POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Design of an high-performance tracking algorithm optimised for the Inner Tracking System of the ALICE experiment

Supervisor
Prof. Stefania Bufalino

Co-supervisor:
Prof. Michelangelo Agnello

Candidate
Iacopo Colonnelli

Ottobre 2017

Contents

List of Tables

List of Figures

Abstract

The main goal of this thesis work is to investigate optimisation possibilities offered by a total or partial migration of the track reconstruction algorithm for Inner Tracking System (ITS) Upgrade to a Graphics Processing Unit (GPU) architecture.

The thesis project has been developed in the framework of A Large Ion Collider Experiment (ALICE) at CERN that after a pause in 2018-2019 (indicated as Long Shutdown 2 (LS2)), will be equipped with an upgraded silicon pixel detector made up of seven layers with a cylindrical geometry to collect new data in the high luminosity era of the Large Hadron Collider (LHC). This study and implementation are mandatory for the experiment to face the enhanced rate of Pb–Pb collisions of up to $6 \times 10^{27} cm^{-2} s^{-1}$ (50 kHz interaction rate) delivered by the LHC during the Run 3 that will follow the LS2 period. Such an improvement leads to an estimated data throughput from the ALICE detector greater than 1 TB/s for Pb–Pb events, that is two orders of magnitude higher than the data throughput of the present running conditions. It is therefore necessary to achieve an important reduction of the data volume as early as possible during the data-flow: this operation is performed by the O^2 (Online/Offline) facility, that reconstructs and filter the data synchronously with the data taking process.

Such a requirement makes it necessary to considerably improve the performances of the current reconstruction algorithms, in order to process that huge amount of data without violating the strict time constraints imposed by the synchronicity. Since part of the O^2 facility will be equipped with GPUs, it is worth to investigate a partial migration of algorithms to a GPU architecture, in particular those steps that are quite slow but easily parallelizable.

This thesis work presents an optimised version of the ITS detector tracklet reconstruction algorithm, that can run on both Central Processing Unit (CPU) and GPU Compute Unified Device Architecture (CUDA) architectures. Track finding and fitting are two of the most computationally challenging problems for event reconstruction in particle physics [1]. Indeed, the track reconstruction algorithm must be able to cope with a very high combinatorial, with thousands of clusters in each detector layer. Moreover, the track reconstruction in the ITS Upgrade is particularly challenging, because on the one hand the track reconstruction will be done online and, on the other hand, the experiment aims to reconstruct all the Pb–Pb collisions that will occur at a rate of 50 kHz [2].

After a brief theoretical analysis of the main algorithm steps and the presentation of a performance benchmark for the serial CPU version of the code, the thesis describes in details the techniques used to realise the CUDA version and the obtained results in terms of speedup.

Chapter 1

The ALICE Experiment

ALICE is a general-purpose detector running at the LHC with the mission to explore the features of the heavy-ion collisions delivered by the LHC. It is designed to study the physics of the strong interaction sector of the Standard Model and in particular the Quark-Gluon Plasma (QGP), using p–p, p–Pb and Pb–Pb collisions at unprecedented energy (order of TeV) and high density and temperature. After the LS2 in 2018-2019, the ALICE experiment will be equipped with an upgraded detector, in order to face the enhanced rate of heavy-ion collisions and collect all the statistics delivered by the LHC.

1.1 Introduction

1.1.1 CERN accelerator complex

ALICE is one of the four major detectors installed at the LHC, together with A Toroidal LHC Apparatus (ATLAS), Compact Muon Solenoid (CMS) and Large Hadron Collider beauty (LHCb).

The LHC is the last element in the chain of machines that accelerate particles to increasingly higher energy, known as CERN accelerator complex [4]. Figure 1.1 shows the CERN LHC ecosystem. The four major experiments, including ALICE, are highlighted with a yellow circle on the bigger ring, while a bunch of other detectors lay on the smaller accelerators. More specifically, the ALICE detector and related facilities are located at the Point 2 of the LHC tunnel, in the district of St. Genis-Pouilly, France.

Protons (p) and lead ions (Pb) follow two different acceleration paths before converging into the LHC. Protons are formed into a container of hydrogen gas, where atoms are immersed into an electric field to strip off their electrons, and injected into Linear Accelerator (LINAC) 2, where they reach an energy of 50 MeV. Then the beam enters sequentially into Proton Synchrotron Booster (PSB), Proton Synchrotron (PS) and Super Proton Synchrotron (SPS), reaching an energy of respectively 1.4 GeV, 25 GeV and 450 GeV, before it finally reaches the LHC. Lead ions start instead from a container of vaporised Pb and

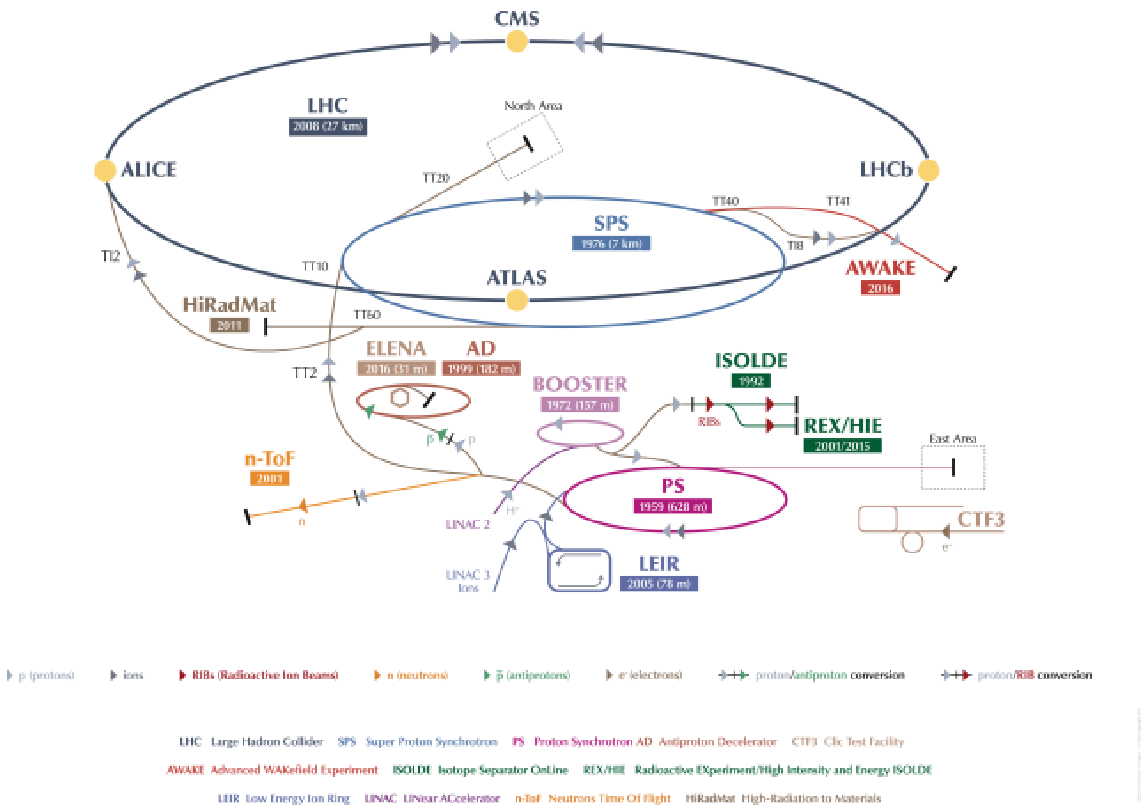


Figure 1.1: The CERN LHC ecosystem. The LHC is the last ring (dark blue line) in a complex chain of particle accelerators. The smaller machines are used in a chain to boost the particles to their final energies and provide beams to a whole set of smaller experiments. [3]

are injected sequentially into LINAC3 and Low Energy Ion Ring (LEIR), where they are splitted into 4 bunches, each containing 2.2×10^8 ions. Bunches are accelerated in groups of two until they reach an energy of 72 MeV, then they are sent into PS and follow the same acceleration chain described for protons.

