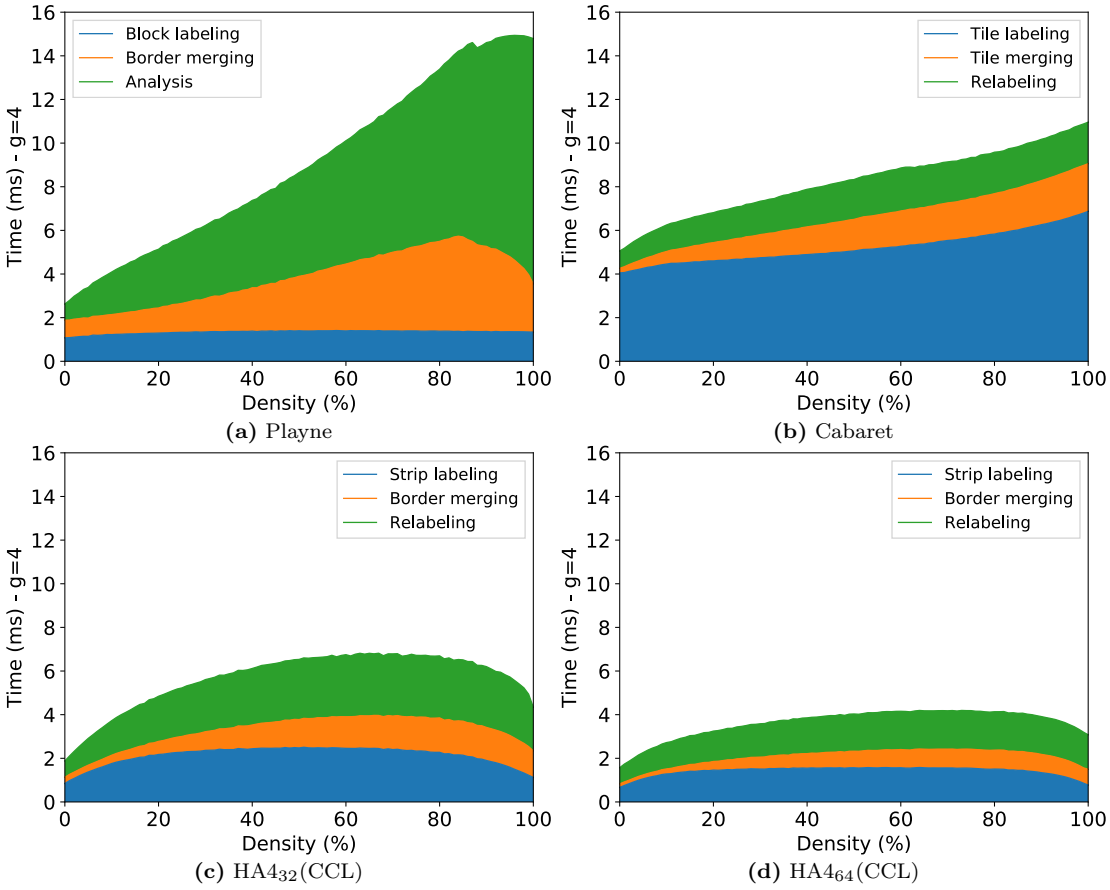
Figure 4.12 shows the execution time of the three steps of Playne, Cabaret and (the two versions of) HA4. Each step is labeled as in the original articles. Steps with the same color perform a similar function.

Thanks to the 64-bit version of HA4, each of the three steps is faster than those of other algorithms. HA4 is - on average over all densities - 2.4× faster than Playne or Cabaret for $g = 4$. When the granularity varies from $g = 1$ (worst case for segment processing) up to $g = 16$, the speedup ratio varies from 1.8 up to 2.7.

Figure 4.13 shows the execution time of CCA algorithms. The two first steps are identical to CCL algorithm. The third step (relabeling) is replaced by the analysis kernel that performs features computation (FC). The average time of FC is 1.75 ms, which is 6.4 times faster than a naive post-FC kernel (11.2 ms). It can be noted that the bump around $d = 64\%$ corresponds to the transition between many small components to a few large components, called percolation threshold, in 4-connectivity.

4.4 FLSL: Faster LSL for GPU

Concurrent voting algorithms rely on atomic read-modify-write instructions. When multiple threads vote in the same cell (i.e. same memory location), their accesses are serialized in order to keep the atomic aspect of the access. For CCA, voting happens when the features of a connected component are computed by multiple



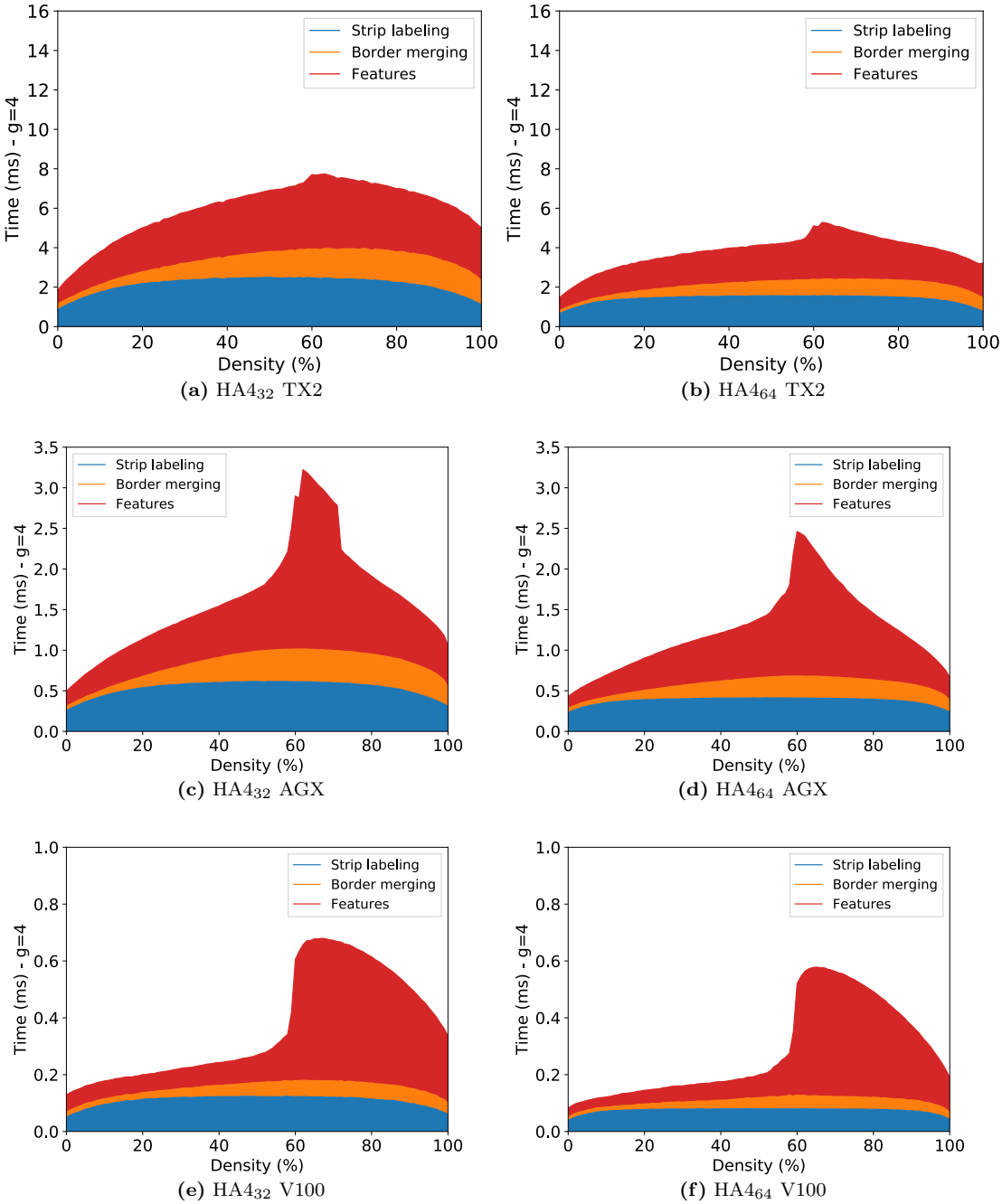
(a) Playne **(b)** Cabaret

(c) $HA4_{32}$(CCL) **(d)** $HA4_{64}$(CCL)

Figure 4.12: Labeling execution time of 2048×2048 images, $g = 4$.

threads in parallel. When a connected component is big, many threads will update the features of this component, and thus perform atomic memory accesses at the same location. This algorithm, even if parallel, performs in the same way as the equivalent sequential algorithm would, and loses all benefits from the parallelism of GPUs. The HA algorithm [76] is the only known CCA algorithm that tackles this issue. Figure 4.14 shows the processing time to process random images on an Nvidia A100 depending on the foreground pixel density for the naive CCA approach and the optimized HA algorithm. It can be clearly seen that the processing time for the naive CCA algorithm is very slow after the percolation threshold at $d = 60\%$ and keeps getting worse. In fact, the maximum processing time (at $d = 100\%$) is 19× higher than before the percolation threshold. At this point, the naive algorithm is fully serialized, and adding more cores will not improve the processing time. The HA algorithm partially solves this issue, but an elongated peak remains at the percolation threshold: the maximum processing time (at $d = 65\%$) is still 8× higher than before the percolation threshold.

An efficient solution to speed up the histogram computation is to have a private copy of the histogram for each thread (or at least warp), and merge them together at the end. However, this technique cannot be used for CCA as the number of cells is much higher (one cell per connected component). Three alternative techniques are proposed in the next sub-sections. They also apply to CCA.



(a) HA4$_{32}$ TX2 **(b)** HA4$_{64}$ TX2

(c) HA4$_{32}$ AGX **(d)** HA4$_{64}$ AGX

(e) HA4$_{32}$ V100 **(f)** HA4$_{64}$ V100

Figure 4.13: Analysis of the execution time for 2048×2048 images, $g = 4$.

4.4.1 Full segments (FLSL)

The HA algorithm [76] processes lines per block of 64 pixels per warp. It groups pixels into sub-segments within those blocks in order to reduce merge conflicts. Therefore a single warp processes exactly 64 pixels per iteration, even if there is a single segment within this block. Consequently, the longer the segments, the less parallelism is used. Moreover, if a segment spans multiple blocks, features for this segment will be updated multiple times (once per block).

In order to avoid those problems, it is possible to use *full* segments, and assign a thread per segment. If a segment spans the entire row, it will still be processed

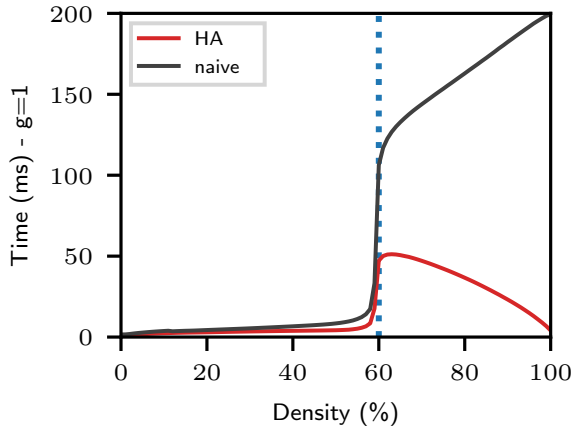


Figure 4.14: Time per image as a function of image density. State-of-the-art algorithms were run on 8192×8192 random images on a A100. Dotted line is the percolation threshold at $d = 60\%$.

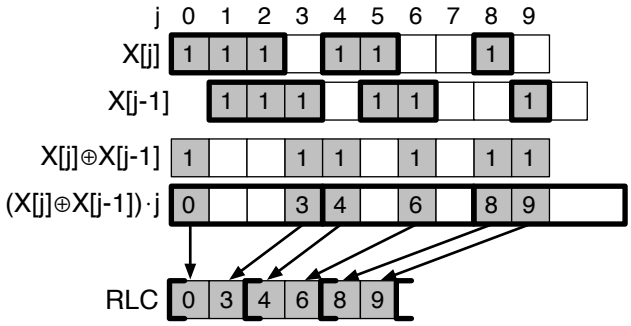


Figure 4.15: Example of a segment and its associated run-length encoding with a semi-open interval $[0, 3[4, 6[8, 9[$ with a 4-wide warp `compress`.

once and by only one thread. There are already CPU algorithms implementing those ideas: the LSL [32] and derivatives. FLSL [111], a variant of LSL for SIMD CPUs (SSE, AVX512, Neon), can be re-designed to target GPUs and address their architectural constraints. The crucial part is to first do a segment detection that consists in a run-length encoder (RLE) and relies on "compress-store", as shown on Figure 4.15. Indeed, the segment boundaries are the positions of the edges (when pixels change value). Compress-store can be implemented rather easily on GPUs thanks to `__ballot_sync` and `__popc` (Algorithm 14). Two passes are required to process a single row: first detect segments (one thread per pixel), and then label segments (one thread per segment). Those passes are done in the same kernel. Like HA or naive, feature updates are done once for each segment in a dedicated kernel after the image has been labeled.

4.4.2 On-The-Fly feature merge (OTF)

Contention can be further reduced by taking advantage of the fact that features can be computed while the connected components are discovered. This requires a concurrent way to move features from a location to another while two labels are merged together.

Algorithm 14: Kernel for FLSL segment detection

```
1  n ← 0 ▷ Number of runs on the line y
2  m_p ← 0 ▷ Previous pixel mask
   ▷ Detect runs
3  for x ← laneid() to width by warp_size do
4      p ← I[y · width + x]
5      m_c ← __ballot_sync(ALL, p)
       ▷ Detect edges
6      m_e ← m_c ⊕ __funnelshift_l(m_p, m_c, 1)
7      m_p ← m_c
       ▷ Count edges before current index
8      er ← n + __popc(m_e ∧ lanemask_le())
9      ER[y · width + x] ← er
       ▷ "Compress store"
10     if m_e ∧ m_l then RLC[y · width + er − 1] ← x
11     n ← n + count_edges(m_e) ▷ same n for the whole warp
12 if n is odd then
13     if tx = 0 then RLC[y · width + n] ← w
14     n ← n + 1
15 if tx = 0 then N[y] ← n
```

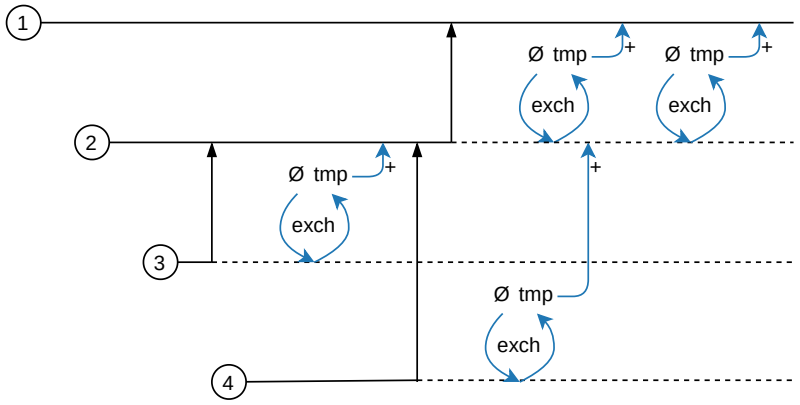


Figure 4.16: Lifelines of labels during OTF merge. Solid black lines are lifelines of labels as root. Lifelines are dashed when label is no longer a root. Black arrows are equivalence recording (Unions). Blue arrows are feature movements. Chronological order is from left to right.

To do so, their instantaneous roots are retrieved, and the higher one is made point to the lower one. The features from the higher one are extracted with an `atomicExch` with 0, preventing them from being extracted multiple times. Those extracted features are then merged with the features of the lower root with an `atomicAdd`. Similarly to Komura's equivalence building [99], those steps need to be repeated if the roots have been altered by another thread, as seen in Algorithm 15.

Figure 4.16 shows an example of such an on-the-fly merge with a representation of the lifelines of each label, the equivalences between labels and feature movements. On this example, the equivalence ② ≡ ① is recorded after ④ ≡ ②, but before features from ④ were merged into ②. Therefore, the thread merging ④ into ② needs to merge ② into its new root ①, otherwise, features will remain in a non-root node

Algorithm 15: Function for on-the-fly merge

```
1  operator otf_merge(l1, l2, L, S)
       ▷ l1 and l2 are two labels to merge
       ▷ L is the equivalence table
       ▷ S is the table in which the features are accumulated
2      l1 ← find_root(l1)
3      l2 ← find_root(l2)
4      __threadfence()
5      while l1 ≠ l2 do
6          if l2 < l1 then swap l1, l2
7          l ← atomicMin(L[l2], l1) ▷ label merge
8          __threadfence()
9          s ← atomicExch(S[l2], 0) ▷ feature extraction
10         atomicAdd(S[l1], s) ▷ feature merge in current root
11         __threadfence()
12         if l = l2 then break
13         l2 ← l
       ▷ Ensure the features have reached an actual root
14     a ← find_root(l1)
15     __threadfence()
16     while a ≠ l1 do
17         s ← atomicExch(S[l1], 0)
18         atomicAdd(S[a], s)
19         __threadfence()
20         l1 ← a
21         a ← find_root(l1)
22         __threadfence()
```

(i.e. the accumulation will be incomplete).

While the number of updates required with OTF is actually higher than without, the number of conflicts is reduced. Indeed, the updates are done while the connected components are not yet fully discovered: threads accumulate into provisional labels rather than final roots.

4.4.3 Conflict detection (CD)

The conflict detection (CD) variant, that transparently replaces the naive way of voting for computing features, can be used to reduce the number of collisions during feature updates. Before merging the features in memory, each thread will check which thread of the warp is processing the same component. This is done with the `__match_any_sync` primitive introduced in Volta GPUs. Threads will elect a leader per component, and accumulate their features into the leader with `__shfl_sync`, as illustrated by Figure 4.17. Then, only the leader actually accumulates the features for the component in memory. This way, only a single thread per warp accumulates features for a component, but multiple components can still be processed in parallel by the warp. The whole process is detailed in Algorithm 16.

This method is classified as "Opportunistic Warp-level Programming" [122] and

Algorithm 16: Function for feature update with conflict detection

```
1  operator feature_update_cd(mask, l, s)
2      peers ← __match_any_sync(mask, l)
3      rank ← __popc(peers ∧ lanemask_lt())
4      leader ← rank = 0
5      peers ← peers ∧ lanemask_gt()
       ▷ Reduce features among peers
6      while __any_sync(mask, peers) do
7          next ← __ffs(peers)
8          s′ ← __shuffle_sync(mask, s, next)
9          if next ≠ 0 then s ← s + s′
10         peers ← peers ∧ __ballot_sync(mask, rank is even)
11         rank ← rank >> 1
12     if leader then atomicAdd(S[l], s)
```

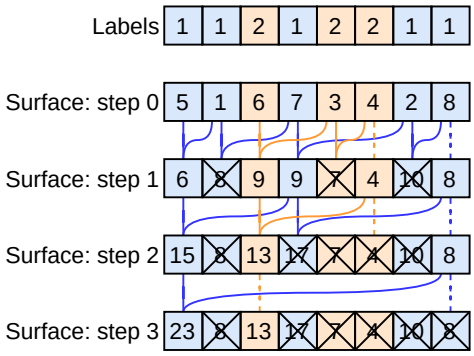


Figure 4.17: Parallel masked reduction for conflict detection during surface computation.

is made possible by the `__match_any_sync` instruction. The other crucial part of this algorithm is the reduction of features into the leader. A warp can process multiple components and thus it is necessary to perform multiple reductions on distinct partitions of the warp in parallel. Such "masked reduction" can be done using the new `__reduce_op_sync` instructions introduced with the Ampere architecture, or emulated using older warp-level intrinsics as in [164]. However, the native instruction was found to be serializing the reductions instead of doing multiple independent reductions in parallel, leading to a worst case 11× slowdown when doing 32 parallel reductions compared to the custom algorithm. For this reason, all benchmarks are using the custom algorithm on all architectures.

4.4.4 Number of updates and conflicts

The number of atomic updates has been precisely measured for each connected component. The number of conflicts is estimated from the number of atomic updates as the probability that two feature updates picked at random are to the same label (i.e. the same memory location) multiplied by the total number of updates. If $\mathcal{U}_l$ is the number of updates of a label l, then the estimated number of conflicts is $\left(\sum_l \mathcal{U}_l^2\right) / \left(\sum_l \mathcal{U}_l\right)$.

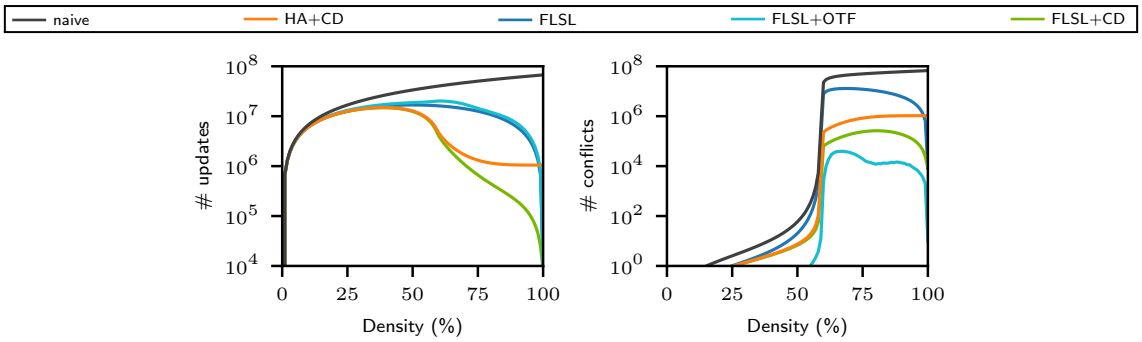


Figure 4.18: Number of atomic updates and conflicts for all versions on 8192×8192 random images with a granularity $g = 1$ as a function of density. Number of conflicts is estimated from a very simple probabilistic model. Logarithmic scale is used to accommodate the wide range of values.

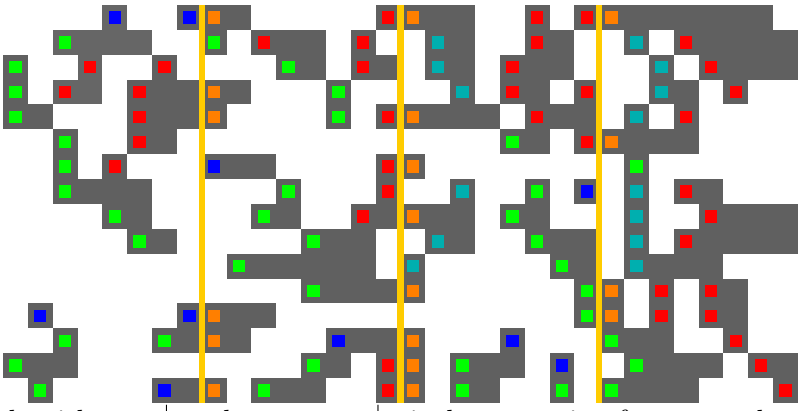
Figure 4.18 shows the number of atomic updates of the features as well as the estimation of the number of conflicting updates. HA and HA+OTF have been omitted from Figure 4.18 as they are almost identical to FLSL and FLSL+OTF respectively. The number of atomic updates of the naive version is linear with the number of foreground pixels, and all other versions reduce the number of updates. *Full* runs (FLSL) decrease the number of updates slightly more than HA, but this effect is mainly visible for high density images. On-the-fly merges (OTF) actually increase the number of updates, especially around the percolation threshold (at $d = 60\%$). Conflict detection (CD) highly reduces the number of updates, especially after the percolation threshold. Figure 4.19 is a visual example of the reduction of the update number (OTF is excluded from this example because of its parallel nature).

Looking at the number of conflicts, the picture is drastically different. First, the number of conflicts before the percolation threshold is tiny for all versions, even the naive one. Then, despite the higher number of updates, OTF actually has the lowest conflict count after the percolation threshold. Indeed, the updates are performed on different labels; hence the probability that two threads are conflicting decreases. The behavior of the other versions is similar to the number of atomic updates.

According to these numbers, CD and OTF are effective in reducing the number of conflicts. CD is also great at reducing the total number of updates, especially when combined with FLSL.

4.4.5 Experimental Evaluation

In order to characterize how algorithms perform, all the variants proposed were run on random images, as well as the state-of-the-art algorithm HA[76]. For reproducible results, MT19937 [124] was used to generate images of varying density ($d \in [0\% - 100\%]$) and granularity ($g \in \{1 - 16\}$) like in [25]. A smaller granularity means more complex images with finer details.



algorithm	updates count	pixels generating feature updates
naive	229	
HA	119	
FLSL	101	
HA+CD	80	
FLSL+CD	48	
lower-bound	10	

Figure 4.19: Example showing the difference in feature updates of the algorithms. For the sake of demonstration, 8-connectivity is used and warps are 8 pixels wide and their vertical boundaries are represented with yellow lines (relevant only for HA algorithms).

The most commonly used features are the number of pixels (i.e. the surface) of the connected component (S) and the bounding box (BB), which gives the extents (x_0, y_0, x_1, y_1) of the component in the x and y directions. Those features are encoded as 32-bit integers, which is enough for images smaller than 2^{32} pixels (65536×65536 images). Another feature of interest is the centroid that can be computed from the surface S and the first statistical raw moments Sx and Sy (respectively the sum of x and y coordinates of the pixels). Those extra features require 64-bit integers in order to handle images larger than 2048×2048 without overflows. It was chosen to compute all 7 features (five 32-bit integers and two 64-bit integers) as it represents a useful set of features for many applications. Moreover, the function `connectedComponentsWithStats` from OpenCV [27] computes exactly those features, but is unfortunately unavailable for GPU.

All versions were run on an Nvidia Tesla A100 GPU for 8192×8192 random images at several granularities ($g \in \{1, 4, 16\}$). For each image, the algorithm was run 20 times and the minimum processing time was taken. Throughputs are averaged over the density range $[0\% - 100\%]$.

Looking at the processing time of all the variants as a function of the density (Figure 4.20, top), the peak at the percolation threshold for HA and FLSL can be seen, as expected from the estimated number of conflicts. OTF is efficient after the percolation threshold ($d \geqslant 60\%$) where most conflicts are expected, but seems to suffer from having more updates before as it is slower than without it. CD is efficient both before the percolation where it basically does not change the processing time, and after the percolation where the conflict reduction is useful. Looking at higher

granularities (less detailed images), it appears that HA+CD is not much different from HA alone, and suffers from processing sub-runs instead of *full* runs. FLSL+CD appears to be the most effective because the conflict detection is applied on more updates than with HA+CD. The picture is mostly the same for all sizes (Figure 4.20, bottom). In particular, FLSL+CD remains the fastest for all sizes and granularities.

Table 4.1 summarizes the average throughput of various algorithms and different configurations. The naive and HA algorithms are shown as a reference point. First, the impact of the OTF and CD transformations to the HA algorithm are shown, then three versions of the proposed new algorithm (FLSL). The configurations have varying granularity to represent images containing different component sizes. The full image configuration represent an image with only one big component, filling the whole space: it is the worst case for the naive CCA algorithm as it maximizes the number of memory conflicts. Indeed, both naive and HA struggle on full images, while OTF and FLSL variants achieve best throughput for full images. FLSL+CD appears to be the fastest for all the benchmarked configurations and is from 4 to 10 times faster than HA alone, on average.

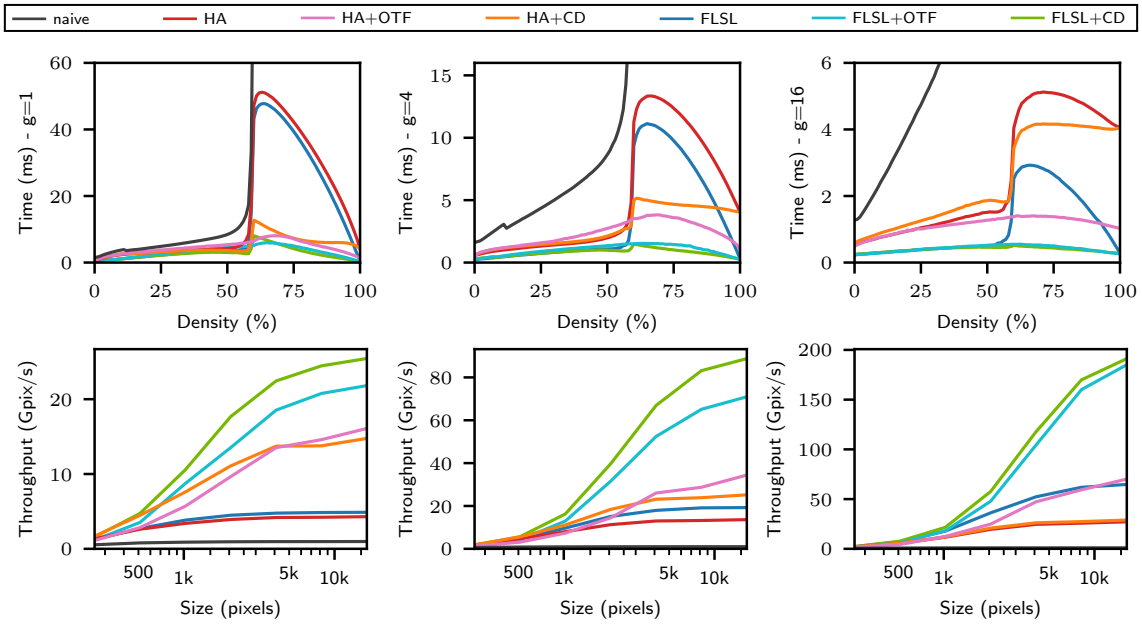


Figure 4.20: Time per image for $g = \{1, 4, 16\}$ on Nvidia A100. Time versus density for 8192×8192 images (top). Average throughput versus size from 256×256 to 16384×16384 (bottom).

Following the work of Hockney [85], the new algorithms were characterized on a big and a little GPU using two metrics: r_∞ and $n_{1/2}$. r_∞ is the maximum or asymptotic performance (here throughput in giga-pixels per seconds (Gpix/s)) and occurs in the limit of infinite image size. $n_{1/2}$ is the half-performance size (here the image size) that is necessary in order to achieve half the maximum performance. It is a measure of the hardware parallelism and also takes into account the effect of image size on the throughput and thus determines the choice of the best algorithm on a particular hardware. Figure 4.21 shows this characterization at different granularity on a low power embedded GPU and a high-end server GPU. In both cases, the $n_{1/2}$ half-performance is reached for realistic image sizes, indicating that the algorithm

Algorithm	$g = 1$	$g = 4$	$g = 16$	full image
naive	0.966	0.994	0.985	0.337
HA	4.22	13.2	25.8	16.6
HA+OTF	14.6	28.7	59.3	66.2
HA+CD	13.8	23.9	27.4	16.6
FLSL	4.85	19.1	61.9	**244**
FLSL+OTF	20.8	65.1	160	238
FLSL+CD	**24.5**	**83.2**	**170**	**244**

Table 4.1: Average CCA throughput (Gpix/s) for 8192×8192 on an Nvidia A100.

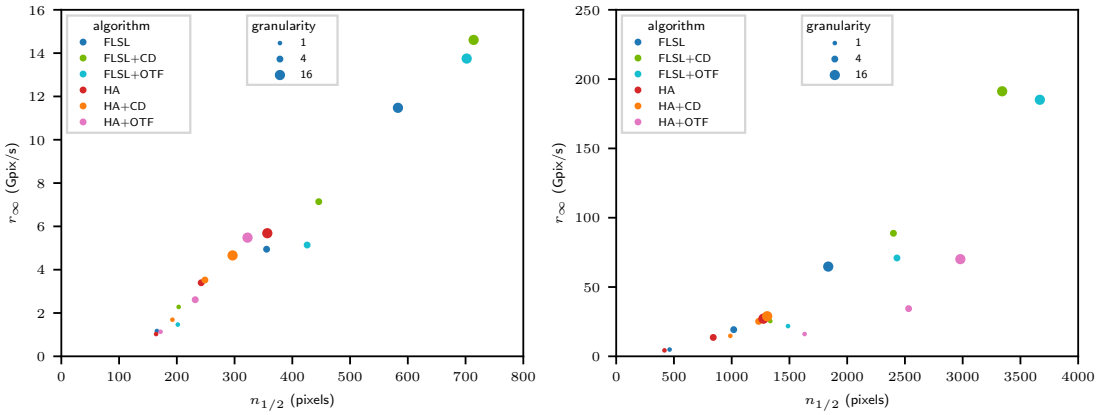


Figure 4.21: r_∞ and $n_{1/2}$ performance of FLSL and HA algorithms for $g = \{1, 4, 16\}$ on a Nvidia Jetson AGX Xavier (left) and a Nvidia A100 GPU (right).

utilizes the computing resources efficiently.

4.5 SIMD Rosenfeld

4.5.1 SIMD Union-Find

The biggest challenge when designing an SIMD CCL algorithm is to design a fast and concurrency-free union-find algorithm to manage equivalences. The union algorithm must take into account the conflicts from multiple equivalence tree roots involved in simultaneous merge operations. Figure 4.22 shows an example of such complex merges. The top row shows the evolution of the root label pointers; and the bottom row shows the pending merge operations in black and the finished merge operations in grey.

Algorithms presented in this section and those following are written for an SIMD cardinality (*card*) of 8 (for the sake of clarity) but can easily be extended to 4 or 16 elements. Unmasked equalities and inequalities between vectors are tested using the intrinsics `cmpeq_epi32_mask` and `cmpneq_epi32_mask`, but are written as mathematical comparisons to be more readable. Masked comparisons are expressed using their corresponding intrinsic.

Algorithm 17 uses gather loads to find the roots of all labels in its parameter vec-

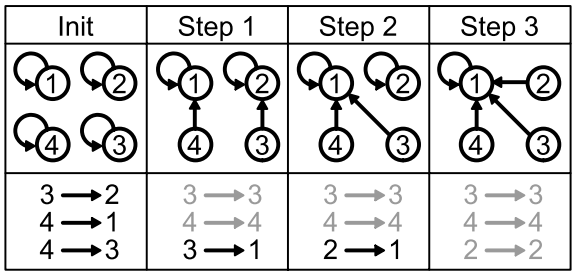


Figure 4.22: Execution of algorithm 18 `VecUnion` with arguments $\vec{e_1} = [3, 1, 2]$, $\vec{e_2} = [4, 4, 3]$ (example of simultaneous unions in 3 steps and their serialization).

tor of label $\vec{e}$. The loop must run until all roots have been found, so the number of iterations is equal to the maximum distance between involved labels and their roots.

Algorithm 17: VecFind($\vec{e}$, T, m)

Input: $\vec{e}$ a vector of label, T an equivalence table, m a mask
Result: $\vec{r}$, the roots of $\vec{e}$

```
1 r ← e
2 done ← 0 // mask
3 while done ≠ m do
4     l ← mask_i32gather_epi32(r, ¬done ∧ m, r, T, sizeof(uint32))
5     done ← mask_cmpeq_epi32_mask(l, r, m) // mask
6     r ← l
7 return r
```

Algorithm 18 is inspired by the Playne-Equivalence reduce function designed for parallel CCL on GPU [139]. The main difference with a GPU algorithm is to use the parallelism within an SIMD vector instead of the parallelism between GPU threads in memory. To solve the concurrency issue the same way the GPU does, the `VecScatterMin` function, which emulates the behavior of the CUDA `atomicMin` function, is introduced. This function takes two vectors $\vec{idx}$ and $\vec{val}$ and tries to perform the operation: $T[\vec{idx}] \leftarrow min(T[\vec{idx}], \vec{val})$. It then returns the old value of $T[\vec{idx}]$ if the operation succeeded, or the current value if another vector element has written it first. Using this function, only one value is written to a memory address at a time, but it is always the minimum value of the concurrent store operations. This allows the `VecUnion` operation to retry until all involved equivalence trees have been merged. Because of the pixel topology, there can be at most $card/2$ simultaneous equivalence tree merges. In practice, the merge vectors are very sparse, allowing for the reduction of the number of operations needed by compressing the vectors. The `VecScatterMin` function is described in Algorithm 19.

4.5.2 SIMD Rosenfeld pixel algorithm

Like its scalar counterpart, the SIMD Rosenfeld pixel algorithm (v1) is a two pass direct CCL algorithm. In order to simplify the algorithm and to improve memory footprint and performance, the equivalence table is embedded into the image. The label creation can now easily be done in parallel as the new label is equal to the

Algorithm 18: VecUnion($\vec{e_1}$, $\vec{e_2}$, T, m)

Input: $\vec{e_1}$, $\vec{e_2}$ two vectors of labels, T an equivalence table, m a mask

```
1  r⃗1 ← VecFind(e⃗1, T, m)
2  r⃗2 ← VecFind(e⃗2, T, m)
3  m ← mask_cmpneq_epi32_mask(m, e⃗1, e⃗2)  // mask
4  while m do
5      r_max, r_min ← max_epu32(r⃗1, r⃗2), min_epu32(r⃗1, r⃗2)
6      r⃗1, r⃗2 ← r_max, r_min
7      r⃗3 ← VecScatterMin(r⃗1, r⃗2, T, m)
8      done ← mask_cmpeq_epi32_mask(m, r⃗1, r⃗3)  // mask
9      r⃗1 ← r⃗3
10     m ← ¬done ∧ m  // mask
```

Algorithm 19: VecScatterMin($\overrightarrow{idx}$, $\overrightarrow{val}$, T, m)

Input: $\overrightarrow{idx}$, $\overrightarrow{val}$ two vectors of labels, T an equivalence table, m a mask

Result: $\vec{r}$, the old values of $T[\overrightarrow{idx}]$

```
1  rotate ← set_epi32(0, 7, 6, 5, 4, 3, 2, 1)
2  idx_c ← maskz_compress_epi32(m, idx)
3  val_c ← maskz_compress_epi32(m, val)
4  n ← popcnt_u32(m)
5  rotate ← maskz_move_epi32(0xFFFF >> (17 − n), rotate)
6  idx_r, val_r ← idx_c, val_c
7  for i ← 0 to n do
8      idx_r ← permute_epi32(rotate, idx_r)
9      val_r ← permute_epi32(rotate, val_r)
10     same_addr ← idx_c = idx_r  // mask
11     val ← mask_min_epu32(val, same_addr, val, old)
12 old ← mask_i32gather_epi32(idx, m, idx, T, sizeof(uint32))
13 new ← maskz_expand_epi32(m, val_c)
14 new ← min_epu32(new, old)
15 mask_i32scatter_epi32(T, m, idx, new, sizeof(uint32))
16 r⃗ ← mask_mov_epi32(new, cmpeq_epi32_mask(new, val), old)
17 return r⃗
```

linear address of the pixel plus 1 to differentiate the background: $i \times w + j + 1$, where (i, j) are the pixel coordinates and w the width of the image. This bijection also allows for faster relabeling as it can be done during the transitive closure step.

Algorithm 20 describes the processing of a pixel vector during the first pass. The pixels are processed in a sequential natural reading order. Neighbor vectors $\vec{a}$, $\vec{b}$, $\vec{c}$, $\vec{d}$ and the current pixel $\vec{x}$ can be obtained by doing aligned loads or by register rotation. Border and corner cases can be handled by setting the out of image pixel vectors to zero. The algorithm starts by constructing unaligned vectors $\vec{ab}$ and $\vec{bc}$ from $\vec{a}$, $\vec{b}$, $\vec{c}$ by doing some element shifting (lines 2 and 3). It also computes the bitmask m corresponding to $\vec{x}$: a bit is set to 1 for a foreground pixel and to 0 for a background pixel (line 4). The next step is to initialize the labels in $\vec{x}$ as shown in figure 4.24. Each pixel either points to itself or to its left-side neighbour (lines 6 to 9). $\vec{dx}$ can now be computed from $\vec{d}$ and $\vec{x}$ and the neighbour labels can be propagated into $\vec{x}$ (lines 10 and 11). The `vec_maskz_min`^+_n function can be imple-

mented using the property of unsigned integers overflow ($-1 =$ `MAX_UINT`) and the `maskz_min_epu32` intrinsics. Finally, $\vec{x}$ can be stored to memory (line 13) and we can call the `VecUnion` function described in subsection 4.5.1 (lines 15 to 18). As previously said, only the stairs and concavity patterns can lead to a union operation. The other configurations are handled with the label propagation step. The equivalence table pointer can be moved by 1 pixel to account for the $+1$ in the labels and simplify memory accesses.

The second pass is the simultaneous transitive closure and relabeling. In this pass the neighbour information is not needed, making the loading pattern straightforward. For each vector of pixel $\vec{e}$, the corresponding equivalence tree root is found and written back to memory. Processing the pixels in the same order as in the first pass allows the algorithm to capitalize on previous iterations to find the roots faster. The true number of label n can be computed using the `popcnt_u32` (population count in a 32-bit integer) instruction and the fact that, by definition, a root label points to itself. Algorithm 21 describes this process.

Figure 4.23 represents the execution of the SIMD Rosenfeld pixel algorithm on a 12×4 image and SIMD register of size of 4. The outlined area shows the steps of algorithm 20. The `VecUnion` operation is not detailed here but the pattern is similar to the one in figure 4.22. The modifications in the image from the scan ($9 \rightarrow 4$, $18 \rightarrow 4$) in figure 4.23 are due to the equivalence table being embedded in the image.

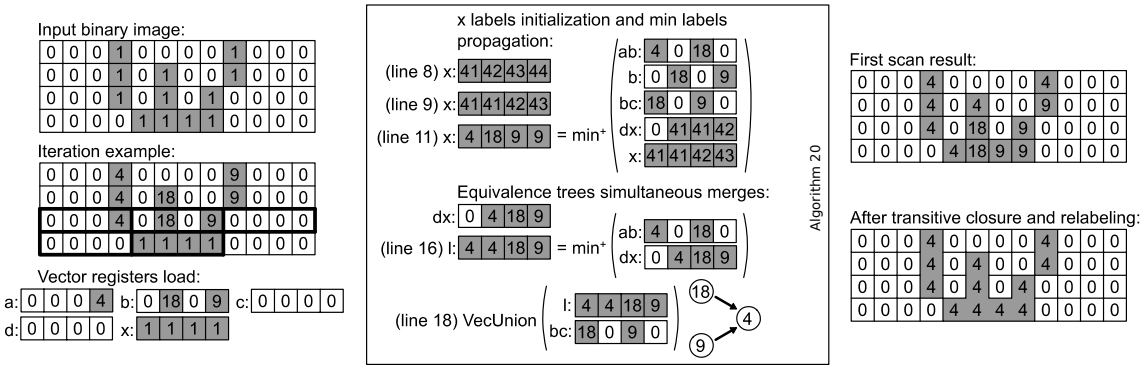


Figure 4.23: Example of an iteration of the SIMD Rosenfeld pixel algorithm.

4.5.3 SIMD Rosenfeld sub-segment algorithm

The SIMD Rosenfeld pixel algorithm works well for low and medium image densities but for high image densities the performance collapses due to the pixel-recursive labels leading to more iterations in the `VecFind` while loop. To address this issue at the cost of a few more operations, the SIMD Rosenfeld sub-segment algorithm (v2) is introduced. The only difference with the SIMD Rosenfeld pixel algorithm lies in the way new labels are produced (Figure 4.24).

A segment is defined as a sequence of same value pixels. A sub-segment is a segment bounded by the size of a vector. The conflict detection instructions from the

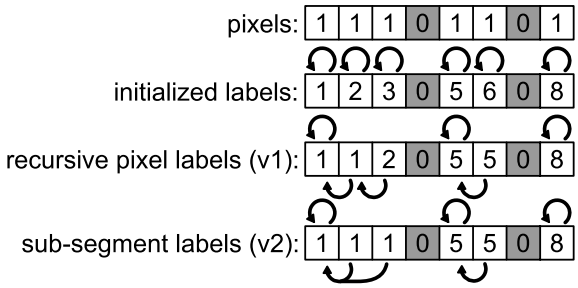


Figure 4.24: Creation of labels in SIMD Rosenfeld pixel algorithm (v1) and SIMD Rosenfeld sub-segment algorithm (v2).

Algorithm 20: SIMD Rosenfeld pixel (v1)

Input: T, the image / equivalence table, $\vec{a}, \vec{b}, \vec{c}, \vec{d}$, four vector of neighbor labels, $\vec{x}$, the current vector of pixels in (i, j), w, the width of the image

1 // Shuffles :
2 $\vec{ab} \leftarrow$ `vec_shift_right_epi32`$(\vec{a}, \vec{b})$// Shift one element and insert the first element of $\vec{b}$
3 $\vec{bc} \leftarrow$ `vec_shift_left_epi32`$(\vec{b}, \vec{c})$
4 $m \leftarrow \vec{x} \neq \vec{0}$ // mask
5 // x labels initialization and min labels propagation :
6 $\overrightarrow{inc} \leftarrow$ `set_epi32`$(7, 6, 5, 4, 3, 2, 1, 0)$
7 $\overrightarrow{base} \leftarrow$ `set1_epi32`$(i \times w + j + 1)$
8 $\vec{x} \leftarrow$ `maskz_add_epi32`$(m, \overrightarrow{base}, \overrightarrow{inc})$
9 $\vec{x} \leftarrow$ `mask_sub_epi32`$(\vec{x}, m \wedge (m << 1), \vec{x}, \vec{1})$
10 $\vec{dx} \leftarrow$ `vec_shift_right_epi32`$(\vec{d}, \vec{x})$
11 $\vec{x} \leftarrow$ `vec_maskz_min`$_5^+(m, \vec{ab}, \vec{b}, \vec{bc}, \vec{dx}, \vec{x})$
12 // Store x :
13 `mask_store_epi32`$(\&T[i][j], m, \vec{x})$
14 // Equivalence trees simultaneous merges :
15 $\vec{dx} \leftarrow$ `vec_shift_right_epi32`$(\vec{d}, \vec{x})$
16 $\vec{l} \leftarrow$ `vec_maskz_min`$^+(m, \vec{ab}, \vec{dx})$
17 $merge \leftarrow (\vec{bc} \neq \vec{0}) \wedge (\vec{b} = \vec{0}) \wedge (\vec{l} \neq \vec{0}) \wedge m$ // mask
18 `VecUnion`$(\vec{l}, \vec{bc}, \&T[0][-1], merge)$

Algorithm 21: SIMD Transitive closure (solve equivalences)

Input: T, the image / equivalence table, w, the width of the image
Result: n, the true number of labels in the image (optional)

1 $n \leftarrow 0$
2 **for each** $\vec{e} \in T$ **do**
3 $m \leftarrow \vec{e} = \vec{0}$ // mask
4 $\vec{e} \leftarrow$ `VecFind`$(\vec{e}, T, m)$
5 `mask_store_epi32`$(\&T[i][j], m, \vec{e})$
6 // (i, j) are the coordinates of $\vec{e}$
7 $\overrightarrow{inc} \leftarrow$ `set_epi32`$(7, 6, 5, 4, 3, 2, 1, 0)$
8 $\overrightarrow{base} \leftarrow$ `set1_epi32`$(i \times w + j + 1)$
9 $\vec{l} \leftarrow$ `maskz_add`$(m, \overrightarrow{base}, \overrightarrow{inc})$
10 $n \leftarrow n +$ `popcnt_u32`$($`mask_cmpeq_epi32_mask`$(m, \vec{l}, \vec{e}))$

AVX512CD instruction set are used to compute a conflict-free subset ($\vec{cfss}$) which, with the count leading zero instruction (`lzcnt_epi32`) and some bit manipulation, allows the retrieval of the index of the first element of a sub-segment. By making

the pixel point directly to the sub-segment start, the number of `VecFind` iterations can be reduced to only one jump per sub-segment. Algorithm 22 describes theses changes and figure 4.25 shows the key states of the sub-segment start computation code. Lines 14 to 16 of algorithm 22 are optional for the correctness of the algorithm but improve the performance.

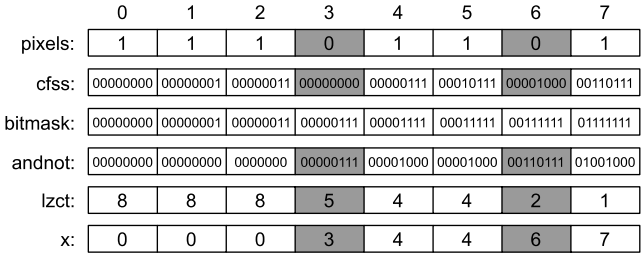


Figure 4.25: Use of conflict detection to find the sub-segment's start indices. In this example, elements have 8 bits and `lzct` count from the 8^{th} bit instead of the 32^{nd}.

Algorithm 22: SIMD Rosenfeld sub-segment (v2)

Input: T, the image / equivalence table, $\vec{a}, \vec{b}, \vec{c}, \vec{d}$, four vector of neighbor labels, $\vec{x}$, the current vector of pixels in (i, j), w, the width of the image

1 // Shuffles :
2 $\vec{ab} \leftarrow$ `vec_shift_right_epi32`$(\vec{a}, \vec{b})$
3 ; $\vec{bc} \leftarrow$ `vec_shift_left_epi32`$(\vec{b}, \vec{c})$
4 $m \leftarrow \vec{x} \neq \vec{0}$
5 // x labels initialization and min labels propagation :
6 $\overrightarrow{bitmask} \leftarrow$ `set_epi32`$(\texttt{0x7F}, \texttt{0x3F}, \texttt{0x1F}, \texttt{0xF}, 7, 3, 1, 0)$
7 $\overrightarrow{cfss} \leftarrow$ `maskz_conflict`(m, x)
8 $\overrightarrow{lzct} \leftarrow$ `lzcnt_epi32`$(\texttt{andnot\_epi32}(\overrightarrow{cfss}, \overrightarrow{bitmask}))$
9 $\overrightarrow{base} \leftarrow$ `set1_epi32`$(i \times w + j + 1 + 32)$
10 $\vec{x} \leftarrow$ `maskz_sub_epi32`$(m, \overrightarrow{base}, \overrightarrow{lzct})$
11 $\vec{dx} \leftarrow$ `vec_shift_right_epi32`$(\vec{d}, \vec{x})$
12 $\vec{x} \leftarrow$ `vec_maskz_min`$_5^+(m, \vec{ab}, \vec{b}, \vec{bc}, \vec{dx}, \vec{x})$
13 // Optional propagation:
14 $\overrightarrow{perm} \leftarrow$ `maskz_sub_epi32`$(m, \vec{32}, \overrightarrow{lzct})$
15 $\vec{x_p} \leftarrow$ `permute_epi32`$(\overrightarrow{perm}, \vec{x})$
16 $\vec{x} \leftarrow$ `vec_maskz_min`$^+(m, \vec{x}, \vec{x_p})$
17 // Store:
18 `mask_store_epi32`$(\&T[i][j], m, \vec{x})$
19 // Equivalence trees simultaneous merges :
20 $\vec{dx} \leftarrow$ `vec_shift_right_epi32`$(\vec{d}, \vec{x})$
21 $\vec{l} \leftarrow$ `vec_maskz_min`$^+(m, \vec{ab}, \vec{dx})$
22 $merge \leftarrow (\vec{bc} \neq \vec{0}) \wedge (\vec{b} = \vec{0}) \wedge (\vec{l} \neq \vec{0}) \wedge m$
23 `VecUnion`$(\vec{l}, \vec{bc}, \&T[0][-1], merge)$

4.5.4 Multi-thread SIMD algorithms

OpenMP is used for the parallel implementation and the assumption is made that the memory model is NUMA with shared memory between processors. SIMD algorithms have an increased pressure on memory bandwidth, which tends to reduce

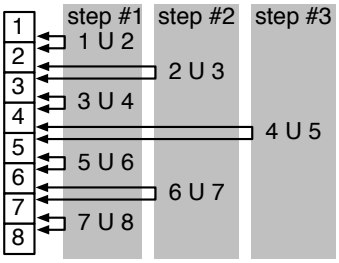


Figure 4.26: Pyramidal border merging of disjoint sets.

the multi-core parallelism efficiency if the application is not compute bound.