Inside the LHC, two beams circulate in two different pipes, one clockwise and the other anticlockwise, until they reach an energy of 6.5 TeV. Then they are forced to collide in four points, where the four aforementioned experiments are installed.

1.1.2 Coordinate system

The ALICE coordinate system is a right-handed orthogonal Cartesian system with the origin corresponding to the beams interaction point inside the LHC [5]. Figure 1.2 gives a clear graphic representation of the ALICE coordinate system, which defines several useful components as follows:

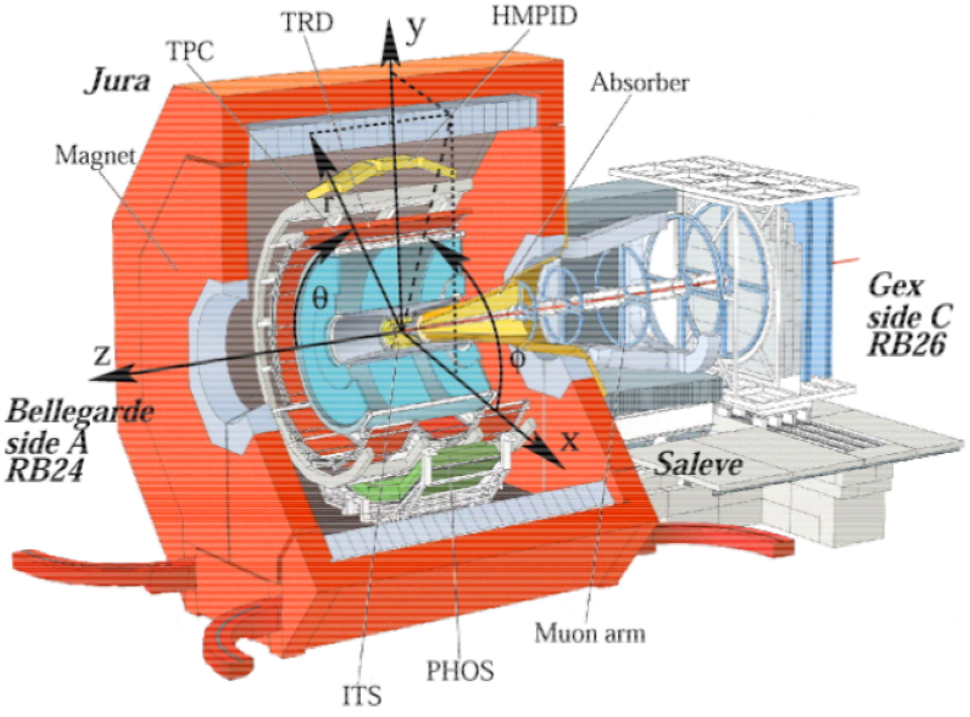


Figure 1.2: Definition of the ALICE coordinate system [5]. Some visual aids are reported in order to better understand axis directions and senses: x-axis goes from Jura mountain to Saleve mountain, while z-axis goes from the town of Bellegarde to the town of Gex, or from RB24 building to RB26 building. Part of the ALICE detectors side labelling system is also reported, with upstream detectors labeled as A and downstream ones labeled as C.

- x-axis, that is perpendicular to the mean beam direction, aligned with the local horizontal and pointing to the centre of the accelerator
- y-axis, that is perpendicular to the x axis and the mean local beam direction, pointing upward
- z-axis, that is parallel to the mean beam direction
- azimuthal angle ϕ, that for an observer standing at positive z increases counterclockwise from x-axis ($\phi = 0$) to y-axis ($\phi = \frac{\pi}{2}$)
- polar angle θ, that increases from positive z-axis ($\theta = 0$) to (x, y) plane ($\theta = \frac{\pi}{2}$). Polar angle coordinate values are usually reported in units of pseudorapidity[1] η

[1]Spatial coordinate that describes the angle of of a particle with respect to its beam axis. If θ is the

- spherical coordinate $r = \sqrt{x^2 + y^2 + z^2}$

1.1.3 Experiment layout

According to [6], the design of the ALICE detector was mainly driven by the high particle multiplicity in central[2] Pb–Pb events; originally a number of particles per pseudorapidity unit ranging between 2000 and 8000 was estimated, that was up to three orders of magnitude larger than in a typical p–p interaction at the same energy. On the contrary, the interaction rate with nuclear beams was low in the beginning (only 10 kHz for Pb–Pb), allowing the use of slow but high granularity detectors like Time Projection Chamber (TPC) and Silicon Drift Detector (SDD), that can only cope with a maximum trigger rate of 1 kHz for minimum bias events.

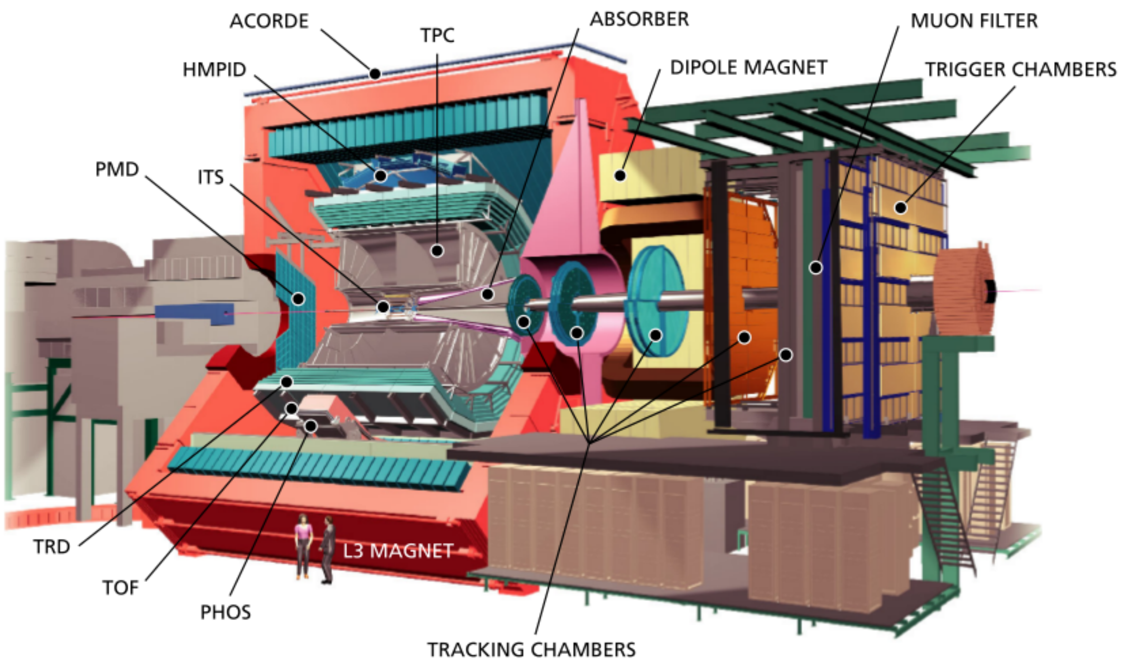


Figure 1.3: The ALICE experiment apparatus at the CERN LHC. For the sake of visibility, the HMPID detector is shown in the 12 o'clock position instead of the 2 o'clock position in which it is actually positioned. [7]

angle between the particle momentum and the positive direction of the beam axis, pseudorapidity is equal to $\eta = -\ln\left[\tan\left(\frac{\theta}{2}\right)\right]$

[2]In the literature events are classified into centrality classes corresponding to percentiles of the total hadronic interaction cross section of the colliding nuclei [2]. Central events have the highest track density.