The approach followed in the parallel implementations of the SIMD Rosenfeld pixel and sub-segment algorithms is based on a divide-and-conquer method described in [32]. The image is split into p sub-images, with p the number of cores. This parallel algorithm minimizes the number of merges required by taking a pyramidal approach, but diminishes the number of active cores at a given time. It needs $\texttt{log}_2(p)$ steps to complete the merge. Each step is fully parallel and does not require `atomic` instructions to update the equivalence table as this scheme merges borders of disjoint sets (Fig. 4.26).

First, each processor core takes a sub-image horizontal strip and applies the first pass of either algorithm. Except for the modified loop indexes, there is no difference between the sequential and parallel code. The next step consists in applying a pyramidal merge. As the image is divided by the number of cores available p, the total number of merges required is equal to $p-1$. For each merge step, the number of active cores is divided, doing a two-way merge until only one core is left. On the border line between two sub-images, a merging pass is applied on each column. For each SIMD vector of pixels, there are at most two calls to `VecUnion`. Algorithm 23 describes the body of the border merge loop. Finally, the second pass is applied which does not require any code modification compared to the sequential version.

Algorithm 23: SIMD Border merging

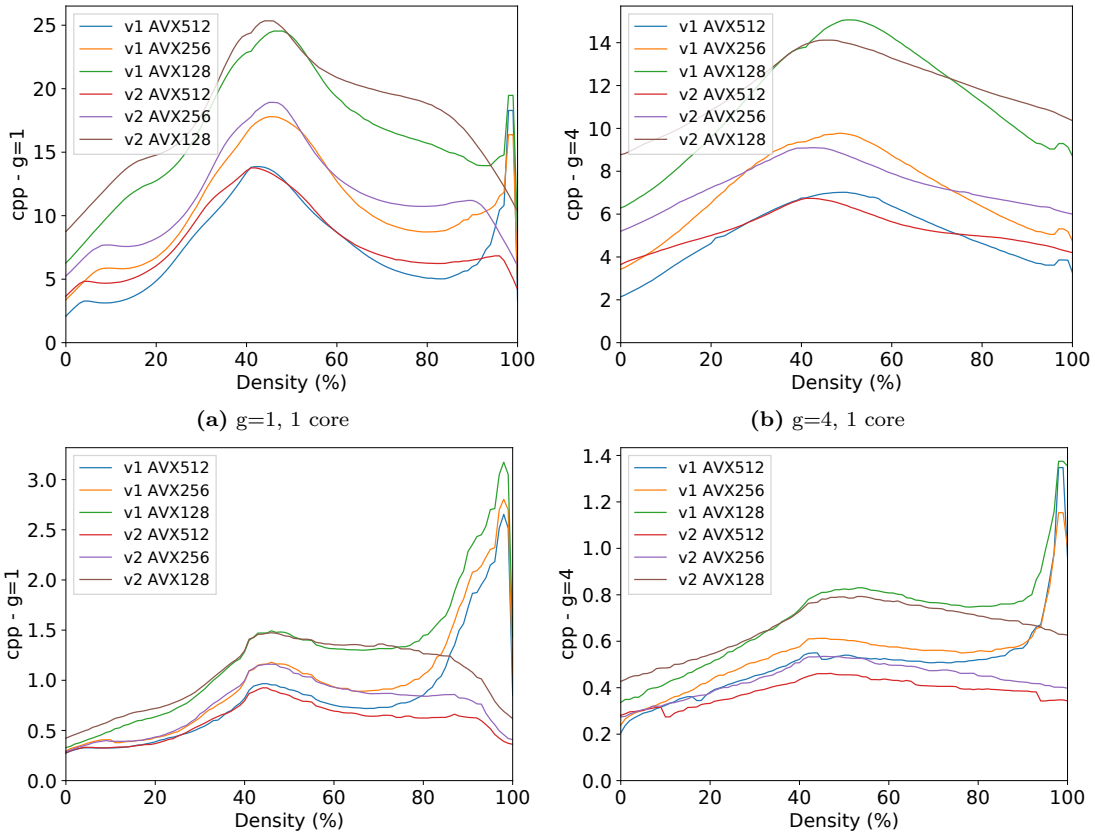
Input: T, the image / equivalence table, $\vec{a}, \vec{b}, \vec{c}, \vec{d}$, four vector of neighbor labels, $\vec{x}$, the current vector of pixels in (i, j), w, the width of the image

1 // Shuffles :
2 $\vec{ab} \leftarrow$ `vec_shift_right_epi32`$(\vec{a}, \vec{b})$
3 $\vec{bc} \leftarrow$ `vec_shift_left_epi32`$(\vec{b}, \vec{c})$
4 $\vec{dx} \leftarrow$ `vec_shift_right_epi32`$(\vec{d}, \vec{x})$
5 $m \leftarrow \vec{x} \neq \vec{0}$
6 // Equivalence trees simultaneous merges :
7 $\vec{l} \leftarrow$ `vec_maskz_min`$^+(m, \vec{ab}, \vec{b})$
8 $merge \leftarrow (\vec{l} \neq \vec{0}) \wedge (\vec{dx} = \vec{0}) \wedge m$ // mask
9 `VecUnion`$(\vec{x}, \vec{l}, \&T[0][-1], merge)$
10 $\vec{l} \leftarrow$ `vec_maskz_min`$^+(m, \vec{b}, \vec{bc})$
11 $merge \leftarrow (\vec{l} \neq \vec{0}) \wedge (\vec{b} = \vec{0}) \wedge m$ // mask
12 `VecUnion`$(\vec{x}, \vec{l}, \&T[0][-1], merge)$

4.5.5 Experimental Evaluation

The performance of the available SIMD vector length (128, 256, 512) was tested in single and multi-core on a dual socket Intel Xeon(R) Gold 6126 running at 2.6 Ghz (turbo-boost off). Before any testing, it was unclear whether the new 512 vector length would have a positive impact on the performance due to the additional stress on the memory bandwidth and the frequency throttle. This is especially true while fully exploiting SIMD and multi-core due to bandwidth saturation. The results are summarized in table 4.2, 4.3 and figures 4.27, 4.28, and are discussed in the following section. The performance of these new algorithms is compared to the classic pixel-based algorithms with DT (Rosenfeld and Rosenfeld+DT) and to the fastest run-length based segment labeling algorithms (LSL_{STD} and LSL_{RLE}) [32].

Figure 4.27 shows the execution time (in cycles per pixels) on images of varying densities. It can be observed that for both versions, there is a bump at around 45% density corresponding to the percolation threshold in 8-connectivity. It can also be seen that the main difference in performance between the pixel and sub-segment algorithms happens for higher densities. The pixel version is slower because the recursive pixel labels lead to a longer while loop in *VecFind* (as seen in Sec. 4.5.3). This performance difference grows with the number of cores due to the added cost of find operations in the border merging step.



(a) g=1, 1 core

(b) g=4, 1 core

(c) g=1, 24 cores

(d) g=4, 24 cores

Figure 4.27: Cycles per pixels for SIMD Rosenfeld pixel (v1) and SIMD Rosenfeld sub-segment (v2), applied to 2048×2048 images.

threading	1-core mono thread			24-core multi thread		
granularity	g=1	g=2	g=4	g=1	g=2	g=4
LSL_{STD}	10.47	6.98	5.91	0.71	0.36	0.28
LSL_{RLE}	16.96	9.31	6.05	0.95	0.45	0.24
Rosenfeld	30.61	19.01	12.49	2.56	1.69	1.27
Rosenfeld+DT	20.01	11.81	8.30	1.95	1.51	1.09
SIMD Rosenfeld $v1_{512}$	**7.61**	**6.07**	**5.06**	0.84	0.52	0.50
SIMD Rosenfeld $v1_{256}$	10.65	8.64	7.08	0.99	0.57	0.55
SIMD Rosenfeld $v1_{128}$	16.40	13.23	11.33	1.26	0.76	0.71
SIMD Rosenfeld $v2_{512}$	7.97	6.54	5.26	**0.58**	**0.41**	**0.38**
SIMD Rosenfeld $v2_{256}$	11.54	9.22	7.35	0.74	0.53	0.44
SIMD Rosenfeld $v2_{128}$	17.89	14.15	11.89	1.07	0.71	0.66

Table 4.2: Average cycles per pixels for 2048×2048 images - best SIMD in bold (lower values are better).

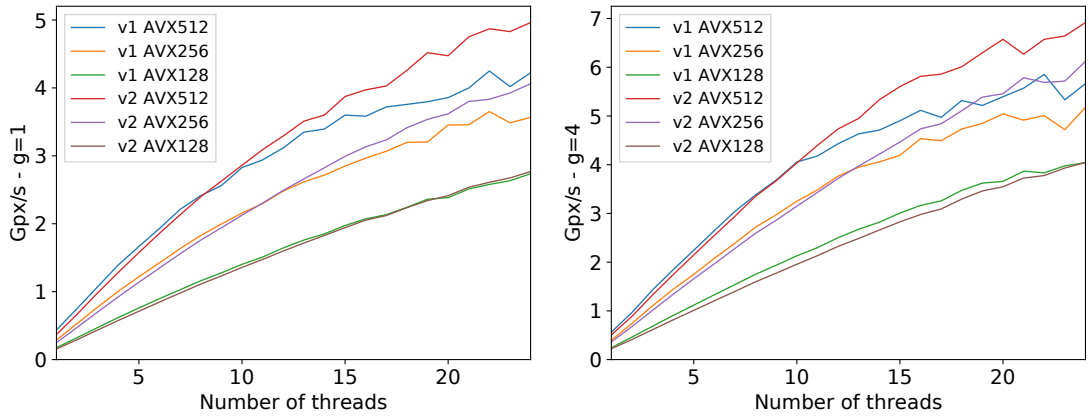
threading	1-core mono thread			24-core multi thread		
granularity	g=1	g=2	g=4	g=1	g=2	g=4
LSL_{STD}	0.27	0.38	0.44	4.12	7.53	9.45
LSL_{RLE}	0.21	0.32	0.46	3.56	6.91	11.80
Rosenfeld	0.13	0.17	0.24	1.15	1.70	2.17
Rosenfeld+DT	0.17	0.25	0.33	1.47	2.02	2.46
SIMD Rosenfeld $v1_{512}$	**0.44**	**0.49**	**0.56**	4.22	5.06	5.66
SIMD Rosenfeld $v1_{256}$	0.29	0.34	0.40	3.57	4.21	5.18
SIMD Rosenfeld $v1_{128}$	0.18	0.21	0.24	2.74	3.37	4.04
SIMD Rosenfeld $v2_{512}$	0.37	0.42	0.51	**4.96**	**5.84**	**6.92**
SIMD Rosenfeld $v2_{256}$	0.25	0.30	0.36	4.06	4.98	6.12
SIMD Rosenfeld $v2_{128}$	0.16	0.19	0.22	2.77	3.48	4.05

Table 4.3: Average throughput (Gpx/s) for 2048×2048 images - best SIMD in bold (higher is better).

SIMD vs scalar: In the single-threaded case, SIMD versions are two times faster than the Rosenfeld scalar versions. They are also faster than the LSL versions. In the multi-threaded case, this ratio grows to ×3. Compared to LSL, the SIMD versions are faster only for g=1, which is the worst case for full-segment labeling like LSL (the strategy to save memory accesses becomes more profitable than in the mono-threaded case where the pressure on the memory is lower, especially in the RLE version).

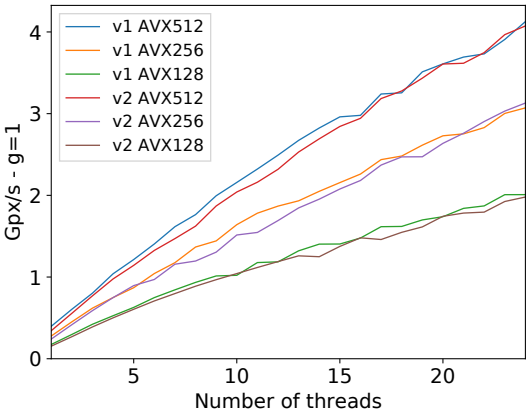
SIMD Scalability (128 / 256 / 512) In the single-threaded case, doubling the SIMD size provides a speedup of around 1.5. In the multi-threaded case, this ratio still exists but only for 128 / 256 registers. For 256 / 512 registers, the ratio drops to 1.2, due to bandwidth saturation and frequency scaling.

Thread scalability: Depending on the SIMD size, the scalability of the algo-

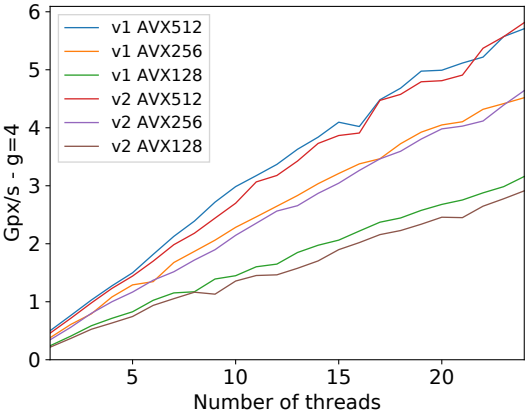


(a) g=1, 2048×2048

(d) g=4, 2048×2048



(b) g=1, 4096×4096



(e) g=4, 4096×4096

Figure 4.28: Average throughput (Gpx/s) for SIMD Rosenfeld pixel (v1) and SIMD Rosenfeld sub-segment (v2).

rithms varies in the interval $\times 10 - \times 13$ for 512-bit registers, and up to $\times 15 - \times 18$ for 128-bit registers. That is an efficiency ranging from 40% (for 512-bit registers) up to 75% (for 128-bit registers). Considering all the instructions for the control-flow and the label propagation within a register, for such an irregular algorithm, the results are acceptable.

Performance with image size: The best performance is achieved when the image fits in the cache for all algorithms. (see Table 4.4). For a granularity equal to 4, LSL_{RLE} is still the fastest algorithm. The new algorithms outperform Rosenfeld+DT by a factor $\times 2.3$ up to $\times 2.8$ for g=4, and from $\times 2.6$ up to $\times 3.4$ for g=1. For larger images, the pixel-based algorithm SIMD v1 becomes faster than the sub-segment-based algorithm SIMD v2. These two new algorithms are especially well-suited to very complex / un-structured / quasi-random images.

4.6 SparseCCL

In this section, I present SparseCCL, a parameterizable connected components labeling and analysis algorithm for sparse images. I then present a specialization of the algorithm in the context of the LHCb experiment, where the very few hits of high-energy particles are scattered across the detector's sensors.

image size	2k images		4k images		8k images	
granularity	g=1	g=4	g=1	g=4	g=1	g=4
LSL_{RLE}	3.56	**11.80**	3.45	**9.16**	3.25	**7.55**
Rosenfeld+DT	1.47	2.46	1.41	2.22	1.37	1.94
SIMD Rosenfeld $v1_{512}$	4.22	5.66	**4.13**	5.71	**3.83**	4.89
SIMD Rosenfeld $v2_{512}$	**4.96**	6.92	4.07	5.81	3.59	4.45

Table 4.4: Average throughput (Gpx/s) for 2k, 4k, 8k images on 24 cores (best performance in bold, for each column).

4.6.1 General parameterizable ordered SparseCCL

In this first version of the algorithm, the image is assumed to be represented by a list of active pixels ordered by their coordinates. This kind of representation allows the algorithm to take advantage of the sparse nature of the data. This is due to the size of the list scaling directly with the number of pixels to label and not the total number of pixels. Other cases when this representation is useful include when the image is too large or if the number of dimensions makes the storage impractical. Figure 4.29 gives an example of such an image and its list representation.

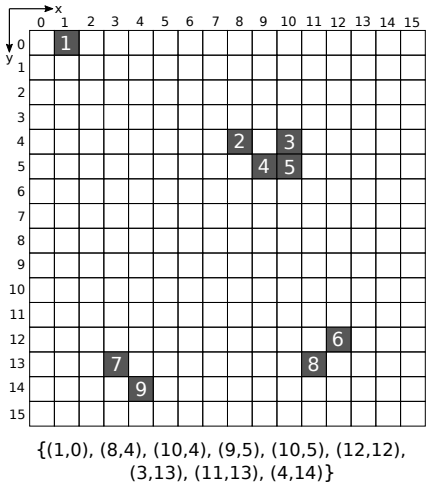


Figure 4.29: Sparse binary image and its list representation. Each entry in the list is a tuple of pixel coordinates (x, y). The list is sorted in column major order and contains $n = 9$ pixels (a density of 3.5%).

The algorithm parameterization is done through the functions `is_adjacent` / `is_far_enough` for the labeling and `init_features` / `accumulate_features` for the analysis. The algorithm is generic enough to be adapted to n-dimensions and every type of connectivity and pixel format. Algorithm 24 gives the complete algorithm.

SparseCCL is designed to minimize the memory footprint in order to have the best data locality and to fit in the L1 cache. The only internal memory it needs is an integer table of size n to store the equivalences. The input table containing the pixels and the output tables containing the connected components features are

Algorithm 24: SparseCCL

```
1  // First scan: pixel association
2  start_j ← 0
3  for i ← 0 to n − 1 do
4      T[i] ← i
5      ai ← i
6      for j ← start_j to i − 1 do
7          if is_adjacent(pixel[i], pixel[j]) then
8              ai ←union(T, ai, find(T, j))
9          else if is_far_enough(pixel[i], pixel[j]) then
10             start_j ← start_j + 1
11 // Second scan: transitive closure and analysis
12 labels ← 0
13 for i ← 0 to n − 1 do
14     if T[i] = i then
15         labels ← labels + 1
16         l ← labels
17         init_features(l, pixel[i])
18     else
19         l ← T[T[i]]
20         accumulate_features(l, pixel[i])
21     T[i] ← l
```

allocated outside of the algorithm. The algorithm is divided into two parts: the first scan where pixels are associated using an equivalence table, and the second scan where equivalences are resolved by performing a transitive closure of the graph embedded in the equivalence table. An `on-the-fly` analysis of connected components can also be added to the second scan.

The first scan iterates over the pixels in the list and adds them one by one to the equivalence table. The equivalence table is an index table implementing a forest of equivalence trees. Each cell of the table corresponds to one pixel; the content of the cell is the index of the parent pixel. A pixel is a root if its entry in the equivalence table is its own index. For each pixel, the algorithm checks on previously added pixels for adjacency and merges their two equivalence trees if they are adjacent. The merging is done by calling the `union` and `find` functions described in algorithms 4 and 3. Because the list is ordered, the algorithm can keep track of a start index to avoid testing pixels that are too far from each other. With this optimization, the first scan complexity becomes $O(kn)$ instead of $O(n(n-1)/2)$, with k a constant much smaller than n, as only the few last pixels are tested. The `is_adjacent` and `is_far_enough` parameterization for 2-dimensions 8-connectivity labeling is described in Algorithms 25 and 26. Here, the adjacency is tested by comparing the L1 distance between two pixels to a radius of 1; the pixels are far enough if the signed distance is bigger than the radius.

Algorithm 25: is_adjacent(p_1, p_2)

```
1  return |p1.x − p2.x| ≤ 1 and |p1.y − p2.y| ≤ 1
```

Algorithm 26: is_far_enough(p_1, p_2)

```
1 return p1.y − p2.y > 1
```

The second scan iterates over each temporary label in the equivalence table. If the label is a root, it creates a new connected component label by incrementing the *labels* counter and initialises the features for this connected component. If the label has a parent, it takes the parent's label and accumulates the features. Because the temporary label of a parent is always smaller than the one of the child, the parent is always already processed. Before continuing to the next label, the equivalence table is updated with the new label. Algorithms 27 and 28 give an example of the features needed to compute the connected components center of gravity $(G_x, G_y) = (S_x/S, S_y/S)$. (S_x, S_y) are the sums of x and y coordinates and S the number of pixels.

Algorithm 27: init_features(label, pixel)

```
1 sum_x[label] ← pixel.x
2 sum_y[label] ← pixel.y
3 sum_n[label] ← 1
```

Algorithm 28: accumulate_features(label, pixel)

```
1 sum_x[label] ← sum_x[label] + pixel.x
2 sum_y[label] ← sum_y[label] + pixel.y
3 sum_n[label] ← sum_n[label] + 1
```

4.6.2 Acceleration structure for un-ordered pixels

In some scenarios, the input list of pixels might not be ordered and even sorting it would take too much time. Furthermore, accessing a full pixel image buffer cannot be afforded because of the poor data locality and the time it would take to reset such a buffer between two images labeling. A compromise can be made by doing dimension reduction: a table is used for each row and pixels are added to their corresponding row's table when they are encountered in the first scan. Now, when checking for adjacency, only the previous, current and next row of the table have to be checked. The pixels within each row are not sorted so they all have to be checked. Each row has a size property (N_{row}) that keeps track of how many pixels were added to the row. When resetting the tables, we only have to set the size of used rows to zero. Table 4.5 shows an example of such structure.

4.6.3 Case study: specialization for LHCb VELO Upgrade

In Chapter 1, I presented the LHCb sub-detectors. Before particle trajectories can be reconstructed, the raw data from each sub-detector must be decoded and prepared. The VELO sub-detector is divided into 52 L-shaped modules. Each module is itself composed by 4 sensors of 3 chips each. The chips have 256×256 pixels, so

Row	N_{row}	Pixels (index, column)
0	1	(5, 1)
1..3	0	
4	2	(9, 8), (7, 10)
5	2	(4, 9), (1, 10)
6..11	0	
12	1	(8, 12)
13	2	(3, 3), (6, 11)
14	1	(2, 4)
15	0	

Table 4.5: Structure representing the received pixels from Figure 4.29.

the sensors have 256 rows and 768 columns. Each pixel is a square with a length of 55 microns. The sensor pixels are packed into Super-Pixels (SP) of size 2×4 pixels, so the sensors have 64 SP rows and 384 SP columns [81][140]. This section presents a specialization of the SparseCCL algorithm for the preparation of the VELO data; the next chapter presents a reconstruction algorithm that uses the clusters prepared by this algorithm.

Figure 4.30 shows the format of a Super-Pixel (SP) encoded in a 32-bit integer. The less significant byte is a bitmask representing the pixels. The row of the SP is found from bit 8 to 13 and the column of the SP is found from bit 14 to 22. The 31^{st} bit is a flag indicating if the SP is isolated, i.e. if it doesn't have any neighbours. The SPs are delivered in small packets of bits called raw banks. There is one raw bank per sensor and each one contains the number of SPs in the bank followed by the encoded SP.

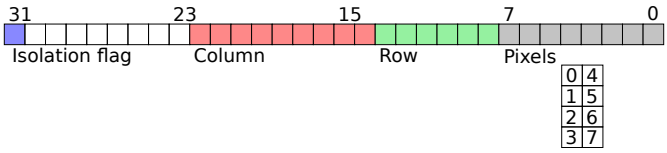


Figure 4.30: CERN LHCb VELO Super-pixel format as represented in a 32-bit integer. The 8 less significant bits are the values of the 2×4 pixels block, as depicted on the bottom right.

To take advantage of the VELO data format, the SparseCCL algorithm was specialized to label SPs instead of pixels. This allows to further reduce the amount of memory needed and to skip a decoding step. The first step is to prepare the data: the SPs that are known to be isolated are removed and resolved using lookup tables. The remaining SPs are tested to check if there is more than one CC inside and split them if necessary. Figure 4.31 shows the two possible configurations for an SP: one CC or two CCs. Because there can be at most 2 clusters per SP, the maximum number of clusters in the image is 2× the number of SPs. Once the SP list is prepared, the algorithm is run using a combination of bitwise operations and a lookup

table to test the adjacency. Another lookup table is used for a fast computation of the first statistical moment and the number of pixels within an SP.

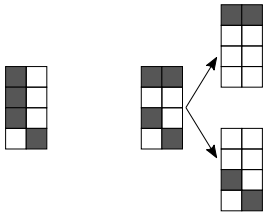


Figure 4.31: A Super-Pixel containing one CC (left) and a Super-Pixel containing two CCs, split in two Super-Pixels (right).

In 8-connectivity CCL there are eight directions of adjacency, but by using symmetries their number can be reduced to four. Figure 4.32 shows the configurations of SPs and the pixels we have to test. Configurations a, b and c are quickly tested using only bitwise operations. While configuration d could be tested the same way, it was found faster to use a 256-entry lookup table using the dark pixels bit pattern as the address. Configuration e shows the pixels required to take the decision to split the SP in two.

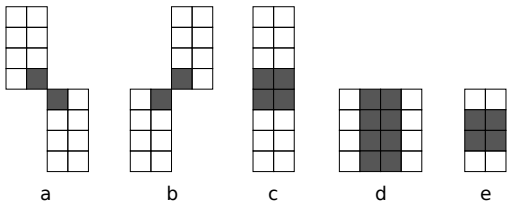


Figure 4.32: a. Diagonal "forward" link, b. Diagonal "backward" link, c. Vertical link, d. Horizontal link, e. Two clusters condition.

4.6.4 Experimental Evaluation

Evaluating CCL algorithms has always been a challenge as their speed is data-dependent. To model natural images, pseudo-random noise images of varying densities and granularity are generated, following [29]. The density parameter controls, at low density, the number of pixels in the image. GCC 8.2 was used with level 3 optimisation level for all algorithms.

In the case of LHCb's VELO detector, simulation data have shown that densities of hits in the sensors are very low and granularity is around 2. This is due to charge sharing in the silicon, when a particle hits the border between 2 or 4 pixels.

Table 4.6 shows the time needed in microseconds to process one image of 768×256 pixels, for state-of-the-art dense CCL algorithms [29] and sparse CCL algorithms. While these algorithms are well optimized, a simple flood fill looking only at active

pixels is 10 times faster at a density of 1%.

	0%	1%
LSL(dense) [32]	235.3	319.9
Rosenfeld+DT (dense)	270.3	315.4
Rosenfeld AVX512 [80]	276.5	303.3
Flood fill (sparse)	0.0	29.9
SparseCCL (ordered)	0.0	31.5
SparseCCL (Super-pixels)	0.0	25.0
SparseCCL (row table)	0.0	17.4
SparseCCL (row table, AVX512)	0.0	19.9

Table 4.6: Processing time of 768×256 pixels images – in microseconds – of dense and sparse algorithms at granularity $g = 1$, on an Intel Xeon Gold 6126 @2.6GHz.

This benchmark shows the time measured in cycles per pixel (cpp). Normalizing by the number of active pixels allows us to see the real impact of the increasing density: the more the connection of pixels impacts the speed, the bigger the slope of the plot will be. Normalizing by the frequency of the machine allows to abstract the frequency of the CPU for a better comparison of architectures.

Four algorithms are evaluated. The first is a flood fill algorithm as described in Section 4.2.1. While the flood fill is generally inefficient on dense images, it was found that it outperforms fast implementations of iterative and two-pass algorithms on sparse images. This is due to its ability to use the pixel list information as a starting point for its connected component mapping. The other three algorithms evaluated are variants of the SparseCCL algorithm. The ordered by row variant is the simple case where the input is an ordered list of single pixels. The row table variant is the algorithm described in Section 4.6.2 that can take an unordered list of single pixels as input. The last variant is the specialization of the algorithm for Super-Pixel encoding as described in Section 4.6.3 where the list of Super-Pixels is assumed to be ordered.

Figure 4.33 shows the measured time in cpp for the four algorithms, for 364×768 pixels images of varying densities from 0% to 2.5%. For comparison, the average density in a simulated VELO sensor for Run 3 is less than 0.1%. The number of pixels n is given by the formula $n = \frac{d}{100} \times w \times h$, where d is the density, w the number of columns and h the number of rows. In the test configuration, the number of pixels ranges from 0 to 6988. For a granularity of 1, the ordered SparseCCL working on pixels has the best behavior at low density. The Super-Pixels variant is slower at low densities because each SP is more likely to contain only 1 pixel. It presents no advantage over the pixel versions. The row table variant starts higher than the ordered one because of the cost of table reset. The flood fill algorithm is significantly slower than other version at low density, but scales better with the number of pixels and eventually becomes faster for densities $> 1.8\%$. The benchmarks were conducted on an Intel Xeon Gold 6162 @2.6GHz (in Figure 4.33) and on an AMD EPYC 7301 @2.2GHz. The cpp (time normalized by frequency in cycle per pixel)

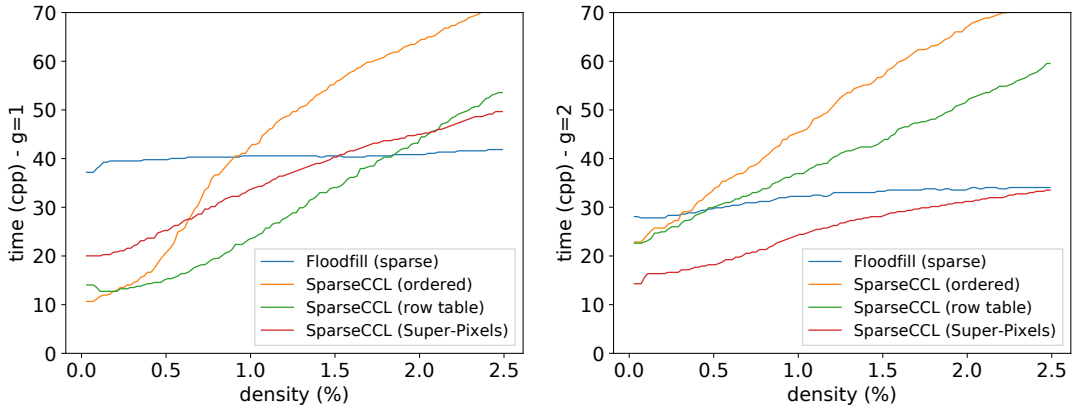


Figure 4.33: Cycles Per Pixel (cpp) for the four algorithms depending on the density of the image, with a granularity g=1 and g=2.

were similar on both architectures. With a granularity of 2, the flood fill algorithm benefits from the data locality induced by the increased granularity. On the contrary, the pixel-based SparseCCL variants are slowed down by it, as the number of tests they have to perform is slightly higher. Thanks to the Super-Pixel encoding, the last variant of SparseCCL specialized for the LHCb experiment is sped up.

4.7 Conclusion

The CCL and CCA algorithms developed in the context of this thesis and presented in this chapter contributed to advance the state-of-art. All the algorithms have been published in international conferences: [112, 111, 80, 77, 76, 113, 78, 79]. While the dense algorithms were outperformed by the SparseCCL algorithm in the context of LHCb VELO data preparation, they found their use in the Meteorix project [144, 42, 126, 44] as part of its processing chain. The Meteorix project aims at detecting meteors in real-time from a nano-satellite. Due to communication budget, all processing must be done on-board within a tiny 7 Watt power envelope.

Chapter 5

VELO reconstruction algorithm

Contents

5.1 Introduction

This chapter presents a new VELO reconstruction algorithm developed especially for Run 3 in the context of this thesis. This algorithm builds upon a rich history of VELO detectors and reconstruction algorithms that started in Run 1 for offline reconstruction and which were optimized over the years. The contribution presented here uses SIMD of modern CPUs to increase the throughput, as well as a new hit pair selection strategy that helps the algorithm to be more regular and scale better with the detector occupancy. Finally, the algorithm is evaluated on high ends x86 systems and compared with the CPU state-of-the-art VELO reconstruction algorithms.

5.2 Tracking algorithms

Charged particles leave energy deposits when they traverse a detector. A "hit" corresponds to an energy deposit in a single physical detector element. For tracking detectors these elements might be a single pixel, a single silicon strip, or a single scintillating fibre. Hits are the most basic raw input data of a tracking algorithm. Depending on the type of detector and its spatial granularity, a particle may leave multiple contiguous hits when traversing a single detector element. In this case these contiguous hits are grouped into connected components, usually called "clusters", and the clusters are given as input to the tracking algorithm. In the LHCb VELO tracking algorithm, the SparseCCL algorithm, presented in the previous chapter, is used. For simplicity, "hit" will be used to describe the prepared input data throughout this chapter.

A typical tracking algorithm consists of three logical elements: clustering, pattern recognition, and track fitting. Clustering is a well known problem in computer vision, where it is referred to as connected component labeling. Pattern recognition consists of choosing a subset of hits which correspond to a single particle traversing a detector. Track fitting consists of finding a trajectory which most accurately represents the path taken by the particle while leaving these hits. An accurate track fit allows the particle's trajectory to be extrapolated beyond the region in which it left hits, and is therefore vital for precisely determining the origin and momentum of the particle. The quality of the track fit can also be used to discriminate between genuine and fake tracks. For this reason many tracking algorithms perform a partial fit during the pattern recognition stage and use its quality to reject the worst track candidates as early as possible.

To evaluate the quality of the pattern recognition, reconstructed tracks of simulated data are compared to the set of reconstructible tracks from the ground truth given by the Monte-Carlo simulation [153, 11, 14, 22]. For the VELO, a particle is considered reconstructible if it leaves at least 3 hits in the detector. A correctly reconstructed track is defined as one that has more than 70% of its hits created by a single true particle. If more than one track candidate is matched to the same true particle, it is referred to as a *clone*. A track candidate that could not be matched to any particle is called a *fake* or "ghost" track. The efficiency, the clone rate and the fake rate are defined as follow:

$$\text{efficiency} = \frac{|\{\text{reconstructed}\}|}{|\{\text{reconstructible}\}|}$$

$$\text{clone rate} = \frac{|\{\text{clones}\}|}{|\{\text{clones}\} \cup \{\text{reconstructed}\}|}$$

$$\text{fake rate} = \frac{|\{\text{fakes}\}|}{|\{\text{fakes}\} \cup \{\text{reconstructed}\}|}$$

A good pattern recognition algorithm should have the highest efficiency and the lowest clone and fake rates possible. As the efficiency only accounts for the track being found but not the quality of the reconstructed tracks, the hit efficiency is defined

as the fraction of hits from a true particle included in the reconstructed track. This metric should be as high as possible and is a good indicator of the quality of the reconstructed tracks. Not all track hits are equally important from a physics point of view. In particular, missing the first hit on the VELO track worsens the resolution on the track's physics parameters far more than missing a hit in the middle of the track. Similarly, missing the last hit on the VELO track worsens the resolution for extrapolating the track to the rest of the LHCb detector.

Once the track candidates are found, a Kalman fit is performed on the hits in order to define the state closest to the beamline and the state at the end of the VELO (z = 770mm). A state consists of the slope of the track (t_x, t_y) at a given set of x, y, z coordinates, and the associated covariance matrix of their uncertainties. Due to multiple scattering, it is expected that the two produced states are slightly different, even in the absence of a strong magnetic field. We define the χ^2 of the track fit as:

$$\chi^2 = \sum_{h=(x,y,z)}^{\{hits\}} (x_0 + h_z t_x - h_x)^2 + (y_0 + h_z t_y - h_y)^2$$

The χ^2 can later be used as a quality metric to remove fake tracks or be included in the vertex χ^2 computation.

5.3 Evolution of the VELO detector and algorithms

The purpose of the Vertex Locator (VELO) detector, located around the interaction region, is to precisely reconstruct the locations of the Primary Vertices (PVs) and separate tracks produced directly in PVs from tracks produced by particles which decay inside the VELO but away from the PVs. For this reason, the reconstruction of tracks in the VELO is the first step of LHCb's overall detector reconstruction. As there is almost no magnetic field inside the VELO, charged particles traverse it in almost straight lines. In this section, I will present the history of the VELO reconstruction both for the current and the upgraded LHCb detector, in order to place the results presented in the rest of this chapter in their proper context.

5.3.1 Reconstruction in the Run 1 and 2 VELO detector

The VELO tracking algorithm used during the first LHC data taking period (Run 1) was developed in 2002 [33]. At that time, the VELO detector geometry was using strips along ϕ, the radial angle around the beamline, and **R**, the distance to the beamline, as shown in Figure 5.1. Consequently, the tracking was done in two steps. First, a 2D tracking was performed in the **R-z** projection where it was easy to find interesting tracks based on the slope and alignments, then a "space" tracking (3D) step matched these **R-z** track candidates to hits on the ϕ strips.

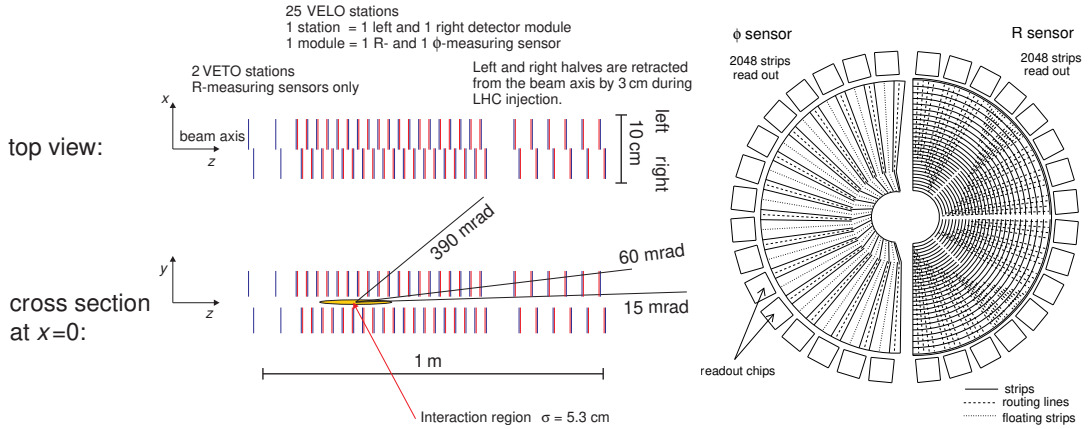


Figure 5.1: VELO (Silicon strip detector) Geometry. Figures from [115], the "beam axis" is what we refer to as beamline in the rest of the paper. The right hand diagram has an example of a ϕ sensor on the left and an **R** sensor on the right. The radius of the detectors is 45 mm.

The 2D tracking began by finding a triplet of aligned hits in three consecutive **R** sensors, then extended it as much as possible by predicting the radius in the next sensors and finding the closest hit. To avoid finding the same track again, used hits were marked and not considered in later searches. To avoid missing tracks due to sensor inefficiencies, the algorithm was allowed to skip one sensor in this search. The algorithm processed the **R** sensors in a single step going toward the interaction region. Due to the criteria applied on the slope and the single pass, no backward tracks were reconstructed. Subsequently, the space tracking was performed for every **R-z** track candidate. A second **R-z** tracking was then performed in the reverse direction to find the backward tracks, which are particularly useful for finding primary vertices. After matching the **R-z** track candidates to hits on the ϕ strips, the best track candidate was selected. The candidate with the highest total number of hits was selected, and the candidate with the best track fit χ^2 in case of equality.

In 2004, the algorithm was updated [34] to fulfill the speed and efficiency requirements of the real-time reconstruction for the different LHCb trigger stages. The changes mainly concerned the tuning of tolerances and search windows. It was noted that large search windows were needed in the track extension step to allow the recovery of tracks not pointing to the beam-line, for which the **R-z** projection is not exactly a straight line, and low momentum tracks with large multiple scattering. In 2007, further tuning and analysis of the algorithm were performed [87].

In 2011, a new implementation of the algorithm was introduced [35], motivated by LHCb's choice to run at twice the design instantaneous luminosity. The consequently higher number of proton-proton collisions per bunch crossing and detector occupancy, required the reconstruction to be optimized again in order to fit into the constraints of LHCb's real-time data processing. In addition, during the 2010 run it was found that the VELO could not come as close to the LHC beamline as expected. Because of this, the search for **R-z** track candidates introduced a further inefficiency for tracks produced away from the beamline. The 2011 algorithm modified the **R-z**

tracking to first search for quadruplets of hits, then triplets of hits among the remaining unused hits. This approach allowed to reduce the fake track rate and sped the algorithm up. The rest of the algorithm remained very similar to the previous implementations. In 2015, a measurement of the reconstruction efficiency of the VELO tracking was published [118].

5.3.2 Reconstruction of the upgraded VELO detector

In addition to the triggerless readout, one of the major changes in the upgraded LHCb detector is the total replacement of the VELO. The detector technology was changed from silicon strips to silicon pixels [165] in order to significantly improve spatial granularity and physics performance even in the higher occupancy conditions of the upgraded LHCb detector. The geometry of the new VELO-PIX detector, presented in Chapter 1, has a broadly similar coverage to the old **R-z** Velo.

In 2009, the simulation framework started supporting the new detector and in 2012 the first version of the pixel VELO tracking algorithm was implemented [24]. In the pixel version, the input of the tracking algorithm are the 3D Cartesian coordinates of the reconstructed hits on each pixel plane. Similar to the previous VELO Tracking algorithm, the tracks are created by looking for pairs of unused hits whose estimated track slope would be compatible with the geometric acceptance of the other LHCb detector components. Subsequently, the track candidates are extended upstream (smaller **z**-position) by extrapolating and looking for the closest hits within a search window. A cut on the maximal scattering angle is added and the search is abandoned if no hits are found on three consecutive stations. Three-hit tracks are kept only if all their hits are unused and their χ^2 is below a parametrizable threshold. A detailed description of the algorithm and its performance was given in the VELO Upgrade TDR [117].

A study was conducted to use vertical vectorization with 128-bit SSE SIMD extension to accelerate part of the algorithm [158] but resulted in a slowdown attributed by the authors to the need for data preparation to take advantage of SIMD loads and stores. At that time, alternative global methods based on the Hough Transform [86] and suitable for parallel architectures were also evaluated [158, 147, 8]. In 2014, a new local search algorithm based on triplet seeding on GPU was presented [18]. The main difference with the previous sequential work was that tracks were seeded and extended upstream independently, in parallel. A post-processing step cleaned ghosts. This preliminary work was further improved in [19].

In 2018, the CPU pixel tracking algorithm was made faster in order to improve the throughput of the first stage of LHCb's real-time reconstruction [49]. The hits were ordered by ϕ and the hit search performed within ϕ-windows, the search for backward and forward tracks was split in two different steps and some "speed-flags" were introduced, allowing for an early rejection of less important track candidates. In 2019, the search by triplet algorithm was revisited for parallel architectures in the context of the Allen project [36, 47, 7]. This new implementation used the ϕ-windows to reduce the combinatorics and uses synchronization between each layer

to avoid track overlap.

5.4 SIMD Velo reconstruction

This section describes the reconstruction algorithm developed for LHCb's VELO detector. This algorithm was designed to take advantage of the SIMD capabilities of modern CPUs and is implemented using the SIMDWrappers library presented in Chapter 2.

5.4.1 Structure of the algorithm

The reconstruction algorithm implements a local search approach based on track following, similar to the previous VELO pattern recognition algorithms described earlier. It consists of two alternative steps: seeding and extending. In the seeding step, new candidates are created from a triplet of hits. In the extending step, existing candidates are extended into the next layer and hits are tested to be added to the candidate. To take advantage of track parallelism with SIMD, the algorithm is structured like the search by triplet algorithm [36] with multiple track candidates being processed simultaneously. Thanks to the synchronization between vector elements being implicit on a CPU, no explicit synchronisation between layers is needed. Each layer's hits are prepared on demand and stored in a small container with an SoA layout. Three layers are needed at a time, so only three containers are allocated on the stack and pointers are rotated to recycle them while moving through all VELO layers. This allows to reduce the algorithm memory footprint and to be more cache-friendly by improving data locality. Two track containers are used to memorize the track candidates created by the seeding and the track extending steps. As in previous algorithms the extending step allows for a layer to be skipped: if no hit is found when extending a track candidate, it is still propagated to the tracks candidates of the next step, but if this happen twice in a row, the candidate is finalized and, if it fulfill the minimal number of hits to be accepted, moved to the output tracks container. It is not uncommon for a track to not leave a hit in a layer, in [36], the authors estimated the probability to 1%, but for it to happen twice in a row the probability is squared. Figure 5.2 shows the data flow within the algorithm.

While this standard alternating combinatorics scheme offer the best compromise between speed and efficiency, other schemes were evaluated. A simple modification of the algorithm is to process each halves separately, this reduces the number of hits per layer and increases the throughput of the algorithm by about 8%. However, it splits the tracks going through both halves, which increases the clone rate and may lower the efficiency if splitting a track results in too few hits in each part. This strategy is not used by LHCb but the implementation was kept as a possible mitigation in case more speed is needed.

One limitation of the standard strategy is its inability to add two hits of the same layer to the same track. In this strategy, a layer is composed of modules from both halves of the VELO, which have a slightly different z coordinate allowing a tiny overlap when the VELO is fully closed. This overlap is sufficient for a track with the

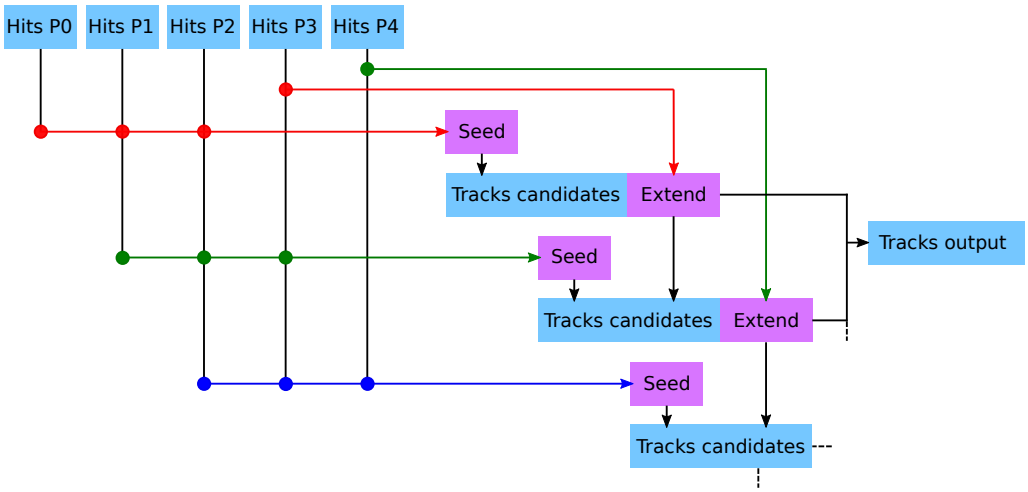


Figure 5.2: Data flow within the algorithm. The hits are taken from the input planes P0 to P26, three by three, and processed in the seeding step to produce tracks candidates, which are then extended with the hits from the next layer. The candidates that could not have been extended are then copied in the tracks output container.

right angle and position to cross both modules of the same layer. While this event is rare, it is interesting for the alignment process as it allows to refine the relative position of the two halves. In order to reconstruct those tracks with all their hits, each module is considered as a separated layer, bringing the number of layers to 52. To retain the high efficiency of the standard strategy, each step must be allowed to skip more layers. The extending step is modified to be allowed to skip three layers instead of one. The seeding step is still operating on three layers, but multiple seeding are performed for each step, allowing at most one layer between each seeding layers. The seeding steps are performed sequentially from the most spaced seeding to the consecutive seeding. This new scheme requires at most five hit containers per step instead of three. Figure 5.3 shows the iteration scheme for the "full" strategy. This strategy has about 12% less throughput than the standard strategy, due to the increased combinatorics, making it inadequate for HLT application, but acceptable for the alignment software that runs at a much lower rate.

5.4.2 Seeding tracks

The track seeding is the most compute intensive part of the algorithm. In a typical upgrade event, one VELO layer contains an average of ∼100 hits. Testing all possible triplet combination would require $O(100^3)$ tests. Previous algorithms reduced the combinatorics by relying on the VELO detector geometry. Since the VELO detector is centered around the LHC beamline, it defines a cylindrical coordinate system with the beamline as its axis. Since most tracks produced in LHC beam crossings come from the beamline, they traverse lines of constant ϕ. Hit-pair candidates can therefore be built by selecting one hit with a given ϕ and then finding all hits on a given second layer which are within a certain ϕ-distance with respect to the first hit.