Figure 1.3 shows the ALICE experiment layout in Run 2[3] It consists of a central barrel embedded in a large solenoid (L3 magnet) with magnetic field $B = 0.5T$, that measures hadrons, electrons and photons, and a forward muon spectrometer. The barrel contains, from the inside out, the Inner Tracking System (ITS), the Time Projection Chamber (TPC), three arrays of Time Of Flight (TOF) detectors for particle identification, the High Momentum Particle Identification (HMPID) based on Ring Imaging Cherenkov (RICH) counters, the Transition Radiation Detector (TRD) and two electromagnetic calorimeters, Photon Spectrometer (PHOS) and Electromagnetic Calorimeter (EMCAL). The forward muon spectrometer is instead a complex arrangement of absorbers, a dipole magnet and several Muon Tracking Chambers (MCHs) and Muon Trigger Systems (MTRs). On top of the magnet that surrounds the central barrel there is an array of scintillators, called ALICE Cosmic Rays Detector (ACORDE), that is used to trigger on cosmic rays.

Table 1.1 reports some details about the geometrical configuration and the main purposes of each detector in ALICE experiment. Several detectors are dedicated to Particle Identification (PID), because it plays an important role in a wide range of ALICE physics analyses. ITS, TPC, TOF detectors and HMPID are involved in hadrons identifications, while the TRD and the calorimeters provide dedicated electrons identification. Nevertheless, the most interesting feature for this thesis work is the ALICE tracking flow, that is described in the next section.

1.1.4 Tracking flow

Looking at the last column of table 1.1, it appears that tracking operations mainly take place in ALICE central barrel, more specifically into ITS, TPC and TRD detectors.

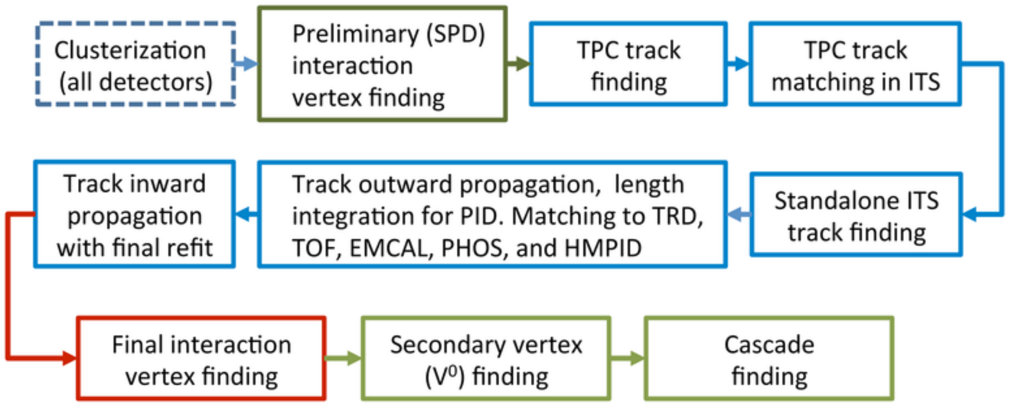

Clusterization (all detectors)
Preliminary (SPD) interaction vertex finding
TPC track finding
TPC track matching in ITS
Standalone ITS track finding
Track outward propagation, length integration for PID. Matching to TRD, TOF, EMCAL, PHOS, and HMPID
Track inward propagation with final refit
Final interaction vertex finding
Secondary vertex (V⁰) finding
Cascade finding

Figure 1.4: ALICE tracking flow in central barrel detectors. [8]

[3]Second LHC activity period, that started in 2015 and it will end on 2018.

Table 1.1: Details about the geometrical configuration and main design purposes of the ALICE detectors. This table has been realised by merging the description of the ALICE apparatus in [8] with the summary of detectors subsystem in [6]. The position column reports the radial distance from the beam axis for the central barrel detectors and the distance along z for the others. Where multiple values are specified, the detector is subdivided in several layers.

Detector	Acceptance		Position (m)	Main purpose
	Polar (η)	Azimuthal (ϕ)		
ITS	± 2, ± 1.4	full	0.039, 0.076	tracking, vertex
	± 0.9, ± 0.9	full	0.150, 0.239	tracking, PID
	± 0.97, ± 0.97	full	0.380, 0.430	tracking, PID
TPC	± 0.9 at r = 2.8 m ± 1.5 at r = 1.4 m	full	0.848, 2.466	tracking, PID
TRD	± 0.84	full	2.90, 3.68	tracking, $e^{\pm}$ id
TOF	± 0.9	full	3.78	PID
HMPID	± 0.6	$1.2 \leq \phi \leq 58.8$	5	PID
PHOS	± 0.12	$220 \leq \phi \leq 320$	4.6	photons
EMCAL	± 0.7	$80^{\circ} \leq \phi \leq 187^{\circ}$	4.36	photons and jets
ACORDE	± 1.3	$-60^{\circ} \leq \phi \leq 60^{\circ}$	8.5	cosmics
MCH	$-4.0 \leq \eta \leq -2.5$	full	-14.22,-5.36	muon tracking
MTR	$-4.0 \leq \eta \leq -2.5$	full	-17.12,-16.12	muon trigger
ZDC	$\lvert\eta\rvert \leq 8.8$	full	± 116	forward neutrons
	$6.5 \leq \lvert\eta\rvert \leq 7.5$	$\lvert\phi\rvert \leq 9.7^{\circ}$	± 116	forward protons
	$4.8 \leq \eta \leq 5.7$	$\lvert 2\phi\rvert \leq 32^{\circ}$	7.25	photons
PMD	$2.3 \leq \eta \leq 3.7$	full	3.64	photons
FMD	$3.62 \leq \eta \leq 5.03$ $1.7 \leq \eta \leq 3.68$ $-3.4 \leq \eta \leq -1.7$	full	3.2 0.752, 0.834 -0.752, -0.628	charged particles
V0	$2.8 \leq \eta \leq 5.1$ $-3.7 \leq \eta \leq -1.7$	full	3.4 -0.897	charged particles
T0	$4.61 \leq \eta \leq 4.92$ $-3.28 \leq \eta \leq -2.97$	full	3.75 -0.727	time, vertex

Figure 1.4 shows the tracking process across the various central barrel detectors as it works in Run 2. The procedure starts with a clusterisation phase, in which raw data are converted into *clusters* characterized by position, time and some additional parameters like energy loss in the crossed detectors and time of flight, together with their associated errors.

Then a preliminary determination of the interaction vertex is performed using clusters in the first two ITS layers. Although full tracks are needed to exactly estimate the position of the primary vertex, this preliminary step is performed because the primary vertex position is necessary to filter out clusters during the next phases of the tracking algorithm and to considerably speed up the whole process.

The interaction vertex is found as the space point to which the maximum number of cluster pairs, called *tracklets*, converge. The first vertex found is, by construction, also the one with the largest number of contributing tracklets and it is assumed to be the *primary vertex* of the event. The case when multiple interaction vertices are reconstructed is called *pile-up*.