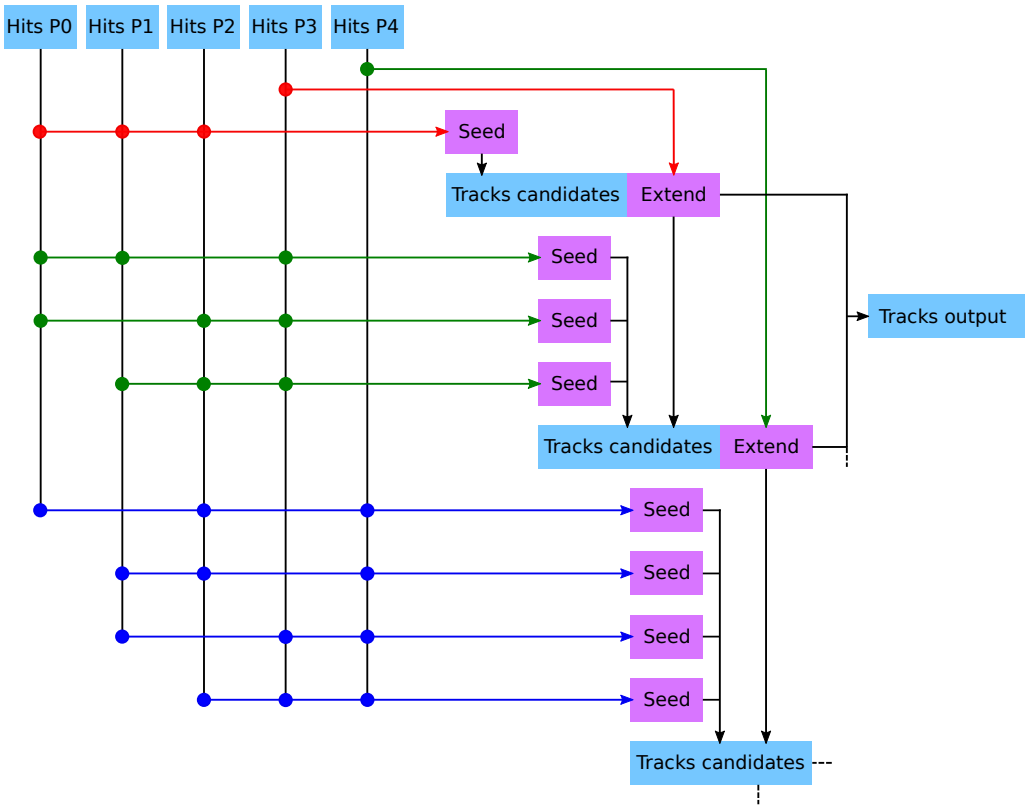


Figure 5.3: Alternative "full" VELO reconstruction. Instead of considering the two halves as part of the same plane, this version has the left half's sensors in even planes and the right half's sensors in odd planes. In order to maintain the correctness of the algorithm, the seeding step must allow to skip one plane, leading to an increased combinatorics.

This approach of state-of-the-art algorithms works well for the majority of tracks that come from the LHC beam line, but requires large ϕ tolerances to accept tracks produced away from the beamline, such as the products of particle decays, or with some multiple-scattering in the RF-foil. Because a lot of LHCb's physics is based on displaced tracks it is important to boost their reconstruction efficiency, even if they represent only a tiny fraction of reconstructible tracks. As shown on the left part of Figure 5.4, a ϕ-window of $\pm 3°$ allows to correctly match 95.56% of hits by looking at a maximum of 10 hits (2.6 in average), while a ϕ-window of $\pm 20°$ is able to build 98.77% of reconstructible pairs at the cost of having to test up to 30 candidates (10.8 in average). Apart from increasing the combinatorics, having a large candidate count increase the probability of generating fake tracks, and for these reasons we want to keep it as low as possible. Also, while having a variable number of candidates can be advantageous on a sequential architecture, a parallel architecture has to synchronize between processing elements so the time for all elements to finish is always the maximum of all elements' time.

Instead of using a ϕ-window, a nearest in ϕ approach where we pick a fixed number of candidates N has been implemented. As shown on the right part of Figure 5.4, 96.59% of hits can be matched with only 3 candidates or 98.89% of hits with 10 candidates, reducing by a factor 3 the maximum number of candidates processed.

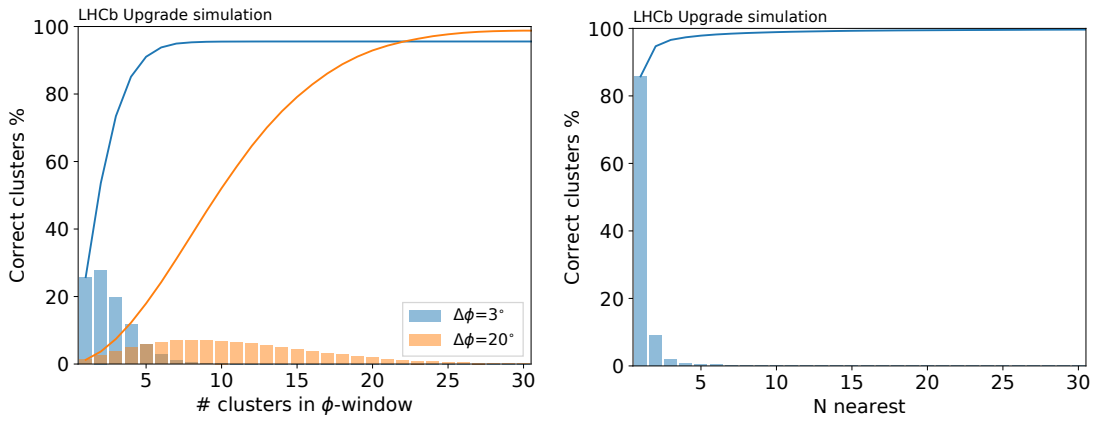


Figure 5.4: On the left, the % of correctly matched hits for the ϕ-window algorithms, depending on the number of hit candidates (3° window is used in the "Fast" configuration of the Search by Pair algorithm and 20° window is used in the "Best" configuration). On the right, the % of correctly matched hits for different number of candidates from our SIMD algorithm. These statistics are averaged on 100 Monte-Carlo simulated events, considering only the track seeding part of the algorithms and without marking used hits.

This allows the SIMD algorithm to be more regular and less data dependent leading to a better utilisation of SIMD processing units. As the number of candidates N is small, they can fit in registers, avoiding costly memory accesses. The candidate positions and indices are stored in an N-sized array of SIMD register types. As the number of candidates N is known at compile time, the compiler is able to fully unroll the loops (loop unwinding) over the candidates and the array to registers, as shown in Listing 4. If N is too large to fit all candidates in registers, the compiler have to place them in memory which adds extra costs to load and store them. By excluding used hits when looking for the N nearest candidates, is allows more distant hits to be tested if the closest ones are already used by another track. This helps to reconstruct displaced tracks in very dense layers. The limitation of processing multiple tracks at a time is that it allows the tracks to share some hits, if two track of the same vector match the same hit simultaneously, potentially leading to clones. If the number of tracks processed in parallel is small, it doesn't have a big impact on the tracking efficiencies and clone rate. Because the hit container is not ordered the probability of sharing a hit among tracks processed in the same SIMD register is decreased, as tracks that have different ϕ angles are unlikely to share hits. For larger SIMD registers widths, the conflict detection instructions available in AVX-512 can be used to remove the clones before propagating them. However, as the clone rate was already low, it was not necessary.

Once the initial pair candidates are built, they are extended in the third layer to search for the hit minimizing the Euclidean distance to the linearly extrapolated position. In this step, all the hits are tested, without testing for the ϕ distance. As in previous algorithms, this Euclidean distance is called the *scattering parameter* and the triplet candidate, built from the pair and the best hit, that minimizes the scattering is kept. The triplet is accepted and added to the track candidate container if its scattering is lower than a configurable threshold value called `max_scatter_seeding`.

```
// Code written with array and loop:
const int N = 3; // constant value known at compile time
float_v arrayOfCandidates[N]; // array of SIMD vectors

for (int i = 0; i < N; i++) { // loop over candidates
  process(arrayOfCandidates[i]);
}

// Code transformed by the compiler:
// unwinded array
float_v candidate_0; // each variable is assigned to a register
float_v candidate_1;
float_v candidate_2;

// unwinded loop over candidates
process(candidate_0);
process(candidate_1);
process(candidate_2);
```

Listing 4: Loop and array unwinding example

When a hit is used in a track, it is removed from the layer's hit container.

It is worth noting that the complexity of the seeding algorithm using a ϕ window scales with the cube of the layer's hit occupancy, while the approach based on the N nearest hits scales with the square of the occupancy thanks to the constant number of pair candidates per initial hit. This property makes this new approach promising for Upgrade II conditions where the occupancy will be much greater.

5.4.3 Extending tracks

After the seeding, the second step of an iteration of the tracking algorithm consists of successively finding hits in the subsequent VELO layers to extend the track candidates. Track candidates are processed in parallel and all non-used hits are tested using the same scattering criteria as in the seeding. The best hit is kept if the scattering is less than the `max_scatter_extending` threshold. If a track miss a hit in one layer it would be split in two, reducing the hit efficiency of both parts and increasing the number of clones. To prevent this, the track candidates not matched with a hit are kept for one more iteration, allowing for one layer to be skipped. The skipped layer counter is reset each time a hit is found. If a track candidate misses two hits in a row, its quality is evaluated to determine if it should be moved to the output tracks container or discarded. The candidate is kept if it contains more than three hits, or if the sum of the scatterings is less than a threshold `max_scatter_3hits`. This threshold for three-hit tracks is more restrictive than the threshold for longer tracks, to limit the number of fake short tracks. The partitioning of track candidates into next track candidates or track output is done using the compress-store pattern described in Chapter 2. Figure 5.5 shows an example of track seeding and extension.

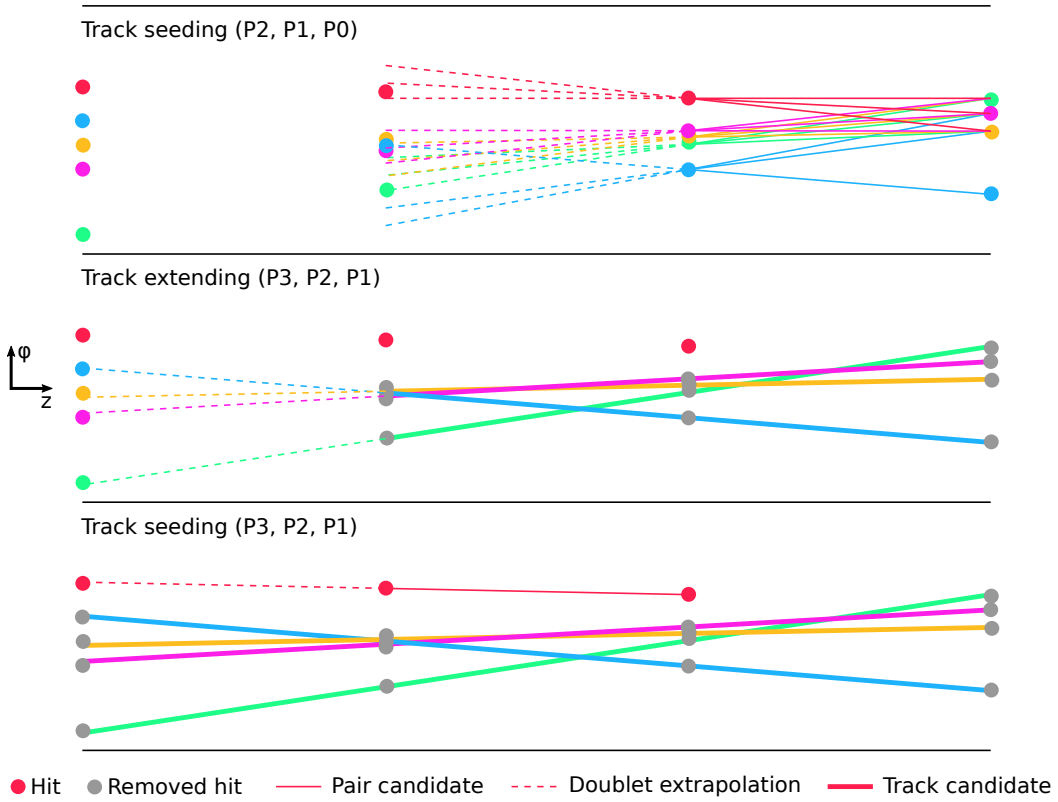


Figure 5.5: The first seeding considers every hit in P1, builds 3 pair candidates with the nearest P0 hits in ϕ and extrapolates the doublet in P2 to find the P2 hit that minimizes the scattering. The used hits are removed, then every track candidate is extrapolated in P3 and extended if a hit is found. The first cycle is complete, and the algorithm performs another seeding in (P3, P2, P1), before continuing. Hits are associated from right to left.

5.4.4 Numerical precision

The VELO reconstruction algorithm itself doesn't suffer from numerical precision issues. The few computation involved in the algorithm consists of simple additions and multiplications of numbers of similar magnitude. While historically double have been used to represents hit coordinates, it was long ago changed to single-precision floating point to save on memory. Recent studies conducted on the GPU Search by Triplet algorithm [48] have shown that even half-precision can be used without noticeable impact on physics performances. Modern x86 instruction sets doesn't yet support FP16 for computation but as SIMDWrapper supports ARM architectures, it could still be tried with a few modifications to the library.

However, after the reconstruction algorithm, the candidate tracks are fitted with a Kalman filter algorithm that involve many floating point computations. The original algorithm was written for double-precision and later changed to single-precision. In the context of this thesis, the filter algorithm was converted to use SIMD and a floating-point absorption issue was noticed and fixed. The filter is parallelized the trivial way, over tracks, processing 8 tracks in parallel using AVX2. To keep the algorithm simple, the track state is separated into two states that

are filtered independently. Together with their respective covariance matrices the two states are (X, T_x, $cov(X,X)$, $cov(X,T_x)$, $cov(T_x,T_x)$) and (Y, T_y, $cov(Y,Y)$, $cov(Y,T_y)$, $cov(T_y,T_y)$). The update equations of the original algorithm were as follow:

$$
\begin{aligned}
pred\_X &= X + dz \cdot T_x && \text{predictions} \\
pred\_cov(T_x,T_x) &= cov(T_x,T_x) \\
pred\_cov(X,T_x) &= cov(X,T_x) + dz \cdot cov(T_x,T_x) \\
pred\_cov(X,X) &= cov(X,X) + 2dz \cdot cov(X,T_x) + dz^2 \cdot cov(T_x,T_x)
\end{aligned}
$$

$$
\begin{aligned}
K_x &= \frac{pred\_cov(X,X)}{hit_{weight} + pred\_cov(X,X)} && \text{gains} \\
K_{T_x} &= \frac{pred\_cov(X,T_x)}{hit_{weight} + pred\_cov(X,X)}
\end{aligned}
$$

$$
\begin{aligned}
X &= pred\_X + K_x * (hit_x - pred\_X) && \text{updates} \\
T_x &= T_x + K_{T_x} * (hit_x - pred\_X) \\
cov(T_x,T_x) &= pred\_cov(T_x,T_x) - K_{T_x} \cdot pred\_cov(X,T_x) \\
cov(X,T_x) &= (1 - K_x) \cdot pred\_cov(X,T_x) \\
cov(X,X) &= (1 - K_x) \cdot pred\_cov(X,X)
\end{aligned}
$$

In the above equations, dz is the distance, in millimeters, between the current and previous hit, along the z axis. The first issue lies in the (T_x, T_x) covariance update, which can be rewritten as:

$$
\begin{aligned}
cov(T_x,T_x) &= pred\_cov(T_x,T_x) - K_{T_x} \cdot pred\_cov(X,T_x) \\
&= cov(T_x,T_x) - \frac{(cov(X,T_x) + dz \cdot cov(T_x,T_x))^2}{hit_{weight} + pred\_cov(X,X)} \\
&= \frac{cov(T_x,T_x) \cdot (hit_{weight} + cov(X,X)) - cov(X,T_x)^2}{hit_{weight} + pred\_cov(X,X)} \\
&\quad + \frac{dz^2 \cdot cov(T_x,T_x)^2 - dz^2 \cdot cov(T_x,T_x)^2}{hit_{weight} + pred\_cov(X,X)}
\end{aligned}
$$

By rewriting the equation this way, we can see a cancelling term which can be 7 orders of magnitude larger than the other terms, due to the relatively large dz^2. This term lead to complete absorption of the other terms in single-precision floating point. When this large term is taken out of the subtraction, the subtracted terms remains of the same magnitude and the operation can be performed safely in single-precision.

Similarly, the second issue is in the update of $cov(X,T_x)$ and $cov(X,X)$. In both equations, the problematic term is $(1 - K_x)$, which can be rewritten as:

$$
\begin{aligned}
(1 - K_x) &= 1 - \frac{pred\_cov(X, X)}{hit_{weight} + pred\_cov(X, X)} \\
&= \frac{hit_{weight} + pred\_cov(X, X) - pred\_cov(X, X)}{hit_{weight} + pred\_cov(X, X)}
\end{aligned}
$$

As before, the canceling terms have a different magnitude than the hit weight, leading to an absorption. Furthermore, rewriting the equation to solve this issue, lead to simpler equations:

$$
\begin{aligned}
cov(X, T_x) &= (1 - K_x) \cdot pred\_cov(X, T_x) = hit_{weight} \cdot K_{T_x} \\
cov(X, X) &= (1 - K_x) \cdot pred\_cov(X, X) = hit_{weight} \cdot K_x
\end{aligned}
$$

While in this algorithm, the equation were simple enough to be worked by hand, it is not always the case and larger algorithms in the LHCb software could benefit from using automated round-off error propagation estimation libraries such as CADNA [91, 55].

5.5 Benchmarks

Following the approach of [49], all the algorithm configurations have been tested within the `GAUDI` framework [119]. To isolate the algorithm time, the framework provides an efficient way of reading data from a local ramdisk and dispatching the events to the different threads. To decouple the single-threaded file I/O from the multi-threaded reconstruction, two buffers are used to hold event data. When a thread reaches the end of the current event buffer, input is swapped to the second buffer. The thread then refills the first buffer from the next file in parallel with the event processing. The sequential work of prefetching events consists in computing the pointer to the start of every event by doing the prefix sum of the event sizes. The decoding of the individual event raw banks is then performed by the thread in charge of the event reconstruction. The throughput is measured in number of events per second (Hertz). To measure the algorithm duration, the timing counters provided by the framework are used. It ignores the first and last 10% of events for stability. All tested software was compiled with GCC 8.2. Unless explicitly specified, all results are given for a number of seeding pair candidate N=3.

Two systems were evaluated: a dual-socket Intel Xeon Gold 6130 and a single socket AMD EPYC "Rome" 7702. The Intel system features AVX-512, AVX2 and SSE instruction sets and scales its frequency according to Table 2.2. The AMD system only has AVX2 and SSE, at a frequency of 2.0 GHz.[1] The dual-socket Intel system has a total number of 32 physical cores and 64 logical cores, while the single socket AMD system has 64 physical cores and 128 logical cores. As every event is independent from the others, memory latency induced by NUMA effects can be avoided by binding the processes to a single NUMA and restricting them to the

[1] All AMD throughput numbers in this chapter are given for 2.0 GHz.

local memory of the NUMA domain. On the Intel system optimal performance was achieved by launching one process per NUMA domain using the `numactl` utility. While the one socket AMD only has one NUMA domain for the whole chip, it was found that best performance is achieved when launching one process per physical compute die, and thus always ran 8 independent jobs on this system.

In all tests, a full VELO reconstruction is ran, consisting in fetching the raw banks, applying a Global Event Cut of the 7% biggest events, preparing the data, performing the actual tracking, and fitting the resulting tracks. For the sake of a fair comparison, the 2018 Search by Pair (SbP) algorithm has been updated to use similarly simplified data structures as the SIMD algorithm, resulting in a speed improvement of ∼40%.

5.5.1 Throughput

All throughput tests were done on minimum bias Monte-Carlo simulation samples. First, the throughput of the new SIMD VELO Tracking algorithm is compared with the improved SbP algorithm, the current state-of-the-art for VELO pattern reconstruction on CPU. The SbP algorithm was originaly coming in two versions: the "fast" configuration favoring speed over efficiency was meant to be used for HLT1 and the "best" configuration favoring efficiency over speed was meant to be used for HLT2. The left side of Figures 5.6 and 5.7 shows the SIMD algorithm is *faster* than both the "fast" and "best" configurations of SbP, on *every* tested architectures, for any number of threads. Using SIMDWrappers, different implementations of the SIMD algorithm were also compared. Interestingly, the AVX512 backend with an SIMD register width of 16 performs less well than the AVX256 with a register width of 8. This can be explained by the frequency scaling issues discussed in Chapter 2. However, thanks to the new instructions introduced with AVX-512, the AVX256 and AVX128 backends bring a 10% improvement over plain AVX2 and SSE respectively. The scalar backend is significantly lower than all other SIMD backends because the compiler is not able to vectorize the filtering pattern. The slow down in speedup progression with increased SIMD parallelism is a combination of frequency scaling and not being able to extract enough parallelism from relatively small loops. The right of Figure 5.7 presents a comparison of the Intel and AMD systems for the relevant backends. Because the Intel setup only has 32 physical cores, the two architectures can only be compared at threads × processes = 32. For this number of threads, the AMD's AVX2 backend provides a 28% improvement over Intel's AVX2 and a 18% improvement over Intel's AVX256, despite the frequency being 20% lower for AMD. AMD's scalar backend also increased the throughput by 23% from Intel's scalar backend.

As the throughput and efficiency of the algorithm both depends on the number of pair candidates considered during the seeding step, it offers a direct tuning parameter to adjust the trade-off between speed and efficiency. Figure 5.8 shows the progression of throughput as a function of the number of pair candidates.

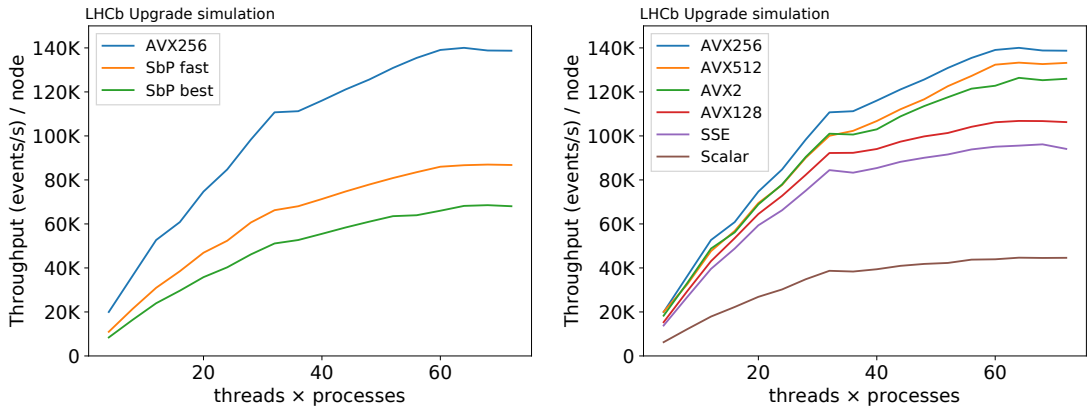


Figure 5.6: Throughput as a function of the number of threads for two processes (one on each NUMA domain) on dual-socket Intel Xeon Gold 6130. On the left: comparison of SIMD VELO Tracking with the SbP algorithm in "fast" and "best" configurations. On the right: comparison of different SIMD backends. [75]

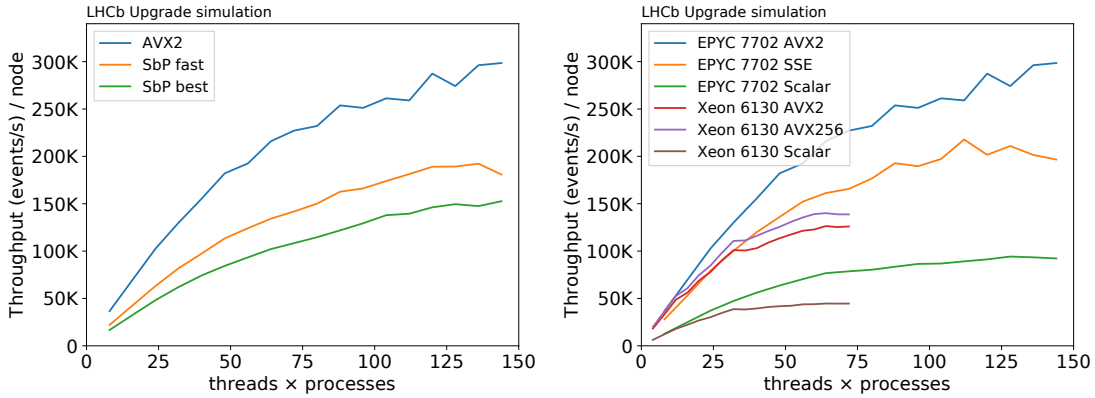


Figure 5.7: On the left: comparison of SIMD VELO Tracking with the SbP algorithm in "fast" and "best" configurations, on a single socket AMD EPYC "Rome" 7702. On the right: comparison of a single socket AMD EPYC "Rome" 7702 and a dual-socket Intel Xeon Gold 6130 for different SIMD backends. [75]

5.5.2 Reconstruction physics efficiency

While minimum bias events are representative of the data seen by the production system in real-time, the majority of them do not contain the physical signals whose efficiency we want to optimize. Therefore, a typical signal of interest for LHCb is simulated to measure the algorithm efficiencies. The decay $B_s^0 \to \phi\phi$ is chosen, as it is the same signal used in all LHCb historical tracking publications.