Subsequently, track finding and fitting are performed both in the ITS and the TPC, following an inward-outward-inward scheme. The first track finding stage takes part in the TPC, the main tracking detector of the experiment central barrel, using a technique based on Kalman Filter [9]. Track seeds are initially built with the first two TPC clusters and the primary vertex point as the third member of the triplet and propagated inward. In subsequent steps, triplets are formed by three TPC clusters instead.

At each step, each seed is updated with the nearest cluster in a proximity cut. A special algorithm prevents multiple reconstructions of the same physical track, by limiting the fraction of possible common clusters in a pair of tracks and rejecting the worse of the two, according to a quality parameter based on the cluster density, number of clusters and momentum [8]. Tracks with at least 20 clusters (out of a maximum of 159) and that miss no more than half of the expected clusters are propagated to the inner radius of TPC detector, where a preliminary energy loss based PID is performed.

Reconstructed TPC tracks are then propagated to the outermost layer of the ITS, forming the seeds for the ITS track finding step. During this phase, an algorithm based on Kalman Filter, similar to the one just described for the TPC, is used to reconstruct track candidates. As a result, each TPC track is associated with a tree of track candidates of the ITS. The highest quality candidate (i.e. the one with the minimum χ^2) from each tree is added to the reconstructed event, forming an ITS+TPC track.

Figure 1.5 shows the TPC tracking efficiency[4] for both p–p and Pb–Pb collisions. Looking at the efficiency trend for low p_T, it is clear that it is impossible to track particles with

[4]According to [10], the tracking efficiency is defined as the probability of reconstructing an embedded simulated track in a data event, given that it could be reconstructed as an isolated track in a simulated event. If $N_{reco,iso}$ is the number of correctly reconstructed simulated tracks and $N_{reco,embed}$ is the number of correctly reconstructed tracks that were also reconstructed in the simulation, efficiency can be expressed as the ratio $\frac{N_{reco,embed}}{N_{reco,iso}}$

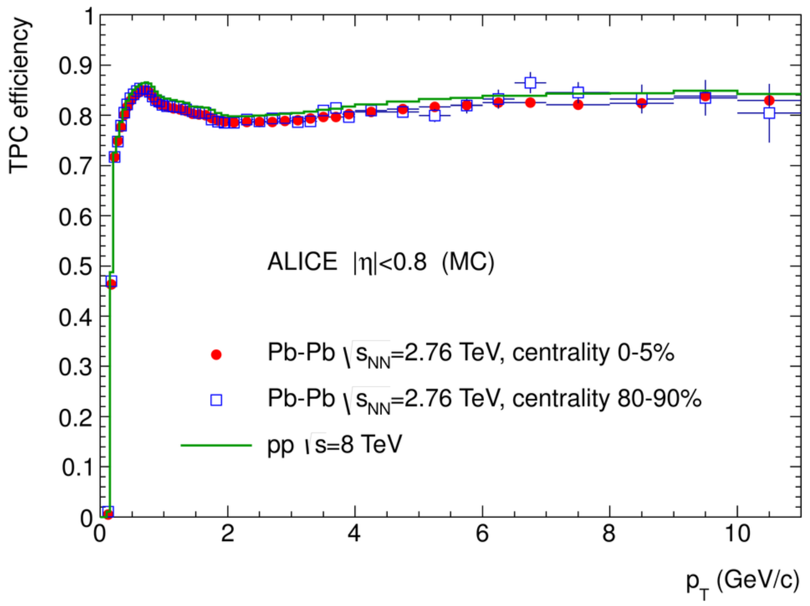


Figure 1.5: TPC track finding efficiency for primary particles in p–p and Pb–Pb collisions with respect to transverse momentum p_T. The results are obtained using as input data from a Monte Carlo simulation. [8]

$p_T \leq 200$ MeV/c using this method, because of the sharp drop due to energy loss and multiple scattering in the detector material. Therefore, an additional standalone ITS reconstruction step is performed with those clusters that were not used to build ITS+TPC tracks [8].

The ITS standalone tracking algorithm used in Run 2 is again based on a Kalman Filter pattern recognition strategy. Helical seeds are initially built up with two clusters from the three innermost layers of ITS and the primary vertex. Such seeds are then propagated to outer layers and updated with clusters filtered with a proximity cut. During the final step, all track candidates are refitted using a Kalman Filter and the best is retained. The entire procedure is repeated few times, gradually relaxing the cuts, to achieve better performances at low p_T. This strategy allows the tracking of particles with p_T down to about 80 MeV/c.

Once the ITS standalone reconstruction phase is completed, the Kalman Filter backward refitting takes place. During each step of this phase both the track length integral and the time of flight expected for various particle species are updated, in order to allow the subsequent particle identification phase to be performed by the TOF detector. When a track reaches the TRD, an attempt to match it with a TRD tracks is made, and the same goes for the TOF detector. Then, tracks are further propagated for matching with signal in outer detectors (namely EMCAL, PHOS and HMPID).

At this point, all tracks are propagated inwards to the innermost ITS layer with a last Kalman Filter refit, completing the inward-outward-inward scheme. These global tracks are used to find the final interaction vertex with an higher precision with respect to the initial position estimate.

Finally the secondary vertex reconstruction process takes place, in which vertices related to photon conversions and particle decays are located.

1.1.5 Long Shutdown 2 Upgrade

In July 2018, after more than 3 years of operation, the CERN accelerator complex will be stopped for about 18 months. According to [11], the main purpose for this LS2 is the upgrade of the LHC injectors, but also a full maintenance of all the accelerator equipments, a consolidation of part of the machine and some activities related to the LHC High Luminosity (HL-LHC) project [12] will take place.

In particular, the LHC will increase its luminosity for Pb–Pb collisions, reaching an instantaneous luminosity of 6×10^{27} cm^{-2}s^{-1}, namely an interaction rate of 50 kHz.

The ALICE detectors must then be upgraded in order to allow the readout of all the delivered interactions. Planned upgrades will enable ALICE to collect 10 nb^{-1} of Pb–Pb, recording about 10^{11} interactions, will enhance vertexing and tracking capabilities at low p_T and will allow data taking at higher rates.

According to [13], ALICE planned upgrades for the LS2 include the following features:

- a new beam pipe[5], with smaller diameter
- an upgraded TPC with Gas Electron Multiplier (GEM) detectors that replace the wire chambers presently used and a new pipelined readout electronics
- upgraded forward trigger detectors
- a new high resolution ITS, with a low-material budget , that will be described in section 1.2.2
- upgraded online systems and offline reconstruction and analysis framework, that will be described in section 1.3.2

1.2 Inner Tracking System

1.2.1 Run 2 detector design

The Inner Tracking System (ITS) is the innermost detector of the ALICE detector, composed of six high-resolution cylindrical silicon detectors located at radii between 39

[5]The tube, kept at ultrahigh vacuum, where particle beams travel and collide. Since all particle interactions take place inside this volume, interaction vertices are always located into the beam pipe.