The efficiency of different track categories are presented as a function of multiple physical parameters of interest. The efficiency as a function of the distance of closest approach to beamline (docaz) is studied to ensure the efficient reconstruction of tracks produced in the decays of long-lived particles. Because of the geometry of the VELO, it is also particularly interesting to plot the efficiency as a function of

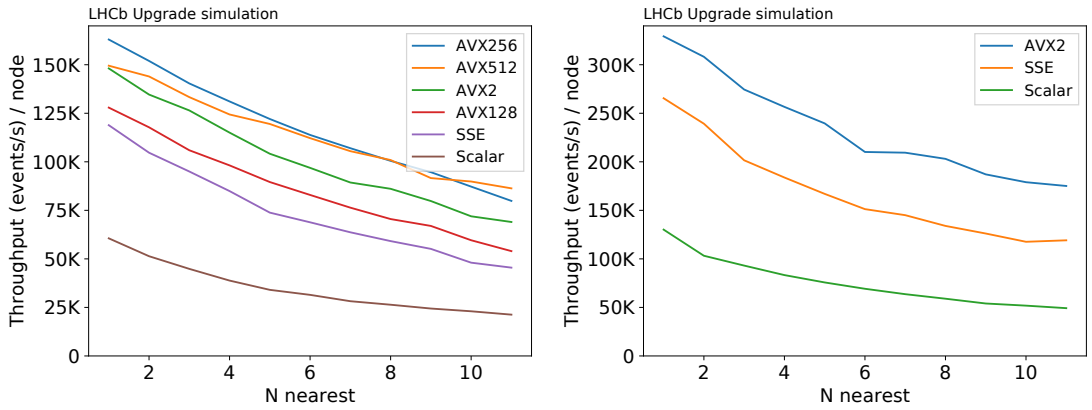


Figure 5.8: Throughput as a function of the number of pair candidates in the track seeding step, for different SIMD backends. On the left: dual-socket Intel Xeon Gold 6130. On the right: single socket AMD EPYC "Rome" 7702. [75]

track pseudorapidity (η).[2] Figure 5.9 shows the efficiency as a function of docaz and η for the two configurations of the SbP algorithm and the AVX256 version of the SIMD VELO Tracking. The docaz efficiencies are plotted in the range $2 < \eta < 5$, which represents the acceptance of the full LHCb detector. Thanks to its nearest ϕ approach, the SIMD VELO Tracking is more efficient than previous state-of-the-art for very displaced tracks, even if the number of evaluated pair candidates is small (N=3). It also has significantly better efficiencies for very small track η. While small η tracks do not pass through the rest of the detector, they can nevertheless play an important role in the reconstruction of PVs.

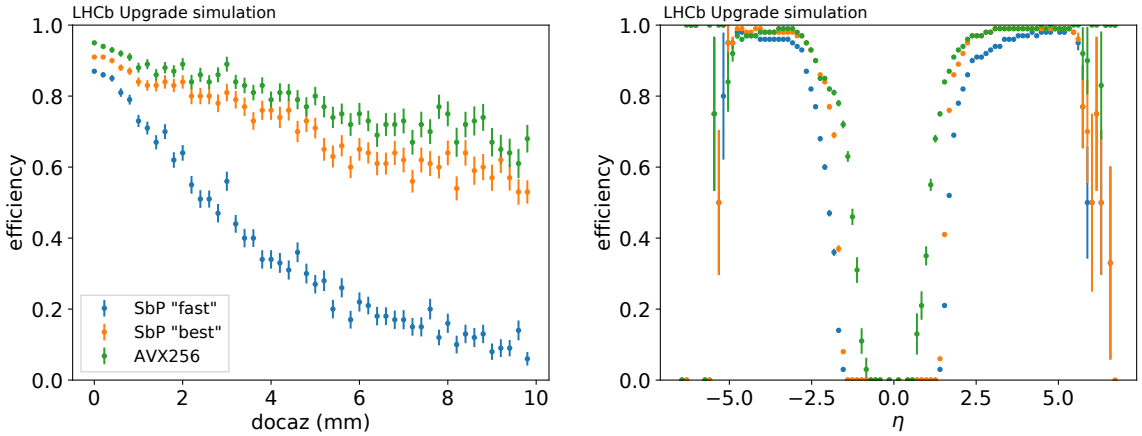


Figure 5.9: Efficiency as a function of the distance of closest approach to the z axis (docaz), in mm, and the pseudorapidity η for Velo tracks. LHCb is mostly interested in tracks with DOCAZ < 1 mm.

The integrated efficiency was also evaluated in the range $2 < \eta < 5$ for different track types:

- all tracks that leaves at least 3 hits in the VELO detector ("VELO tracks")

[2] The pseudorapidity of a track is given by $-\ln\left[\tan\left(\frac{\theta}{2}\right)\right]$, where θ is the angle of the track to the beamline

which are mostly produced directly in the PVs,

- tracks that come from the decay of a hadron containing a bottom quark and traverse the rest of the LHCb tracking detectors ("From B"),
- tracks that come from the decay of a hadron containing a charm quark and traverse the rest of the LHCb tracking detectors ("From D"),
- tracks that come from the decay of a hadron containing a strange quark and traverse the rest of the LHCb tracking detectors ("Strange").

Table 5.1 compares the efficiencies for these categories and compares the fake rate for the two configurations of SbP and AVX256 SIMD VELO Tracking. More efficient algorithms tends to produce more fakes due to the higher number of combinations tested. Still, the SIMD algorithm produces fewer fakes than the "best" SbP. The SIMD algorithm is more efficient for "VELO" and "From B" categories, while being within 1% of the "best" SbP for "From D" and "Strange" categories. While the results presented here only allow for $N = 3$ pair candidate, the algorithm can outperform the other algorithms in all categories for $N \geq 6$. The clone rates are similarly presented in Table 5.2, and the SIMD algorithm produces fewer clones than previous approaches. The efficiencies are computed exactly using ground truth from the Monte-Carlo simulation and the uncertainties were computed using the *normal approximation* method described in [131].

	Velo	From B	From D	Strange
SbP "fast"	93.05 ± 0.08	95.64 ± 0.32	95.29 ± 0.52	79.34 ± 0.74
SbP "best"	97.62 ± 0.05	98.71 ± 0.18	$\mathbf{99.05 \pm 0.24}$	$\mathbf{97.46 \pm 0.29}$
VELO SIMD	$\mathbf{98.20 \pm 0.04}$	$\mathbf{99.12 \pm 0.15}$	98.99 ± 0.24	96.82 ± 0.32

Table 5.1: Efficiencies for tracks that are not electrons in the range $2 < \eta < 5$.

	Velo clones	From B clones	From D clones	Strange clones
SbP "fast"	2.31 ± 0.05	0.89 ± 0.15	1.42 ± 0.29	1.54 ± 0.23
SbP "best"	2.75 ± 0.05	0.84 ± 0.14	1.25 ± 0.27	$\mathbf{0.82 \pm 0.17}$
VELO SIMD	$\mathbf{1.35 \pm 0.04}$	$\mathbf{0.68 \pm 0.13}$	$\mathbf{0.90 \pm 0.23}$	$\mathbf{0.82 \pm 0.17}$

	Fakes
SbP "fast"	$\mathbf{0.83 \pm 0.02}$
SbP "best"	1.22 ± 0.02
VELO SIMD	1.04 ± 0.02

Table 5.2: Clone rates (in %) on 1000 $B_s^0 \to \phi\phi$ events, for $2 < \eta < 5$ tracks. The lower, the better.

Figure 5.10 shows the impact of varying the number of seeding pair candidates on the efficiency and fake rate. As N increases, the efficiencies go up, but as more pairs are tested, the probability of finding randomly aligned hits also increases leading to more fakes. It can be noted that already at N=1, the efficiencies are higher than

the "fast" SbP algorithm and could offer a viable mitigation for HLT1. However, to minimize the disparity between HLT1 and HLT2 track reconstructions, a default value of N=3 was chosen for both configurations as it seems to be a good trade-off between throughput, efficiency and fake rate.

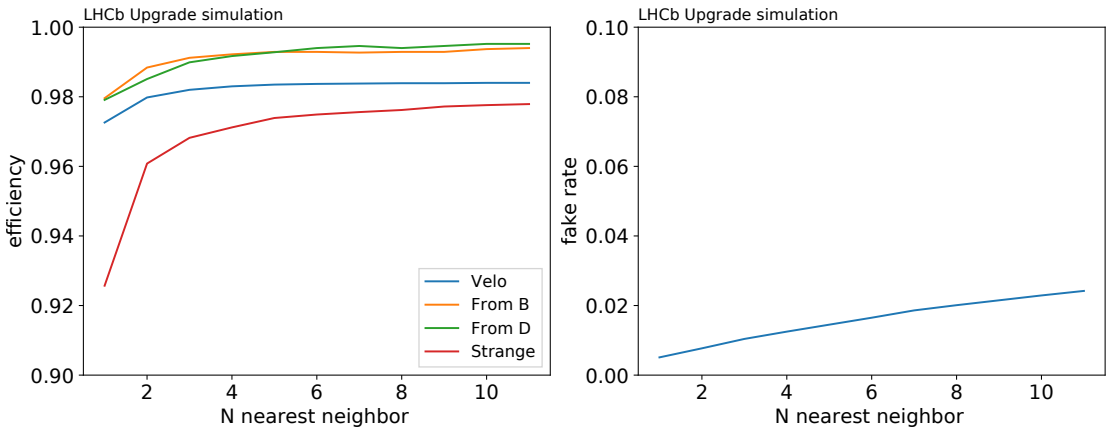


Figure 5.10: Efficiencies for the different track categories described in the text as a function of the number of seeding candidates. Higher efficiencies and lower fake rate are better.

Table 5.3 shows the hit efficiencies for the same track categories. The "All hits efficiencies" section shows that in average, the SIMD algorithm finds more correct hits than the SbP algorithms. The "First 3 hits" and "last hits" categories are important to correctly extrapolate the track to the beamline or to the end of the VELO detector. Again, the SIMD algorithm demonstrates better, or within 1%, efficiencies than the SbP algorithms.

	Velo	From B	From D	Strange
All hits efficiencies:				
SbP "fast"	90.05 ± 0.03	92.40 ± 0.15	92.82 ± 0.22	84.69 ± 0.23
SbP "best"	94.19 ± 0.03	97.61 ± 0.09	97.50 ± 0.13	97.47 ± 0.10
VELO SIMD	$\mathbf{96.97 \pm 0.02}$	$\mathbf{97.96 \pm 0.08}$	$\mathbf{98.05 \pm 0.12}$	$\mathbf{97.58 \pm 0.10}$
Efficiencies of first 3 hits:				
SbP "fast"	91.74 ± 0.05	93.93 ± 0.22	94.32 ± 0.33	76.54 ± 0.45
SbP "best"	93.86 ± 0.04	97.44 ± 0.14	97.54 ± 0.22	96.80 ± 0.19
VELO SIMD	$\mathbf{97.08 \pm 0.03}$	$\mathbf{98.18 \pm 0.12}$	$\mathbf{98.13 \pm 0.19}$	$\mathbf{97.39 \pm 0.17}$
Efficiencies of last hits:				
SbP "fast"	87.95 ± 0.04	90.77 ± 0.21	91.43 ± 0.31	88.97 ± 0.26
SbP "best"	94.07 ± 0.03	$\mathbf{97.64 \pm 0.11}$	97.48 ± 0.17	$\mathbf{97.72 \pm 0.12}$
VELO SIMD	$\mathbf{96.60 \pm 0.02}$	97.53 ± 0.11	$\mathbf{97.85 \pm 0.16}$	97.12 ± 0.14

Table 5.3: Average hit efficiencies of reconstructed tracks in the range $2 < \eta < 5$. All results are given for an SIMD register width of 256-bits. (Best efficiencies in bold)

5.6 Conclusion

In this chapter, a new tracking algorithm for the VELO detector of the LHCb experiment specialized to take advantage of SIMD general purpose multicore processors has been presented. It was compared to previous state-of-the-art algorithms and showed a significant speedup and in some cases increase in efficiency over all previous alternatives. This allows the SIMD algorithm to be used for all stages of LHCb's real-time data processing. It was also evaluated on two high-end systems from Intel and AMD and the impact of the SIMD extensions on the performance was shown. The AVX256 version of the presented algorithm improves by a factor 2 the throughput of the Search by Pair algorithm in "best physics" configuration, while maintaining a similar level of physics performance. This algorithm has been published in a journal: [75].

Chapter 6

Scalability of the LHCb software

Contents

6.1 Introduction

This chapter aims to show the impact of the software optimisations and choice of architecture on LHCb's High Level Trigger. Both HLT1 and HLT2 are evaluated on different CPU architectures from Intel, AMD and ARM. In 2020, the LHCb collaboration decided to use GPUs in the event builder farm to run HLT1 instead of moving the events to an HLT1 CPU filter farm through a high speed network. The studies presented here focus on the legacy scenario of an HLT1 running on CPU. For HLT2, plans were not changed and the software will run on a CPU filter farm.

6.2 Evaluation of HLT1 on CPUs

As presented in Chapter 1, the goal of LHCb's HLT1 is to reduce the event rate to a level that can be processed by the full analysis of HLT2, while remaining efficient on the full range of signals of interest for LHCb. In practice, HLT1 is looking for tracks or two-track vertices displaced from the primary vertices and for leptons, particularly muons, regardless of their displacement.

From Autumn 2018 to Summer 2019 the evaluated reconstruction sequence for HLT1 closely followed the "displaced track reconstruction" fallback scenario described in the Trigger Technical Design Report [116], which didn't included the lepton reconstruction. The evaluated HLT1 sequence consisted of the following steps:

- Decoding and clustering of the VELO raw data;

- Decoding of the UT raw data, already clustered;
- Decoding of the SciFi raw data, already clustered;
- Sorting of the VELO, UT and SciFi clusters to speed up the subsequent reconstruction algorithms;
- Finding VELO tracks;
- Building primary vertices from VELO tracks;
- Selecting VELO tracks with an impact parameter greater than 100 μm (IP cut);
- Searching for UT hits matching the selected VELO tracks, within a 800 MeV transverse momentum window, building VELO-UT (upstream) tracks;
- Searching for SciFi hits matching VELO-UT tracks, within a 1000 MeV transverse momentum window, building forward tracks;
- Fitting the resulting forward tracks using a parameterized Kalman filter;

Figure 6.1 describes the HLT1 throughput evolution during this period. The throughput was monitored continuously using nightly tests running on a reference node consisting of a dual socket Intel Xeon E5-2630V4 CPU. At that time, the trigger architecture considered was an HLT farm of at least 1000 of such reference node. The goal was therefore to reach a throughput of at least 30 KHz per node in order for the farm to sustain the event rate of 30 MHz of Run 3. This goal was reached in May 2019 thanks to three major improvements:

- Applying a tighter track tolerance criteria on the SciFi reconstruction algorithm;
- Reworking the algorithm logic of the SciFi reconstruction [70];
- Replacing the VELO reconstruction algorithm by the new SIMD algorithm presented in this thesis and adapting the event model to be SIMD friendly;

Along the way, throughput improvements allowed to loosen some track selections. For instance, the IP cut was completely removed allowing for better efficiencies in upstream and forward tracks.

In November 2019, the evaluation of HLT1 was extended beyond the reference node to more recent hardware, especially to the new AMD Zen 2 architecture that was just released. Figure 6.2 shows HLT1 throughput on an AMD EPYC "Rome" 7702 as a function of the number of threads used. To evaluate the impact of AMD chiplet architecture on HLT1, the throughput was measured for 1, 2, 4 and 8 independent processes assigned to different NUMA domains (corresponding to the 8 different chiplets). Due to the embarrassingly parallel nature of HLT1, it can be split in any number of independent processes each executing on different events. Maximum throughput was achieved when launching one process per NUMA domain.

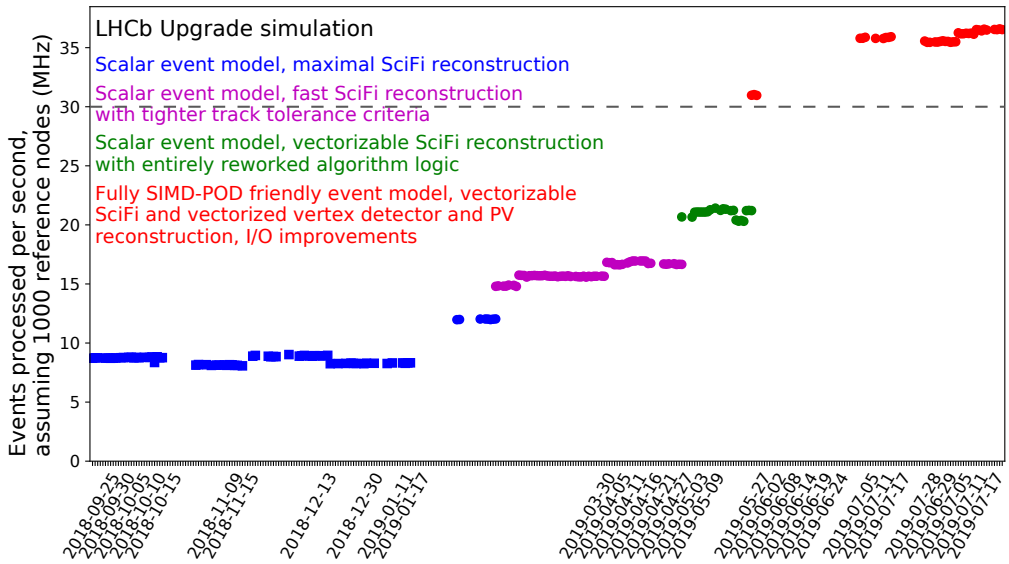


Figure 6.1: Evolution of the upgrade LHCb HLT1 throughput between autumn 2018 and summer 2019. The dates on the x-axis are not fully chronological because various independent nightly tests with overlapping time intervals were integrated in one timeline. The target goal of 30 MHz is depicted by a dashed line. [3]

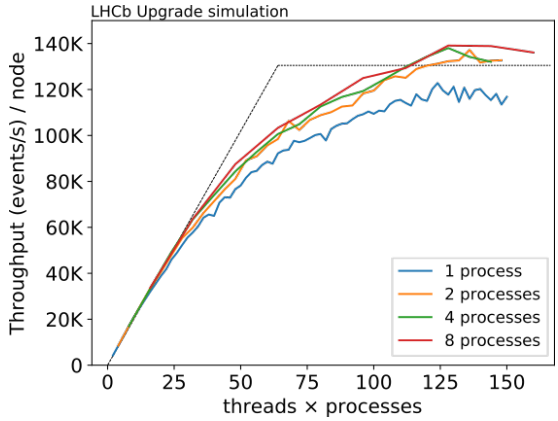


Figure 6.2: HLT1 throughput on an AMD EPYC "Rome" 7702 as a function of the number of threads used. Maximum throughput is achieved when launching one process per NUMA domain.

Table 6.2 shows the throughput of HLT1 on different x86 CPUs from Intel and AMD. When possible, the measurements were made on two sockets systems, noted 2/2 under the #CPU column, but some of the machines used didn't have the second socket populated (they are noted 1/2). For these machines, the performance of one socket was measured and extrapolated, the throughput in the last column are given for one and two socket, when applicable.

In addition to x86, which remains the main CPU architecture targeted, additional CPUs using ARM architectures were evaluated. Previous work have paved the way to run the LHCb software stack on ARM by progressively adapting the framework [93, 141]. However, due to the lack of continuous integration for non-x86 architectures, some ARM specific bugs and incompatibilities were later introduced

Name	Cores	Freq (GHz)	L3 (MB)	TDP (W)	#CPU	HLT1 evts/s
EPYC 7302	16c/32t	3.0/3.3	128	155	2/2	47.5k/95k
EPYC 7452	32c/64t	2.35/3.35	128	155	1/2	79k/158k
EPYC 7702	64c/128t	2.0/3.35	256	200	1/2	140k/280k
EPYC 7742	64c/128t	2.25/3.4	256	225	1/2	150k/300k
Ryzen 9 3900X	12c/24t	3.8/4.6	64	105	1/1	46k/–
Xeon Gold 6130	16c/32t	2.1/3.7	22	125	2/2	29k/58k
Xeon Platinum 9242	48c/96t	2.3/3.8	77	350	2/2	100k/200k
Xeon E5-2630V4 (Reference node)	10c/20t	2.2/3.1	25	85	2/2	18.5k/37k

Table 6.1: HLT1 Throughput on x86 in November 2019. Throughputs are given for 1 and 2 CPU per node (if applicable).

and had to be resolved prior to testing. Additionally, a Neon backend was added to the SIMDWrappers library. ARM Neon is a 128-bit SIMD instruction set similar to x86 SSE.

ThunderX2 is a family of 64-bit multi-core ARM server microprocessors introduced by Cavium in early 2018 succeeding the original ThunderX line. The model that was tested is the flagship CN9980 featuring 32 physical cores with a 4 way Simultaneous Multi-Threading (SMT), bringing the number of logical cores to 128. Figure 6.3 shows a comparison of HLT1 on a single Cavium ThunderX2 and a single Intel Xeon 6130. This Xeon was chosen due to its similar process, frequency, launch date and price allowing to focus the comparison on the architectural choices. When the two machines parallelism are fully utilized, the performances are roughly the same, with a slight 2.7% advantage for the Xeon, but the ThunderX2 has 47% more raw scalar performances, without SIMD optimizations. The throughput scales linearly with the number of physical cores for both architectures. The contribution of SMT to the throughput depends on the instructions, it is more beneficial for the scalar backends as they use more instruction level parallelism and use more diverse instructions, while the SIMD implementations tends to be more regular and make heavy uses of the same units. The 4 way SMT of the ThunderX2 allows it to close the gap with the Xeon, even though it is using a 128-bit wide SIMD versus a 256-bit wide SIMD.

Announced in 2017, SVE is a vector length agnostic SIMD instruction set aimed at HPC and machine learning applications. At the time of writing, the only hardware implementation of SVE is the Fujitsu A64FX CPU, with a maximum vector width of 512 bits. The A64FX has 48 physical cores with no SMT and like all ARM processors, also features Neon for backward compatibility. To feed the SIMD compute units, the A64FX doesn't have a L3 cache but is instead directly connected to an off-chip HBM2 memory offering a peak of 1 TB/s reading bandwidth. As the HBM2 acts as the Random Access Memory (RAM) of the system, it is limited to the 32 GB available, which is not a problem for the HLT1 application that fit within

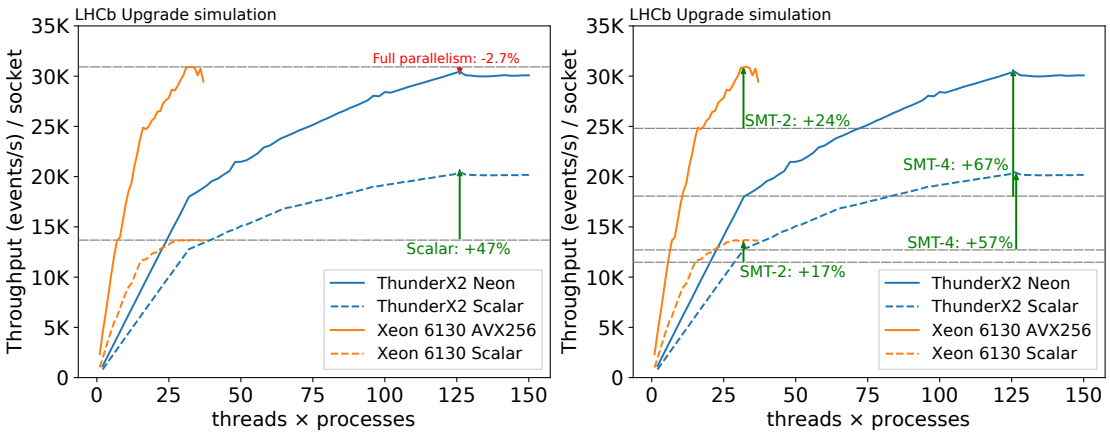


Figure 6.3: Comparison of HLT1 on a single Cavium Thunder X2 (32c/128t) and a single Intel Xeon 6130 (16c/32t). The plot on the left is showing the impact of SIMD on the throughput. The plot on the right is showing the impact of Simultaneous Multi-Threading (SMT) on each configuration.

this budget.

By default, SVE intrinsic types provided by the compiler are sizeless, meaning they cannot be wrapped in an object, a feature of C++ SIMDWrappers relies on. However, since the hardware is known to have a vector width of 512-bits, the GNU C Compiler (GCC) can be instructed to use only this width, with the `-msve-vector-bits=512` option. In addition of allowing the SVE types to be wrapped, it also provides the compiler with more hints about the hardware enabling it to further optimize for it. For the benchmarks, the version 10.3.0b of GCC for ARM 64-bits was used.

Figure 6.4 shows a comparison of the Fujitsu A64FX and the Cavium ThunderX2 processors on the Velo only sequence and HLT1. The Velo only sequence is the SIMD Velo reconstruction algorithm presented in Chapter 5 and account for a bit less than half of the HLT1 sequence. Due to its heavy use of SIMD, it benefits a lot from the 512-bits wide SVE compared to using only Neon. It can be noted that the Neon implementation of the A64FX is provided for backward compatibility and is not recommended for performance applications. The rest of HLT1 is also benefiting from SVE, but not as much as the first 50%. This is explained by the fact that other algorithms of HLT1 still use some scalar part and use SIMD opportunistically rather than in their core design. As the LHCb framework was optimized for an embarrassingly parallel problem, where event can be processed fully independently, it cannot take advantage of the very high bandwidth offered by the A64FX and could benefit from more physical cores or SMT.