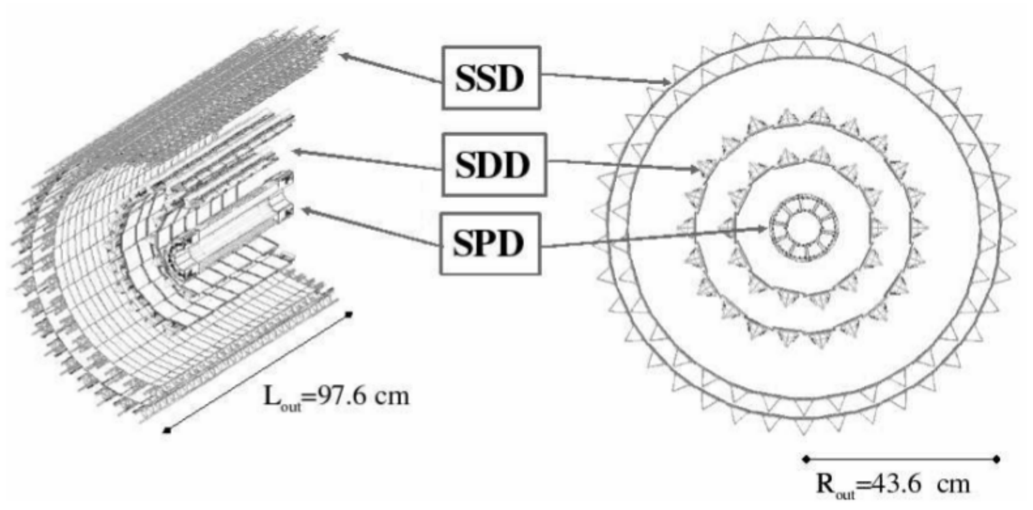


Figure 1.6: Layout of the Run 2 ITS detector [6]

mm and 430 mm. More specifically, as clearly shown in figure 1.6, the innermost two layers are equipped with two Silicon Pixel Detectors (SPDs), the following two layers with Silicon Drift Detectors (SDDs) and the two outer layers with Silicon Strip Detectors (SSDs). The ITS innermost layer is located at the minimum radius allowed by the size of the *beam pipe*, that is a beryllium cylinder with a radius of 3 cm, and provides a mechanical support to avoid relative motion during measurements. The outer radius is instead determined by the necessity to match the ITS tracks with the ones reconstructed in the TPC.

The ITS contributes to practically all physics topics addressed by the ALICE experiment because it is crucial to determine the point where the collisions happen. According to [7], its main tasks are:

- the localisation of the primary vertex with a resolution better than 100 μm
- the reconstruction of secondary vertices from decays of hyperons and D and B mesons
- the tracking and identification of particles with low momentum ($p_T \leq 100$ MeV/c), as described in section 1.1.4
- the improvement of momentum and angle resolution for the high-p_T particles which traverse also the TPC
- the reconstruction, albeit with limited momentum resolution, of the particles that traverse dead regions of the TPC

The design of ITS has been optimised for efficient track finding and high impact parameter[6] resolution, by taking into account the following factors:

- *Acceptance*: the ITS covers a pseudorapidity range of $|\eta| < 0.9$ for vertices located within ± 53 mm with respect to the nominal interaction point (the so called interaction diamond). The first layer covers a more extended pseudorapidity of $|\eta| < 1.98$ in order to provide, together with Forward Multiplicity Detectors (FMD), continuous coverage for the measurement of charged particle multiplicity which corresponds to the particles density of one collision.

- *Energy loss measurement*: ITS contribute to PID through the measurement of particle energy loss dE/dx. In order to apply the PID algorithm, at least four measurements are necessary, which implies that at least four layers out of the six need analogue readout.

- *Material budget*: the amount of material in the active volume has to be reduced to a minimum, in order to avoid as much as possible multiple scattering effects that dominate the momentum and impact parameter resolution for particles with low p_t. However, SDD and SSD must have a minimum thickness of approximately 300 μm to provide a reasonable signal-to-noise ratio (SNR) and they must partially overlap to cover the entire solid angle. This allows the ITS to achieve a relative momentum resolution better than 2% for pions with 100 MeV/c < p_T < 3 GeV/c.

- *Granularity and spatial precision*: the upper limit of the theoretical estimated track density is 8000 tracks per unit of η. This means that the ITS must be able to simultaneously detect more than 15000 tracks, with several millions of effective cells in each layer. Spatial resolution of the ITS detectors determines the impact parameter measurement resolution and is an essential element of momentum resolution for particles with $p_T > 3$ GeV/c. In order to satisfy the minimum resolution requirements, the ITS detectors have a spatial resolution of the order of a few tens of μm, with a best precision of 12 μm for the innermost detectors.

- *Radiation levels*: the total amount of radiation received by the ITS during the expected lifetime of the ALICE experiment varies from a few krad for outer detectors to about 220 krad for the inner layers. Each sub–detector is designed to withstand the expected ionizing radiation doses during ten years of operation.

- *Readout rate*: the ALICE experimental setup can be used with two different readout

[6]The impact parameter is defined as the vector connecting the centres of the colliding nuclei projected on the transverse plane to the nuclei momenta [2]. Such quantity is one of the most relevant variables in ALICE physics analyses.

configurations, operated simultaneously with two different triggers[7]. The *centrality trigger* activates the readout of the whole ALICE detector, including all ITS layers, while the *muon arm trigger* activates only the readout of a subset of fast readout detectors, including the two innermost layers of the ITS. Therefore, pixel detectors readout is set at less than 400 μs.

Table 1.2: Characteristics of the six ITS layers, the beam-pipe and the thermal shields. [13]

Layer	r (cm)	$\pm z$ (cm)	Number of modules	Active area per module $r\phi \times z$ (mm^2)	Intrinsic resolution (μm) $r\phi$	Intrinsic resolution (μm) z	Material budget X/X_0 (%)
Beam pipe	2.94	-	-	-	-		0.22
ITS layer 1	3.9	14.1	80	12.8×70.7	12	100	1.14
ITS layer 2	7.6	14.1	160				1.14
Th. shield	11.5	-	-	-	-		0.65
ITS layer 3	15.0	22.2	84	70.2×75.3	35	25	1.13
ITS layer 4	23.9	29.7	176				1.26
Th. shield	31.0	-	-	-	-		1.65
ITS layer 5	38.0	43.1	748	73×40	20	830	0.83
ITS layer 6	43.0	48.9	950				0.83

The main parameters of each layer, including the beam pipe and the thermal shields, are summarised in table 1.2.

1.2.2 ITS Upgrade

The present ITS precision in the determination of the track impact parameter is adequate to physics analyses on particles with $p_T > 2$x GeV/c, but for particles at low momenta the statistical significance of measurements is insufficient. For example, the charm baryon Λ_c has a mean proper decay length of 60 μm, that is lower than current ITS impact parameter resolution in the p_T range of the majority of its daughter particles. Therefore, charm baryons, as well as beauty mesons, beauty baryons and hadrons with multiple heavy quarks produced in central Pb–Pb collisions are currently not accessible with the running ALICE experiment configuration.

[7]In this context, the term trigger is used to refer to a set of hardware and software settings that gives the opportunity to record only those events that are useful for the physics analyses

A major limitation of the Run 2 ITS detector is the poor maximum readout rate of 1 kHz, irrespective of detector occupancy. This limitation due to the hardware limits ALICE to capture only a small fraction of the full Pb–Pb collision rate of 8 kHz delivered by the present LHC and would outrageously limit the use of 50 kHz Pb–Pb collision rate provided in Run 3.

Finally, another major limitation in the present ITS is the impossibility to access the detectors during maintenance and repair interventions. Rapid accessibility of the detector is a main requirement for the ITS Upgrade.

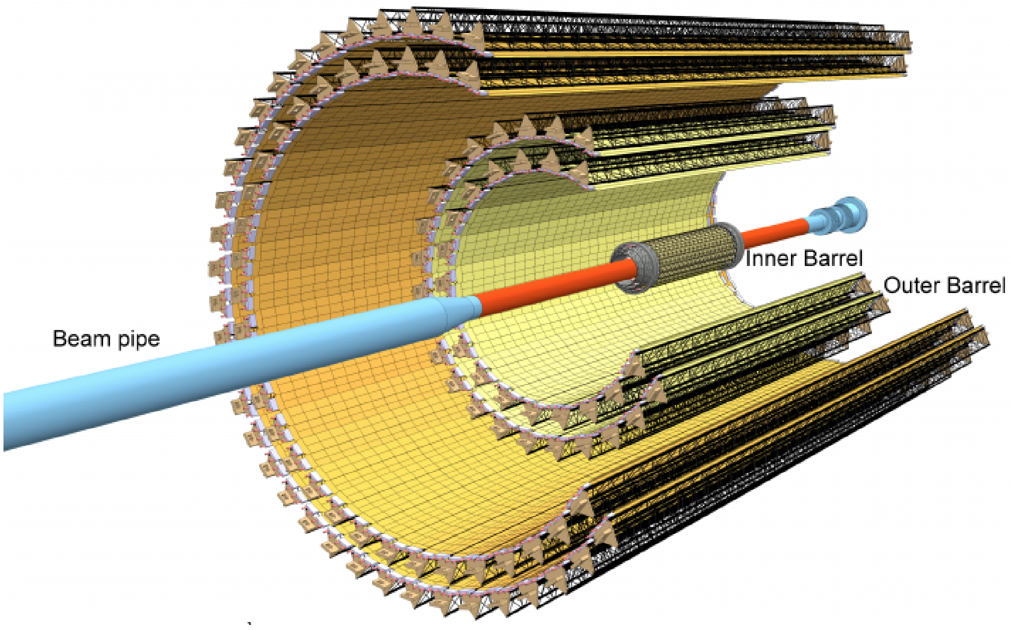


Figure 1.7: Layout of the upgraded ITS detector

The idea for the design of the ITS Upgrade is to entirely replace the existing ITS detector with a new one, composed of three inner layers of pixel detectors and four outer layers with either silicon strip detectors or pixel detectors with a lower granularity. Figure 1.7 shows the layout of the upgraded ITS detector. More specifically, according to [13], the following requirements need to be fulfilled to cope with the LHC interaction rate expected during Run 3:

- *Reduction of the distance between the ITS and the beam pipe*: the introduction of a new beam pipe with a smaller outer radius of 19.8 mm (with respect to the 30 mm radius of Run 2) allows the installation of an additional detector layer with a radius of about 22 mm.

- *Reduction of the material budget*: in order to improve the impact parameter resolution, it is particularly important to reduce the material budget for the first detector layer. Moreover, reducing the overall material budget will improve also the tracking performance and the momentum resolution. The use of Monolithic Active Pixel Sensors (MAPSs) will allow the material budget per layer to be reduced by a factor of 7 (50 μm instead of 350 μm). Furthermore, the optimisation of the analogue front-end timing specifications and readout architecture will reduce the power density by a factor of 2 and will increase the pixel density by a factor of 50. Finally, an improved electrical power and signals distribution scheme will reduce the material budget of electrical power and signal cables by a factor of 5.

- *Geometry and segmentation*: The upgraded ITS detector consists of seven concentric cylindrical layers covering a radial extension between 22 mm and 430 mm with respect to the beam line.
- *Energy loss measurement*: the new detector will preserve PID capabilities, but in the case where all 7 layers would be implemented with MAPS technology, the performance would be slightly reduced with respect to the present ITS
- *Readout time*: the upgraded ITS aims to read the data related to each individual interaction, up to a rate of 50 kHz for Pb–Pb collisions and 2 MHz for p–p collisions.

Table 1.3: Characteristics of the ITS upgrade scenario. The numbers in brackets refer to the case of microstrip detectors [13].

Layer	r (cm)	$\pm z$ (cm)	Intrinsic resolution (μm)		Material budget X/X_0 (%)
			$r\phi$	z	
Beam pipe	2.0	-	-		0.22
ITS layer 1	2.2	11.2	4	4	0.30
ITS layer 2	2.8	12.1			
ITS layer 3	3.6	13.4			
ITS layer 4	20.0	39.0	4 (20)	4 (830)	0.30 (0.83)
ITS layer 5	22.0	41.8			
ITS layer 6	41.0	71.2	4 (20)	4 (830)	0.30 (0.83)
ITS layer 7	43.0	74.3			

Main parameters of the new beam pipe and the upgraded ITS layers, are summarised in table 1.3. Values in brackets refer to the strip detectors design of the four outer layers, while the other values refer to MAPS technology detectors. Comparing these values to the ones in table 1.2, it appears that the radius of the outermost ITS remains unchanged, while the radius of the innermost layer is considerably lower in the upgraded detector.

The result of simulations indicates that an improved tracking efficiency and p_T resolution for the ITS standalone tracking can be achieved by grouping the seven layers in an innermost triplet, an intermediate pair and an outermost pair, as shown in figure 1.7. In particular, the track position resolution at the primary vertex is improved by a factor of 3 and the standalone tracking efficiency becomes comparable to what can be achieved combining the information of the ITS and TPC in Run 2.

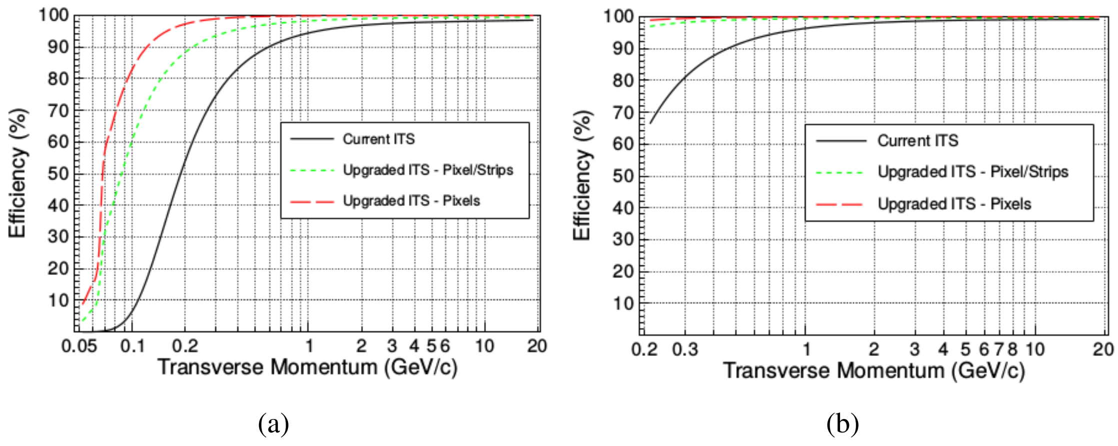


Figure 1.8: Tracking efficiency of charged pions for the current and upgraded ITS in the ITS stand-alone (left panel) and ITS-TPC combined (right panel) tracking modes [13]

Figure 1.8 compares the tracking efficiency for the two versions of the ITS detector, both for the standalone ITS tracking and for the ITS-TPC combined tracking. The upgraded ITS layout allows an impressive improvement to be obtained for $p_T < 1$ GeV/c, in particular if all 7 layers are equipped with MAPS detectors.

1.3 Online/Offline computing system

1.3.1 AliROOT framework

The ALICE Run 2 offline framework, called *AliROOT* [14], was developed to reconstruct and analyze data, to study different physics topics, coming from both simulations and real interactions. It was also used to perform simulations necessary to optimize the ALICE detectors design..