6.3 Evaluation of HLT2 on CPUs

Unlike HLT1, the HLT2 sequence is not yet fully defined and multiple sequences are under evaluation. Depending of the optimizations that can still be achieved before the start of data taking, the scenario with the best compromise of speed and physic

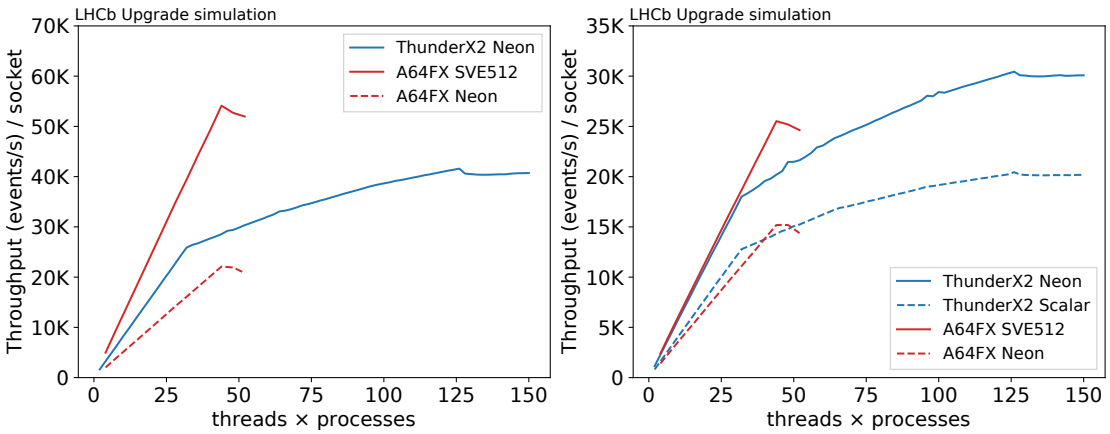


Figure 6.4: On the left, throughput of the Velo only sequence on the Fujitsu A64FX, using Neon and SVE, and Cavium ThunderX2 using Neon. On the right, throughput of HLT1 on the Fujitsu A64FX, using Neon, and Cavium ThunderX2 using Neon and Scalar backends.

efficiency will be selected. Figure 6.5 shows the two extremes of the evaluated sequences. The baseline sequence was directly adapted from Run2 offline analysis and offer the best physics quality. The fastest sequence was fine-tuned to improve the throughput while maintaining a reasonable level of efficiency.

The HLT2 sequence starts by running HLT1 again. Because the data is buffered between the two trigger stages, the alignment and calibration can improve the result of HLT1, rerunning it ensure the best version of the calibration is used. Next, alternative reconstruction algorithms are run to get the downstream tracks and additional long tracks, by seeding tracks in the SciFi and matching them with VELO tracks. This procedures helps to get new particles that would otherwise be ignored by the HLT1 sequence, but also creates a lot of duplicate tracks. A bidirectional Kalman filter is then used to fit all the trajectories, using the hits inside every detector available. As the tracks are fitted, the quality of the tracks are assessed using a χ^2 test and only the best tracks are kept. Duplicates are also removed during this step. The fitted tracks are then used in particle identification algorithms using the RICHs and calorimeters data. Lastly, a large number of selection lines are evaluated to decide if the event should be kept. Each selection line design is driven by a specific physic analysis.

As HLT2 must be as close as possible to offline analysis quality, compromises are much harder to make than in HLT1 where the looser selections give some room for inefficiency. In addition, the algorithms of HLT2 need to handle complexes edge cases that make them difficult to vectorize efficiently. For this reasons, HLT2 is still mostly scalar. One exception, the RICH raycasting and particle identification was the earliest introduction of SIMD in LHCb's framework, using the VC library. However, since VC does not support AVX-512 or ARM SIMD extensions, the RICH code fallbacks on a scalar backend for these architectures. As a results, SIMD does not have any impact on HLT2 which scales only with the number of cores and the hardware optimizations of scalar paths. Figure 6.6 shows a throughput comparison of HLT2 baseline on a Cavium ThunderX2, an Intel Xeon 6130 and a Fujitsu

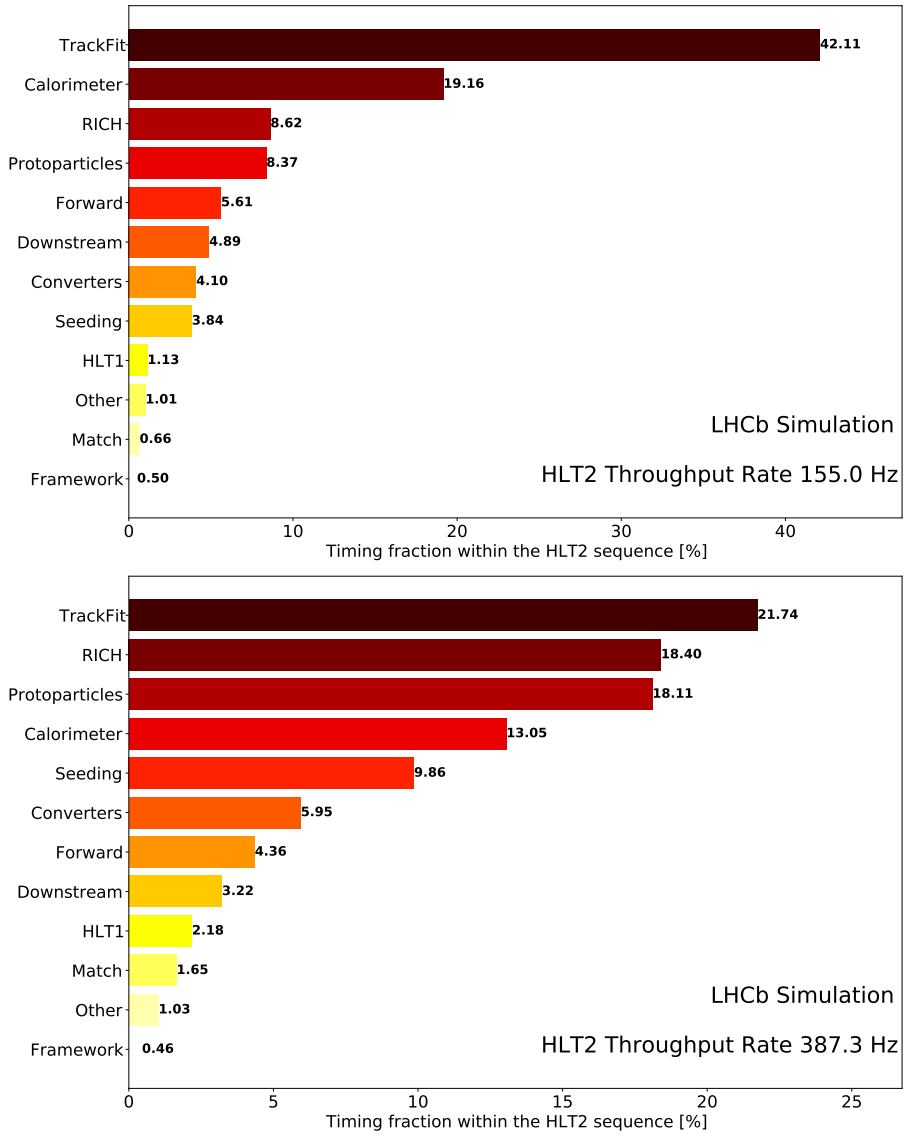


Figure 6.5: Timing fraction and throughput of the baseline (top) and fastest (bottom) HLT2 sequences on the reference node Xeon E5-2630V4 from the LHCbPR continuous evaluation, August 26th 2021.

A64FX processors. Unlike HLT1, the SVE version of HLT2 offer no benefit over the Neon version on the A64FX. But the simultaneous multi-threading and higher core count of the ThunderX2 makes it a competitive option over the Intel Xeon 6130.

At the time of writing, optimization efforts are focused on the Kalman track fitter algorithm, which takes the largest fraction of the HLT2 sequence. Theoretical studies have shown a promising speedup when using SIMD to optimize the Kalman filter computations at the core of the fitter [37, 108, 114, 109, 110]. However, integrating these findings into an algorithm usable in HLT2 has proven difficult as a large amount of time is spent preparing the measurements that has to be fitted, in particular the tracks have to be extrapolated through the magnetic field represented as a voxel grid, making the vectorization of the fitter over tracks more challenging.

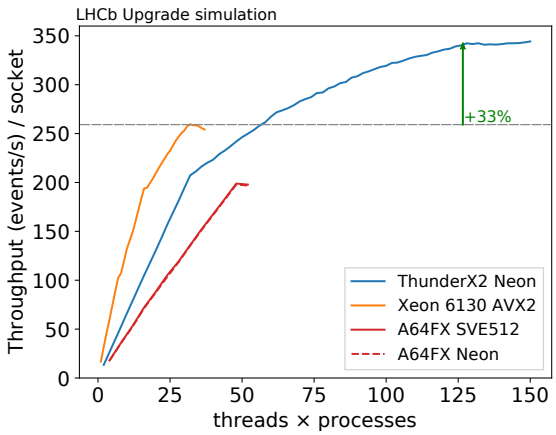


Figure 6.6: Throughput comparison of HLT2 baseline on a Cavium ThunderX2, an Intel Xeon 6130 and a Fujitsu A64FX processors. Only a negligible part of HLT2 takes advantage of SIMD extensions, leading to no differences between SVE512 and Neon backends.

6.4 Conclusion

This chapter has shown the performances of the HLT1 and HLT2 software running on modern CPUs from different vendors. Since 2019, the goal of running HLT1 at a throughput of 30 MHz on a reference farm of 1000 Intel Xeon E5-2630V4 CPU has been reached. New architectures allows to reach up to 300 KHz per node, allowing to reduce the size of the CPU filtering farm by a factor 10. HLT2 development is following the path of HLT1 by reusing the SoA event model. At the time of writing, the largest algorithm of HLT2 algorithm remains the Kalman track fitter.

Conclusion

LHCb is realizing an ambitious upgrade of its detector and trigger system. In order to collect more data, the luminosity within the detector is increased, leading to more collisions per bunch crossing. To cope with this change, detector parts have to be replaced. Notably, the VELO sub-detector is upgraded to use a pixel array instead of crossing strips, allowing it to better distinguish between different particle trajectories. As the detector is observing more collisions and the data volume is becoming too great to be stored, the trigger have to handle up to 3 TB/s of data in real-time and make intelligent decisions to keep the most interesting events. Moving from an hardware trigger to a full software trigger operating at 30 MHz requires to rethink the algorithms used to better fit general purpose processors. Through this thesis, modern CPUs and GPUs strengths and weaknesses have been analysed with a selection of irregulars, non-trivial to parallelize, algorithms. The findings were then applied to develop new algorithms for the LHCb trigger and guide the general architecture of the LHCb software framework.

The connected component labeling and analysis problem, a classic algorithm in computer vision, was studied on GPUs and SIMD CPUs leading to the discovery of several new algorithms improving upon the state-of-the-art on both CPUs and GPUs for dense images, finding applications beyond the context of this thesis. In addition, a new parameterizable connected component labeling and analysis algorithm for sparse images of densities lower than 0.5%, and a specialization of this algorithm for the LHCb VELO data preparation was introduced in Chapter 4.

In Chapter 5, a new tracking algorithm for the VELO detector of the LHCb experiment specialized to take advantage of SIMD general purpose multi-core processors was presented. It was compared to previous state-of-the-art algorithms and showed a significant speedup and in some cases increase in efficiency over all previous alternatives. The algorithm was also evaluated on two high-end systems from Intel and AMD and the impact of the SIMD extensions on the performance was shown. The AVX256 version of the algorithm improved by a factor 2 the throughput of the Search by Pair algorithm in "best physics" configuration, while maintaining a similar level of physics performance. This allows the same SIMD algorithm to be used for all stages of LHCb's real-time data processing. Furthermore, the nearest ϕ neighbors approach used by the new algorithm was found to scales better with the occupancy of the detector than the ϕ window approach used in previous versions, making it a suitable candidate for future upgrades.

The SIMDWrappers library developed for this thesis and presented in Chapter 2 is now a part of the LHCb framework and used in many algorithms of the trig-

ger system. It supports x86's SSE, AVX2 and AVX-512 instruction sets as well as ARM's Neon and SVE. The multiple backends of AVX-512 with different widths, makes this library a valuable tool to study the impact of vector length and frequency scaling on an algorithm.

This thesis contributed to the effort of LHCb's Real Time Analysis (RTA) group to reach the LHCb's ambitious goal that is a software trigger running at 30 MHz, as shown in Chapter 6. Insights gained from the optimisation of HLT1 were also used to improve HLT2 and to guide early research dedicated to future upgrades. As architectures continue to evolve, and are becoming more and more parallel, software has to be adapted to stay efficient. Tools and libraries must be developed to help physicists build fast algorithms that meet the experience's physics efficiency requirements.

Publications

In addition, this research work yielded the following publications:

- Journals:
  - "A fast and efficient SIMD track reconstruction algorithm for the LHCb upgrade 1 VELO-PIX detector", JINST 2020 [75]
- International conferences:
  - "Taming Voting Algorithms on Gpus for an Efficient Connected Component Analysis Algorithm", ICASSP 2021 [112]
  - "Evolution of the energy efficiency of LHCb's real-time processing", CHEP 2021 [6]
  - "How to speed Connected Component Labeling up with SIMD RLE algorithms", WPMVP 2020 [111]
  - "Designing efficient SIMD algorithms for direct Connected Component Labeling", WPMVP 2019 [80]
  - "SparseCCL: Connected Components Labeling and Analysis for sparse images", DASIP 2019 [77]
  - "A new Direct Connected Component Labeling and Analysis Algorithms for GPUs", DASIP 2018 [76]
  - "Energy and Execution Time Comparison of Optical Flow Algorithms on SIMD and GPU Architectures", DASIP 2018 [135]
  - "METEORIX: a cubesat mission dedicated to the detection of meteors", COSPAR 2018 [144]
- National conferences and communications:
  - "Taming Voting Algorithms on GPUs for an Efficient Connected Component Analysis Algorithm", GTC 2021 [113]
  - "A new Direct Connected Component Labeling and Analysis Algorithm for GPUs", GTC 2019 [78]
  - "Étiquetage et analyse en composantes connexes sur GPUs", COMPAS 2019 [79]
  - "Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU", COMPAS 2018 [133]

– “Comparaison de la consommation énergétique et du temps d’exécution d’un algorithme de traitement d’images optimisé sur des architectures SIMD et GPU”, GdR SOC2 2018 [134]

List of Figures

List of Tables

Bibliography

[1] "Intel Xeon Silver 4114 frequencies," https://en.wikichip.org/wiki/intel/xeon_silver/4114#Frequencies, accessed: 2019-08-20.

[2] "Simdprune library," https://github.com/lemire/simdprune, accessed: 2019-08-20.

[3] "Evolution of the upgrade LHCb HLT1 throughput," Jul 2019. [Online]. Available: https://cds.cern.ch/record/2684267

[4] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[5] R. Aaij, S. Benson, M. D. Cian, A. Dziurda, C. Fitzpatrick, E. Govorkova, O. Lupton, R. Matev, S. Neubert, A. Pearce, and et al., "A comprehensive real-time analysis model at the LHCb experiment," *Journal of Instrumentation*, vol. 14, no. 04, p. P04006–P04006, Apr 2019. [Online]. Available: http://dx.doi.org/10.1088/1748-0221/14/04/P04006

[6] R. Aaij, D. H. Cámpora Pérez, T. Colombo, C. Fitzpatrick, V. V. Gligorov, A. Hennequin, N. Neufeld, N. Nolte, R. Schwemmer, and D. Vom Bruch, "Evolution of the energy efficiency of LHCb's real-time processing," in *25th International Conference on Computing in High-Energy and Nuclear Physics*, vol. 251, Online, France, May 2021, p. 04009. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03270156

[7] R. Aaij *et al.*, "Allen: A high level trigger on GPUs for LHCb," 2019.

[8] A. Abba, F. Bedeschi, M. Citterio, F. Caponio, A. Cusimano, A. Geraci, F. Lionetto, P. Marino, M. J. Morello, N. Neri, D. Ninci, A. Piucci, M. Petruzzo, G. Punzi, F. Spinella, S. Stracka, D. Tonelli, and J. Walsh, "A specialized track processor for the LHCb upgrade," CERN, Geneva, Tech. Rep. LHCb-PUB-2014-026. CERN-LHCb-PUB-2014-026, Mar 2014. [Online]. Available: https://cds.cern.ch/record/1667587

[9] A. A. AbuBaker, R. Qahwaji, S. S. Ipson, and M. S. Saleh, "One scan connected component labeling technique," *2007 IEEE International Conference on Signal Processing and Communications*, pp. 1283–1286, 2007.

[10] O. Abuzaghleh, B. D. Barkana, and M. Faezipour, "Noninvasive real-time automated skin lesion analysis system for melanoma early detection and prevention," *IEEE journal of translational engineering in health and medicine*, vol. 3, pp. 1–12, 2015.

[11] S. Agostinelli *et al.*, "Geant4: A simulation toolkit," *Nucl. Instrum. Meth.*, vol. A506, p. 250, 2003.

[12] S. Aiola, Y. Amhis, P. Billoir, B. K. Jashal, L. Henry, A. O. Campos, C. M. Benito, F. Polci, R. Quagliani, M. Schiller, and et al., "Hybrid seeding: A standalone track reconstruction algorithm for scintillating fibre tracker at lhcb," *Computer Physics Communications*, vol. 260, p. 107713, Mar 2021. [Online]. Available: http://dx.doi.org/10.1016/j.cpc.2020.107713

[13] A. Alfonso Albero, V. Batozskaya, S. Benson, M. Bjorn, S. R. Blusk, V. G. Chobanova, G. M. Ciezarek, A. Dziurda, C. Fitzpatrick, K. Govorkova, K. Heinicke, D. Hill, N. P. Jurik, X. Liu, O. Lupton, P. Mackowiak, C. Marin Benito, D. Martinez Santos, M. Materok, R. Matev, A. Modden, V. Mueller, A. Pearce, M. Ramos Pernas, N. A. Skidmore, E. A. Smith, S. Stahl, R. Vazquez Gomez, M. Wang, M. P. Whitehead, C. N. Weisser, A. Xu, and L. E. Yeomans, "Upgrade trigger selection studies," CERN, Geneva, Tech. Rep., Sep 2019. [Online]. Available: https://cds.cern.ch/record/2688423

[14] J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Dubois *et al.*, "Geant4 developments and applications," *IEEE Trans.Nucl.Sci.*, vol. 53, p. 270, 2006.

[15] G. Amadio, P. Canal, D. Piparo, and S. Wenzel, "Speeding up software with VecCore," *Journal of Physics: Conference Series*, vol. 1085, p. 032034, sep 2018. [Online]. Available: https://doi.org/10.1088%2F1742-6596%2F1085%2F3%2F032034

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

[17] O. Aumage, D. Barthou, and A. Honorat, "A Stencil DSEL for Single Code Accelerated Computing with SYCL," in *SYCL 2016 1st SYCL Programming Workshop during the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Barcelone, Spain, Mar. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01290099

[18] A. Badalov, D. Campora, G. Collazuol, M. Corvo, S. Gallorini, A. Gianelle, E. Golobardes, D. Lucchesi, A. Lupato, N. Neufeld, R. Schwemmer, L. Sestini, and X. Vilasis-Cardona, "GPGPU opportunities at the LHCb trigger," CERN, Geneva, Tech. Rep. LHCb-PUB-2014-034. CERN-LHCb-PUB-2014-034, May 2014. [Online]. Available: https://cds.cern.ch/record/1698101

[19] A. P. Badalov, "Coprocessor integration for real-time event processing in particle physics detectors," 2016.

[20] J. Barnat, P. Bauch, L. Brim, and M. Ceška, "Computing strongly connected components in parallel on CUDA," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 544–555.

[21] G. Barnes, R. Brown, M. Kato, D. Kuck, D. Slotnick, and R. Stokes, "The ILLIAC IV Computer," *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 746–757, 1968.

[22] I. Belyaev *et al.*, "Handling of the generation of primary events in Gauss, the LHCb simulation framework," *J. Phys. Conf. Ser.*, vol. 331, p. 032047, 2011.

[23] S. Benson, V. Gligorov, M. A. Vesterinen, and J. M. Williams, "The LHCb turbo stream," *Journal of Physics: Conference Series*, vol. 664, no. 8, p. 082004, dec 2015. [Online]. Available: https://doi.org/10.1088/1742-6596/664/8/082004

[24] T. Bird, T. Britton, O. Callot, V. Coco, P. Collins, T. Evans, T. Head, K. Hennessy, W. Hulsbergen, D. Hynds, P. Jalocha, M. John, T. Ketel, M. Kucharczyk, D. Martinez-Santos, W. Qian, K. Rinnert, H. Schindler, T. Skwarnicki, H. Snoek, P. Tsopelas, and D. Vieira, "VP Simulation and Track Reconstruction," CERN, Geneva, Tech. Rep. LHCb-PUB-2013-018. CERN-LHCb-PUB-2013-018, Oct 2013. [Online]. Available: https://cds.cern.ch/record/1620453

[25] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Toward reliable experiments on the performance of connected components labeling algorithms," *Journal of Real-Time Image Processing (JRTIP)*, pp. 1–16, 2018.

[26] C. Bozzi, S. Ponce, and S. Roiser, "The core software framework for the LHCb Upgrade," *EPJ Web Conf.*, vol. 214, p. 05040, 2019.

[27] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[28] L. Cabaret and L. Lacassagne, "A review of world's fastest connected component labeling algorithms : Speed and energy estimation," in *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2014, pp. 1–8.

[29] ——, "What is the world's fastest connected component labeling algorithm ?" in *IEEE International Workshop on Signal Processing Systems (SiPS)*, 2014, pp. 97–102.

[30] L. Cabaret, L. Lacassagne, and D. Etiemble, "Distanceless label propagation: an efficient direct connected component labeling algorithm for GPUs," in *IEEE International Conference on Image Processing Theory, Tools and Applications (IPTA)*, 2017, pp. 1–8.

[31] ——, "Distanceless label propagation: an efficient direct connected component labeling algorithm for GPUs," in *International GPU Technical Conference (GTC)*, 2017.

[32] ——, "Parallel Light Speed Labeling for connected component analysis on multi-core processors," *Journal of Real Time Image Processing*, vol. 15, no. 1, pp. 173–196, 2018.

[33] O. Callot, "Velo tracking for the High Level Trigger," CERN, Geneva, Tech. Rep. LHCb-2003-027, Apr 2003. [Online]. Available: http://cds.cern.ch/record/691694

[34] ——, "Online Pattern Recognition," CERN, Geneva, Tech. Rep. LHCb-2004-094. CERN-LHCb-2004-094, Oct 2004. [Online]. Available: http://cds.cern.ch/record/800610

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

[36] D. H. Cámpora Pérez, N. Neufeld, and A. Riscos Núñez, "A fast local algorithm for track reconstruction on parallel architectures," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019, pp. 698–707.

[37] D. H. Campora Perez, "LHCb Kalman Filter cross architectures studies," Oct 2016. [Online]. Available: https://cds.cern.ch/record/2229971

[38] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, and C. Jego, "MIPP: a Portable C++ SIMD Wrapper and its use for Error Correction Coding in 5G Standard," in *The 4th Workshop on Programming Models for SIMD/Vector Processing (WPMVP 2018)*. Vienna, Austria: ACM Press, Feb. 2018. [Online]. Available: https://hal.inria.fr/hal-01888010

[39] CERN, "LHC Guide," Mar 2017, cERN Document Server. [Online]. Available: https://cds.cern.ch/record/2255762

[40] F. Chang, C.-J. Chen, and C.-J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Comput. Vis. Image Underst.*, vol. 93, pp. 206–220, 2004.

[41] J. Chassery and A. Montanvert, *Géometrie discrète en analyse d'image*. Traité des Nouvelles technologies, Hermes, 1991.

[42] H. Chen, N. Rambaux, and J. Vaubaillon, "Accuracy of meteor positioning from space- and ground-based observations," *Astronomy and Astrophysics - A&A*, vol. 642, p. L11, Oct. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02963290

[43] W. Chen, M. L. Giger, and U. Bick, "A fuzzy c-means (FCM)-based approach for computerized segmentation of breast lesions in dynamic contrast-enhanced mr images," *Academic radiology*, vol. 13, no. 1, pp. 63–72, 2006.

[44] F. Colas *et al.*, "FRIPON: a worldwide network to track incoming meteoroids," *Astronomy and Astrophysics - A&A*, vol. 644, no. 6, p. A53, Dec. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03097279

[45] L. Collaboration, "LHCb Tracker Upgrade Technical Design Report," Tech. Rep., Feb 2014. [Online]. Available: https://cds.cern.ch/record/1647400

[46] H. Cragon and W. Watson, "The TI Advanced Scientific Computer," *Computer*, vol. 22, no. 1, pp. 55–64, 1989.