The AliROOT framework massively exploits ROOT [15] functionalities. ROOT is an object oriented (OO) framework for large-scale data handling applications, is written in C++ and offers advanced statistical analysis functions, advanced visualisation tools and the possibility to use C++ as a scripting language, as well as many other features. In particular, all the results shown this thesis work have been prepared by using the ROOT framework.

Users can interact with ROOT via a Graphical User Interface (GUI), the command line or batch scripts.

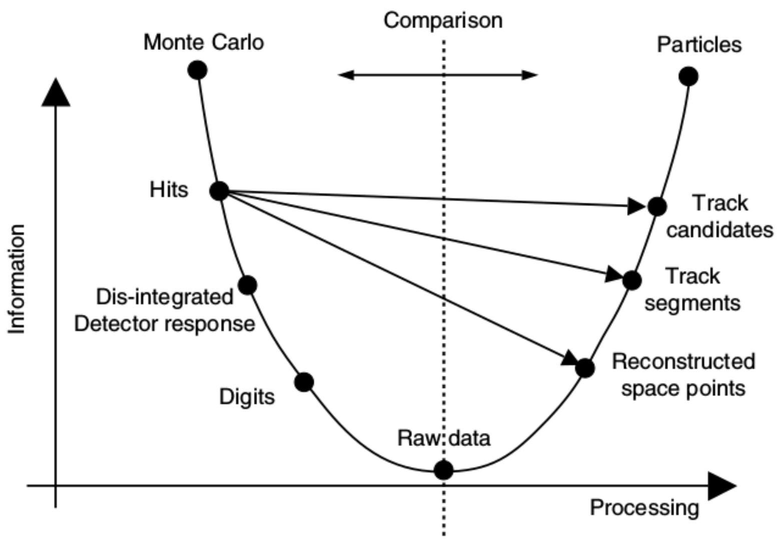


Figure 1.9: AliROOT data processing flow [7]

Figure 1.9 schematically shows the data processing flow in the AliROOT framework. Data are generated with simulation programs, namely Monte Carlo (MC) event generators combined with detector response simulation packages, with the full information about particles momentum and PID.

AliROOT relies on external MC tools to simulate heavy-ion collision events at the LHC energy, like Heavy-Ion Jet Interaction Generator (HIJING) [16,17], DPMJET version II.5 [18] and String-Fusion Model (SFM) [19], mediated by specific interfaces. AliROOT provides also tools to assemble events from different generators, creating the so called *event cocktails*, and to manage the particle correlation in a controlled way (*afterburners*).

The next step in the simulation chain is the detector response simulation, that is necessary to study in detail ALICE physics capabilities and to verify the functionality of the framework itself. AliROOT provides a Virtual Monte Carlo (VMC) interface [20], implemented via C++ virtual classes, in order to make the caller code independent of the real simulation process implementation, that is demanded to external tools like the GEANT3 [21], GEANT4 [22] or FLUKA [23] transport codes.

Finally the AliROOT framework provides a track reconstruction suite that can work with data coming from both the simulation chain and the real detectors Data Acquisition (DAQ) process. The algorithms used in this step are those described in section 1.1.4. The obtained tracks can be compared with the initial generated set, in order to evaluate correctness and performances of the adopted reconstruction algorithm.

1.3.2 The O^2 facility

The highly increased interaction rate that will affect ALICE in Run 3 will result in an estimated data throughput from the detector greater than 1 TB/s for Pb–Pb events. It is therefore necessary to achieve a maximal reduction of the readout, as early as possible during the data flow, in order to minimize the cost of the computing system for both data processing and storage.

The O^2 facility, the Online-Offline computing system that will assist the ALICE experiment during Run 3, has been designed to reach such a challenging goal. It will be a high-throughput, heterogeneous system, with nodes equipped with hardware acceleration and a software framework that will provide an abstraction layer to allow the same code to deliver its functionalities on different platforms, from laptops to the complete O^2 system itself.

Figure 1.10 shows the functional flow of the O^2 system. Data will be transferred from the detectors to the facility, via optical read-out links, in the form of several constant data streams. Dedicated time markers, synchronized with the LHC clock, will divide these streams into pieces called Time Frames (TFs). In particular, [24] distinguishes between Sub-Time Frames (STFs), that contain raw data from a single First Level Processor (FLP), and Compressed Time Frames (CTFs), that contain processed raw data of all the active detectors and that, once written, become read only data.

The optimal TF size is a trade-off between several criteria, involving the amount of data loss, synchronisation between O^2 components, calibration efficiency and data distribution. Any TF size between 20 ms and 100 ms is considered to be suited for calibration/reconstruction processes, while a finer TF granularity makes data buffering and distribution easier. [24] identifies a TF duration of 20 ms (a TF rate of 50 Hz) as the selected design value, with a TF size of 10 GB before compression and a 0.5% of data loss at frame boundaries.

Due to the local and independent nature of the involved data, a first stage of data processing, including local calibration and detector specific pattern recognition, will be performed with an high degree of parallelism and some of the raw data will be already replaced by the result of the processing. During this step, each FLPs collects the detector data at a rate of up to 3.2 GB/s from up to 48 read-out links, with a total rate of above 1.1 TB/s over approximately 8300 read-out links. Data are compressed by a factor of 2.5, merged, split into STFs and buffered, in order to be dispatched to Event Processing Nodes (EPNs) for aggregation.

A second, global step is carried out synchronously with the data taking, in order to assemble the data from all the detector inputs and to perform a global calibration. This step takes place in EPNs, where each cluster is assigned to a track with an additional reduction factor of 8 in the data volume. Results of this phase are then stored in the O^2 farm or parked in the Tier 0 if the farm capacity is exhausted. The total throughput to data storage reaches 90 GB/s (above 60 MB/s per EPN) after compression.

A final, asynchronous data processing step takes place before permanently store the