[47] D. H. Cámpora Pérez, "Optimization of high-throughput real-time processes in physics reconstruction," Nov 2019. [Online]. Available: https://idus.us.es/handle/11441/91352

[48] D. H. Cámpora Pérez, N. Neufeld, and A. Riscos Núñez, "Search by triplet: An efficient local track reconstruction algorithm for parallel architectures," *Journal of Computational Science*, vol. 54, p. 101422, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750321001071

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

[50] J. Denoulet, G. Mostafaoui, L. Lacassagne, and A. Mérigot, "Implementing motion markov detection on general purpose processor and associative mesh," in *Computer Architecture and Machine Perception (CAMP)*. IEEE, 2005.

[51] D. Etiemble, "45-year cpu evolution: one law and two equations," 2018.

[52] W. Farhat, H. Faiedh, C. Souani, and K. Besbes, "Real-time embedded system for traffic sign recognition based on ZedBoard," *Journal of Real-Time Image Processing*, vol. 16, no. 5, pp. 1813–1823, 2019.

[53] R. J. Fisher and H. G. Dietz, "Compiling for simd within a register," in *Languages and Compilers for Parallel Computing*, S. Chatterjee, J. F. Prins, L. Carter, J. Ferrante, Z. Li, D. Sehr, and P.-C. Yew, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 290–305.

[54] M. Gottschlag and F. Bellosa, "Mechanism to Mitigate AVX-Induced Frequency Reduction," *arXiv e-prints*, p. arXiv:1901.04982, Dec 2018.

[55] S. Graillat, F. Jézéquel, and R. Picot, "Numerical Validation of Compensated Algorithms with Stochastic Arithmetic," *Applied Mathematics and Computation*, vol. 329, pp. 339–363, Jul. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01367769

[56] C. Grana, F. Bolelli, L. Baraldi, and R. Vezzani, "YACCLAB - Yet Another Connected Components Labeling Benchmark," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, 2016, pp. 3109–3114.

[57] C. Grana, D. Borghesani, and R. Cucchiara, "Fast block based connected components labeling," in *2009 16th IEEE International Conference on Image Processing (ICIP)*, 2009, pp. 4061–4064.

[58] ——, "Optimized Block-Based Connected Components Labeling With Decision Trees," *IEEE Transactions on Image Processing*, vol. 19, no. 6, pp. 1596–1609, 2010.

[59] P. Guillou, B. Pin, F. Coelho, and F. Irigoin, "A Dynamic to Static DSL Compiler for Image Processing Applications," 2017, version longue de <hal-01352808¿, article présenté à "19th Workshop on Compilers for Parallel Computing", Valladolid. [Online]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-01665055

[60] S. Gupta, D. Palsetia, M. A. Patwary, A. Agrawal, and A. Choudhary, "A new parallel algorithm for two-pass connected component labeling," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2014, pp. 1355–1362.

[61] O. Haggui, C. Tadonki, L. Lacassagne, F. Sayadi, and B. Ouni, "harris corner detection on a numa manycore."

[62] S. Happy and A. Routray, "Automatic facial expression recognition using features of salient facial patches," *IEEE transactions on Affective Computing*, vol. 6, no. 1, pp. 1–12, 2014.

[63] R. Haralick, "Some neighborhood operations," in *Real-Time Parallel Computing Image Analysis*. Plenum Press, 1981, pp. 11–35.

[64] G. Harnisch, A. Gozillon, R. Keryell, L.-Y. Yu, R. Wittig, and L. Forget, "SYCL for Vitis 2020.2: SYCL & C++20 on Xilinx FPGA," Apr. 2021, iWOCL SYCLCon 2021 : 9th International Workshop on OpenCL and SYCL, SYCLCon ; Conference date: 28-04-2021 Through 29-04-2021. [Online]. Available: https://www.iwocl.org/

[65] M. Harris, "https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/," NVIDIA, 2013. [Online]. Available: https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/

[66] ——, "https://devblogs.nvidia.com/using-shared-memory-cuda-cc/," NVIDIA, 2013. [Online]. Available: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

[67] ——, "https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/," NVIDIA, 2013. [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[68] ——, "https://developer.nvidia.com/blog/six-ways-saxpy/," NVIDIA, 2013. [Online]. Available: https://developer.nvidia.com/blog/six-ways-saxpy/

[69] ——, "https://developer.nvidia.com/blog/simple-portable-parallel-c-hemi-2/," NVIDIA, 2015. [Online]. Available: https://developer.nvidia.com/blog/simple-portable-parallel-c-hemi-2/

[70] C. Hasse, "Alternative approaches in the event reconstruction of LHCb," 2019, presented 12 Dec 2019. [Online]. Available: https://cds.cern.ch/record/2706588

[71] L. He, Y. Chao, and K. Suzuki, "An efficient first-scan method for label-equivalence-based labeling algorithms," *Pattern Recognition Letters*, vol. 31, no. 1, pp. 28–35, 2010.

[72] ——, "A new two-scan algorithm for labeling connected components in binary images," in *Proceedings of the World Congress on Engineering*, W. Congress, Ed., vol. 2, 2012, pp. p1141–1146.

[73] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao, "The connected-component labeling problem: a review of state-of-the-art algorithms," *Pattern Recognition*, vol. 70, pp. 25–43, 2017.

[74] ——, "The connected-component labeling problem: a review of state-of-the-art algorithms," *Pattern Recognition*, vol. 70, pp. 25–43, 2017.

[75] A. Hennequin, B. Couturier, V. Gligorov, S. Ponce, R. Quagliani, and L. Lacassagne, "A fast and efficient SIMD track reconstruction algorithm for the LHCb upgrade 1 VELO-PIX detector," *Journal of Instrumentation*, vol. 15, no. 06, pp. P06 018–P06 018, jun 2020. [Online]. Available: https://doi.org/10.1088/1748-0221/15/06/p06018

[76] A. Hennequin, Q. L. Meunier, L. Lacassagne, and L. Cabaret, "A new direct connected component labeling and analysis algorithm for GPUs," in *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2018, pp. 1–6.

[77] A. Hennequin, B. Couturier, V. V. Gligorov, and L. Lacassagne, "SparseCCL: Connected Components Labeling and Analysis for sparse images," in *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2019, pp. 65–70.

[78] A. Hennequin and L. Lacassagne, "A new Direct Connected Component Labeling and Analysis Algorithm for GPUs," in *GPU Technology Conference (GTC)*, San Jose, United States, Mar. 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02198012

[79] A. Hennequin, L. Lacassagne, and I. Masliah, "Étiquetage et analyse en composantes connexes sur GPUs," in *COMPAS*, Anglet, France, Jun. 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02179411

[80] A. Hennequin, I. Masliah, and L. Lacassagne, "Designing efficient simd algorithms for direct connected component labeling," in *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP'19. New York, NY, USA: ACM, 2019, pp. 4:1–4:8. [Online]. Available: http://doi.acm.org/10.1145/3303117.3306164

[81] K. Hennessy and LHCb VELO Upgrade Collaboration, "LHCb VELO upgrade," *Nuclear Instruments and Methods in Physics Research A*, vol. 845, pp. 97–100, Feb 2017.

[82] U. H. Hernandez-Belmonte, V. Ayala-Ramirez, and R. E. Sanchez-Yanez, "Enhancing CCL Algorithms by Using a Reduced Connectivity Mask," in *Pattern Recognition*, J. A. Carrasco-Ochoa, J. F. Martínez-Trinidad, J. S. Rodríguez, and G. S. di Baja, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 195–203.

[83] W. Herr and B. Muratori, "Concept of luminosity," 2006. [Online]. Available: https://cds.cern.ch/record/941318

[84] W. D. Hillis, *The connection machine*. MIT Press, 1989.

[85] R. Hockney, "Characterization of parallel computers and algorithms," *Computer Physics Communications*, vol. 26, no. 3, pp. 285 – 291, 1982. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0010465582901187

[86] P. V. C. Hough, "Machine Analysis of Bubble Chamber Pictures," *Conf. Proc.*, vol. C590914, pp. 554–558, 1959.

[87] D. Hutchcroft, "VELO Pattern Recognition," CERN, Geneva, Tech. Rep. LHCb-2007-013. CERN-LHCb-2007-013, Mar 2007. [Online]. Available: https://cds.cern.ch/record/1023540

[88] W. W. Hwu, Ed., *GPU Computing Gems*. Morgan Kaufman, 2001, ch. 35: Connected Component Labeling in CUDA.

[89] F. Irigoin, P. Jouvelot, and R. Triolet, "Author Retrospective for Semantical Interprocedural Parallelization: An Overview of the PIPS Project," ser. International Conference on Supercomputing, U. Banerjee, Ed., vol. 25th Anniversary Volume. France: Utpal Banerjee, Apr. 2014, pp. 12–14. [Online]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-00984684

[90] K. A. Joshi and D. G. Thakore, "A survey on moving object detection and tracking in video surveillance system," *International Journal of Soft Computing and Engineering*, vol. 2, no. 3, pp. 44–48, 2012.

[91] F. Jézéquel and J.-M. Chesneaux, "CADNA: a library for estimating round-off error propagation," *Computer Physics Communications*, vol. 178, no. 12, pp. 933–955, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465508000775

[92] P. Karpiński and J. McDonald, "A High-performance Portable Abstract Interface for Explicit SIMD Vectorization," in *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'17. New York, NY, USA: ACM, 2017, pp. 21–28. [Online]. Available: http://doi.acm.org/10.1145/3026937.3026939

[93] S. V. Kartik, B. Couturier, M. Clemencic, and N. Neufeld, "Measurements of the LHCb software stack on the ARM architecture," *J. Phys. Conf. Ser.*, vol. 513, p. 052014, 2014.

[94] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[95] R. Keryell, R. Reyes, and L. Howes, "Khronos SYCL for OpenCL: A Tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2791321.2791345

[96] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, "Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages," in *25th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2012)*, vol. 7760. Tokyo, Japan: Springer Berlin Heidelberg, Sep. 2012, pp. pp 174–189, 15 pages. [Online]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-00742536

[97] N. Khan, I. Ahmed, M. Kiran, H. Rehman, S. Din, A. Paul, and A. G. Reddy, "Automatic segmentation of liver & lesion detection using h-minima transform and connecting component labeling," *Multimedia Tools and Applications*, vol. 79, no. 13, pp. 8459–8481, 2020.

[98] M. Klaiber, D. Bailey, and S. Simon, "A single cycle parallel multi-slice connected components analysis hardware architecture," *Journal of Real-Time Image Processing*, 2016.

[99] Y. Komura, "Gpu-based cluster-labeling algorithm without the use of conventional iteration: application to swendsen-wang multi-cluster spin flip algorithm," *Computer Physics Communications*, pp. 54–58, 2015.

[100] M. Kretz and V. Lindenstruth, "Vc: A c++ library for explicit vectorization," *Softw. Pract. Exper.*, vol. 42, no. 11, pp. 1409–1430, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1149

[101] L. Lacassagne, D. Etiemble, A. Hassan-Zahraee, A. Dominguez, and P. Vezolle, "High level transforms for SIMD and low-level computer vision algorithms," in *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, 2014, pp. 49–56.

[102] L. Lacassagne and B. Zavidovique, "Light Speed Labeling: Efficient connected component labeling on RISC architectures," *Journal of Real-Time Image Processing*, vol. 6, no. 2, pp. 117–135, 2011.

[103] L. Lacassagne, L. Cabaret, D. Etiemble, F. Hebbache, and A. Petreto, "A new SIMD iterative connected component labeling algorithm," in *Principles and Practice of Parallel Programming / WVMVP*. Barcelone, Spain: ACM, Mar. 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01361101

[104] L. Lacassagne, A. Manzanera, J. Denoulet, and A. Mérigot, "High performance motion detection: some trends toward new embedded architectures for vision systems," *Journal of Real-Time Image Processing*, vol. 4, no. 2, pp. 127–146, Jun. 2009. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01131002

[105] L. Lacassagne, A. Manzanera, and A. Dupret, "Motion detection: Fast and robust algorithms for embedded systems," in *International Conference on Image Processing (ICIP)*, Le Caire, Egypt, Nov. 2009. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01130889

[106] D. J. Lange, "The EvtGen particle decay simulation package," *Nucl. Instrum. Meth. A*, vol. 462, pp. 152–155, 2001.

[107] C. Lauter, "A new open-source simd vector libm fully implemented with high-level scalar c," in *2016 50th Asilomar Conference on Signals, Systems and Computers*, 2016, pp. 407–411.

[108] F. Lemaitre, "Tracking haute frequence pour architectures SIMD : optimisation de la reconstruction LHCb," Feb 2019, presented 13 Feb 2019. [Online]. Available: https://cds.cern.ch/record/2668250

[109] F. Lemaitre, B. Couturier, and L. Lacassagne, "Cholesky Factorization on SIMD multi-core architectures," *Journal of Systems Architecture*, Jun. 2017. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01550129

[110] ——, "Small SIMD Matrices for CERN High Throughput Computing," in *WPMVP 2018 Workshop on Programming Models for SIMD/Vector Processing*. Vienna, Austria: ACM Press, Feb. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01760260

[111] F. Lemaitre, A. Hennequin, and L. Lacassagne, "How to speed Connected Component Labeling up with SIMD RLE algorithms," in *Workshop on Programming Models for SIMD/Vector Processing (WPMVP@PPoPP)*, San Diego, Californie, United States, Feb. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02492824

[112] ——, "Taming Voting Algorithms on Gpus for an Efficient Connected Component Analysis Algorithm," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Toronto, Canada: IEEE, Jun. 2021, pp. 7903–7907. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03330414

[113] ——, "Taming Voting Algorithms on GPUs for an Efficient Connected Component Analysis Algorithm," in *GPU Technical Conference*, San Jose, United States, Apr. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03210776

[114] F. Lemaitre and L. Lacassagne, "Batched Cholesky Factorization for tiny matrices," in *Design and Architectures for Signal and Image Processing (DASIP)*. Rennes, France: ECSI, Oct. 2016, pp. 1–8. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01361204

[115] LHCb Collaboration, "LHCb VELO TDR: Vertex locator. Technical design report," CERN, Tech. Rep. CERN-LHCC-2001-011, 2001.

[116] ——, *LHCb trigger system: Technical Design Report*, ser. Technical design report. LHCb. Geneva: CERN, 2003, revised version number 1 submitted on 2003-09-24 12:12:22. [Online]. Available: http://cds.cern.ch/record/630828

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

[118] ——, "Measurement of the track reconstruction efficiency at LHCb. Measurement of the track reconstruction efficiency at LHCb," *JINST*, vol. 10, no. CERN-LHCB-DP-2013-002. CERN-LHCB-DP-2013-002. LHCB-DP-2013-002, p. P02007. 24 p, Aug 2014. [Online]. Available: https://cds.cern.ch/record/1748269

[119] ——, "Upgrade Software and Computing," CERN, Geneva, Tech. Rep. CERN-LHCC-2018-007. LHCB-TDR-017, Mar 2018. [Online]. Available: https://cds.cern.ch/record/2310827

[120] ——, "Comparison of particle selection algorithms for the LHCb Upgrade," Dec 2020. [Online]. Available: https://cds.cern.ch/record/2746789

[121] C. M. LHCb Collaboration, "Computing Model of the Upgrade LHCb experiment," CERN, Geneva, Tech. Rep., May 2018. [Online]. Available: https://cds.cern.ch/record/2319756

[122] Y. Lin and V. Grover, "https://devblogs.nvidia.com/using-cuda-warp-level-primitives/," NVIDIA, 2018. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

[123] G. Litjens, C. I. Sánchez, N. Timofeeva, M. Hermsen, I. Nagtegaal, I. Kovacs, C. Hulsbergen-Van De Kaa, P. Bult, B. Van Ginneken, and J. Van Der Laak, "Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis," *Scientific reports*, vol. 6, 2016.

[124] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *Transactions on Modeling and Computer simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[125] N. Maurice, J. Sopena, and L. Lacassagne, "Un nouvel algorithme efficace de Split & Merge pour systèmes embarqués," in *COMPAS 2021 - Conférence francophone d'informatique en Parallélisme, Architecture et Système*, Lyon, France, Jul. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03330463

[126] M. Millet, N. Rambaux, A. Petreto, F. Lemaitre, and L. Lacassagne, "Détection temps réel de météores à bord d'un nanosatellite, application au projet Meteorix," in *ORASIS 2021*. Saint Ferréol, France: Centre National de la Recherche Scientifique [CNRS], Sep. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03339645

[127] E. Mobs, "The CERN accelerator complex - August 2018. Complexe des accélérateurs du CERN - Août 2018," Aug 2018, general Photo. [Online]. Available: https://cds.cern.ch/record/2636343

[128] E. Mozef, S. Weber, J. Jaber, and E. Tisserand, "Parallel architecture dedicated to connected component analysis," in *Proceedings of 13th International Conference on Pattern Recognition*, vol. 4. IEEE, 1996, pp. 699–703.

[129] S. Nazlibilek, D. Karacor, T. Ercan, M. H. Sazli, O. Kalender, and Y. Ege, "Automatic segmentation, counting, size determination and classification of white blood cells," *Measurement*, vol. 55, pp. 58–65, 2014.

[130] Nvidia, *CUDA Toolkit Documentation 11.3.1.* Nvidia, 2021.

[131] M. Paterno, "Calculating efficiencies and their uncertainties," 2004.

[132] M. Patwary, J. Blair, and F. Manne, "Experiments on union-find algorithms for the disjoint-set data structure," in *International symposium on experimental algorithms (SEA)*, L. . Springer, Ed., 2010, pp. 411–423.

[133] A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. Meunier, and L. Lacassagne, "Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU," in *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS 2018)*, Toulouse, France, Jul. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01835219

[134] ——, "Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU," GdR SOC2, Jun. 2018, poster. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01835240

[135] ——, "Energy and Execution Time Comparison of Optical Flow Algorithms on SIMD and GPU Architectures," in *Conference on Design and Architectures for Signal and Image Processing (Dasip 2018)*, Porto, Portugal, Oct. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01925886

[136] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–13.

[137] D. Piparo, V. Innocente, and T. Hauth, "Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions," *Journal of Physics: Conference Series*, vol. 513, no. 5, p. 052027, jun 2014. [Online]. Available: https://doi.org/10.1088/1742-6596/513/5/052027

[138] F. Pisani, T. Colombo, N. Neufeld, U. Marconi, R. Krawczyk, D. Galli, R. Schwemmer, and P. Durante, "Network simulation of a 40 MHz event building system for the LHCb experiment," *EPJ Web Conf.*, vol. 245, p. 01012. 8 p, 2020. [Online]. Available: http://cds.cern.ch/record/2756290

[139] D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[140] T. Poikela, M. D. Gaspari, J. Plosila, T. Westerlund, R. Ballabriga, J. Buytaert, M. Campbell, X. Llopart, K. Wyllie, V. Gromov, M. van Beuzekom, and V. Zivkovic, "VeloPix: the pixel ASIC for the LHCb upgrade," *Journal of Instrumentation*, vol. 10, no. 01, pp. C01 057–C01 057, jan 2015. [Online]. Available: https://doi.org/10.1088%2F1748-0221%2F10%2F01%2Fc01057

[141] L. Promberger, M. Clemencic, B. Couturier, A. B. Iartza, and N. Neufeld, "Porting the LHCb Stack from x86 (Intel) to aarch64 (ARM) and ppc64le (PowerPC)," *EPJ Web Conf.*, vol. 214, p. 05016, 2019.

[142] C. J. Purcell, "The control data star-100: Performance measurements," in *Proceedings of the May 6-10, 1974, National Computer Conference and Exposition*, ser. AFIPS '74. New York, NY, USA: Association for Computing Machinery, 1974, p. 385–387. [Online]. Available: https://doi.org/10.1145/1500175.1500257

[143] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[144] N. Rambaux, D. Galayko, G. Guignan, J. Vaubaillon, L. Lacassagne, P. Keckhut, A. C. Levasseur-Regourd, A. Hauchecorne, M. Birlan, G. Augarde, S. Barnier, S. Ben Kemmoum, A. Bigot, P. Boisse, M. Capderou, A. Chu, F. Colas, F. DESHOURS, Y. Fargeix, A. Hennequin, T. Koehler, M. Lumbroso, J.-F. Mariscal, D. Portela-Moreira, J. Raffard, J.-L. Rault, T. Romera, C. Tob, and B. Zanda, "METEORIX: a cubesat mission dedicated to the detection of meteors," in *COSPAR 2018, 42nd Assembly*, Pasadena, United States, Jul. 2018. [Online]. Available: https://hal-insu.archives-ouvertes.fr/insu-01851524

[145] S. F. Reddaway, "DAP—a Distributed Array Processor," *SIGARCH Comput. Archit. News*, vol. 2, no. 4, p. 61–65, Dec. 1973. [Online]. Available: https://doi.org/10.1145/633642.803971

[146] L. Riha and M. Mareboyana, "GPU accelerated one-pass algorithm for computing minimal rectangles of connected components," in *IEEE Workshop on Applications of Computer Vision*, 2010, pp. 479–484.

[147] L. Ristori, "An artificial retina for fast track finding," *Nucl. Instrum. Meth.*, vol. A453, pp. 425–429, 2000.

[148] A. Rosenfeld and J. Platz, "Sequential operator in digital pictures processing," *Journal of ACM*, vol. 13,4, pp. 471–494, 1966.

[149] K. Rupp, "https://github.com/karlrupp/microprocessor-trend-data," 2020. [Online]. Available: https://github.com/karlrupp/microprocessor-trend-data

[150] R. M. Russell, "The CRAY-1 Computer System," *Commun. ACM*, vol. 21, no. 1, p. 63–72, Jan. 1978. [Online]. Available: https://doi.org/10.1145/359327.359336

[151] J. Salau and J. Krieter, "Analysing the space-usage-pattern of a cow herd using video surveillance and automated motion detection," *Biosystems Engineering*, vol. 197, pp. 122–134, 2020.

[152] J. Seo, S. Chae, J. Shim, D. Kim, C. Cheong, and T.-D. Han, "Fast Contour-Tracing Algorithm Based on a Pixel-Following Method for Image Sensors," *Sensors*, vol. 16, no. 3, 2016. [Online]. Available: https://www.mdpi.com/1424-8220/16/3/353

[153] T. Sjöstrand, S. Mrenna, and P. Skands, "A brief introduction to PYTHIA 8.1," *Comput. Phys. Commun.*, vol. 178, pp. 852–867, 2008.

[154] T. Sjöstrand, S. Mrenna, and P. Skands, "Pythia 6.4 physics and manual," *Journal of High Energy Physics*, vol. 2006, no. 05, p. 026–026, May 2006. [Online]. Available: http://dx.doi.org/10.1088/1126-6708/2006/05/026

[155] F. Spagnolo, S. Perri, and P. Corsonello, "An efficient hardware-oriented single-pass approach for connected component analysis," *Sensors*, vol. 19, no. 14, 2019.

[156] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[157] J. W. Tang, N. Shaikh-Husin, U. U. Sheikh, and M. N. Marsono, "A linked list run-length-based single-pass connected component analysis for real-time embedded hardware," *Journal of Real-Time Image Processing*, vol. 15, no. 1, pp. 197–215, 2018.

[158] R. Ticse, D. H. Campora Perez, R. Schwemmer, and N. Neufeld, "An SIMD parallel version of the VELO Pixel track reconstruction for the LHCb upgrade," CERN, Geneva, Tech. Rep. LHCb-PUB-2013-007. CERN-LHCb-PUB-2013-007. LHCb-INT-2013-030, Jun 2013. [Online]. Available: https://cds.cern.ch/record/1554078

[159] B. E. Tweddle and A. Saenz-Otero, "Relative computer vision-based navigation for small inspection spacecraft," *Journal of Guidance, Control, and Dynamics*, vol. 38, no. 5, pp. 969–978, 2015.

[160] G.-J. van den Braak, C. Nugteren, B. Mesman, and H. Corporaal, "Gpu-vote: A framework for accelerating voting algorithms on gpu," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 945–956.

[161] F. Veillon, "One pass computation of morphological and geometrical properties of objects in digital pictures," *Signal Processing*, vol. 1,3, pp. 175–179, 1979.

[162] L. Vincent and P. Soille, "Watersheds in digital spaces: An efficient algorithm based on immersion simulations," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, no. 6, pp. 583–598, Jun. 1991. [Online]. Available: https://doi.org/10.1109/34.87344

[163] H.-M. Weng and C.-T. Chiu, "Resource efficient hardware implementation for real-time traffic sign recognition," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 1120–1124.

[164] E. Westphal, "https://developer.nvidia.com/blog/voting-and-shuffling-optimize-atomic-operations/," NVIDIA, 2015. [Online]. Available: https://developer.nvidia.com/blog/voting-and-shuffling-optimize-atomic-operations/

[165] M. Williams, "Upgrade of the LHCb VELO detector," *Journal of Instrumentation*, vol. 12, no. 01, pp. C01 020–C01 020, jan 2017. [Online]. Available: https://doi.org/10.1088%2F1748-0221%2F12%2F01%2Fc01020

[166] K. Wu, E. Otoo, and A. Shoshani, "Optimizing connected component labeling algorithms," *Pattern Analysis and Applications*, 2008.

[167] C. Zhao, G. Duan, and N. Zheng, "A hardware-efficient method for extracting statistic information of connected component," *Journal of Signal Processing Systems*, vol. 88, no. 1, pp. 55–65, 2017.

[168] G. Ziegler and A. Rasmusson, "Efficient volume segmentation on the GPU," in *GPU Technology Conference*, Nvidia, Ed., 2010, pp. 1–44.