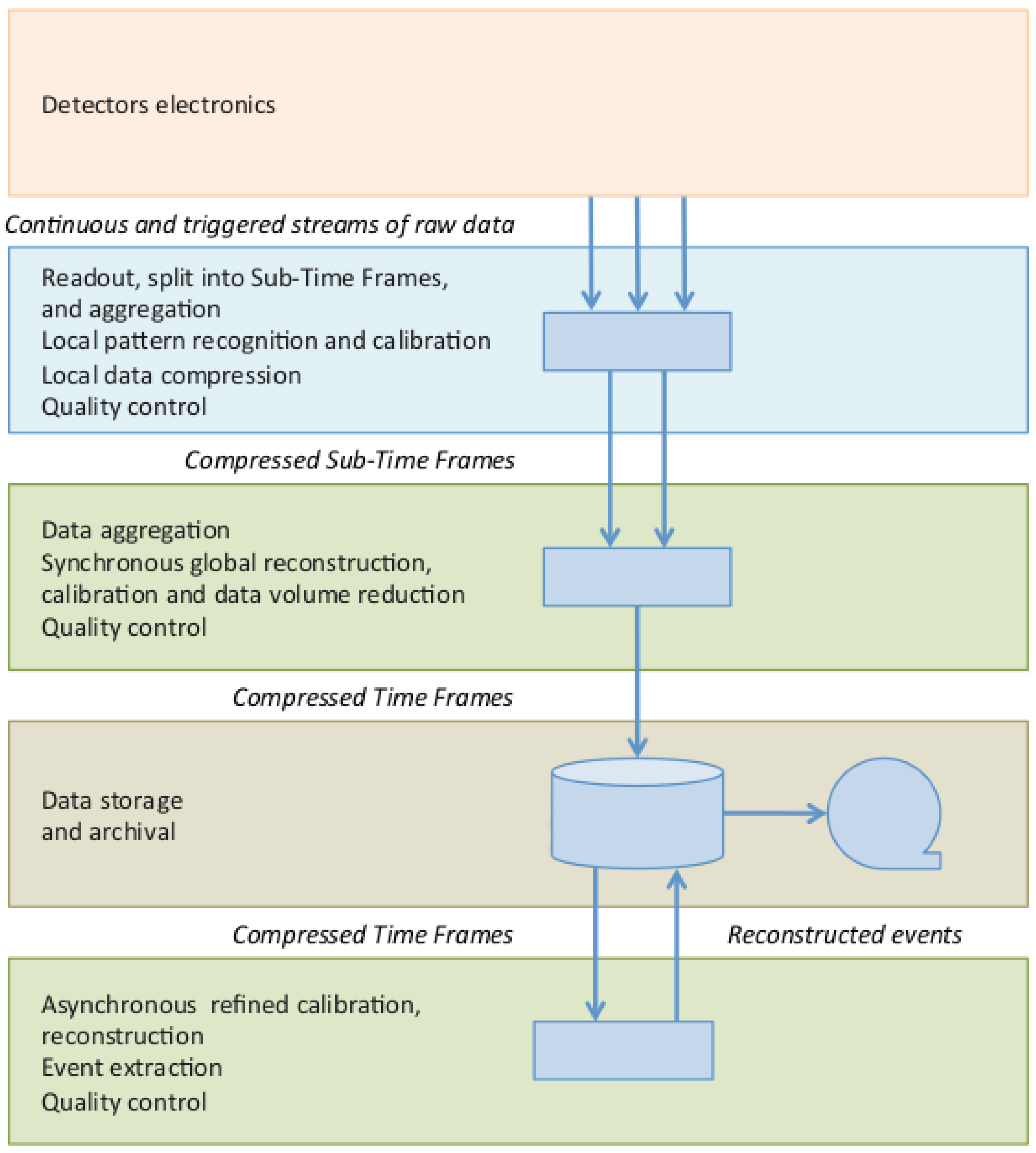

Detectors electronics
Continuous and triggered streams of raw data
Readout, split into Sub-Time Frames, and aggregation
Local pattern recognition and calibration
Local data compression
Quality control
Compressed Sub-Time Frames
Data aggregation
Synchronous global reconstruction, calibration and data volume reduction
Quality control
Compressed Time Frames
Data storage and archival
Compressed Time Frames
Reconstructed events
Asynchronous refined calibration, reconstruction
Event extraction
Quality control

Figure 1.10: O^2 facility data processing flow [24]

reconstructed events. This step will probably use computing resources from the Worlwide LHC Computing Grid (WLCG), in conjunction with the O^2 system, in order to successfully absorb the peak needs.

The main difference between O^2 and the current AliROOT framework is the presence of synchronous calibration and reconstruction phases, that are necessary to guarantee a

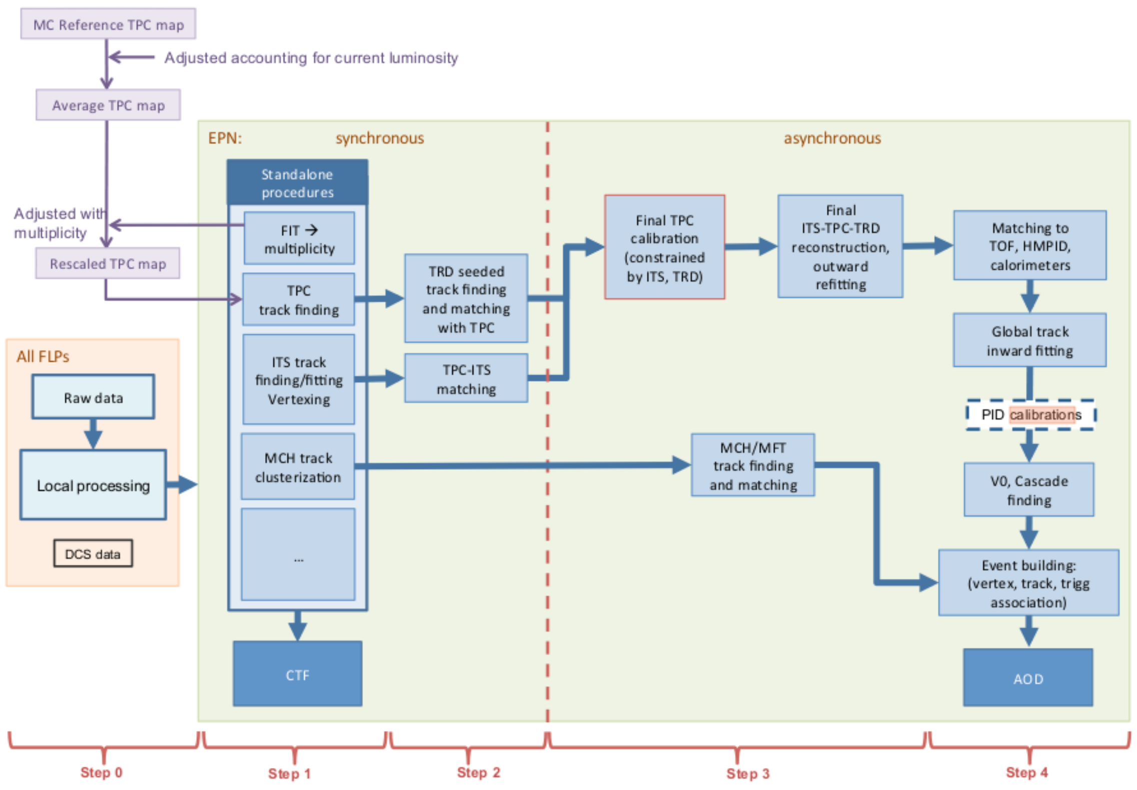

MC Reference TPC map
Adjusted accounting for current luminosity
Average TPC map
Adjusted with multiplicity
Rescaled TPC map
All FLPs
Raw data
Local processing
DCS data
EPN:
synchronous
asynchronous
Standalone procedures
FIT → multiplicity
TPC track finding
ITS track finding/fitting Vertexing
MCH track clusterization
...
CTF
TRD seeded track finding and matching with TPC
TPC-ITS matching
Final TPC calibration (constrained by ITS, TRD)
Final ITS-TPC-TRD reconstruction, outward refitting
Matching to TOF, HMPID, calorimeters
Global track inward fitting
PID calibrations
V0, Cascade finding
MCH/MFT track finding and matching
Event building: (vertex, track, trigg association)
AOD
Step 0
Step 1
Step 2
Step 3
Step 4

Figure 1.11: Schematic outline of the O^2 facility calibration and reconstruction data flow [24]

considerable reduction of permanently stored data. Figure 1.11 shows in detail the five steps of reconstruction and calibration data flows in the O^2 framework. In particular, it appears that standalone track-finding is carried out during Step 1 for both ITS and TPC detectors, on EPNs. Then, during Step 2, ITS-TPC matching is performed, as well as the TRD tracking using TPC tracks as seeds, while a final ITS-TPC-TRD matching takes place during Step 3.

Contrary to the strategy adopted in Run 2, where most of the tracks in the ITS are the result of a prolongation of the TPC tracks (as described in 1.1.4), in Run 3 the TPC detector will need the information about the ITS tracks in order to carry out the final calibration during Step 3. As things stand, at least a partial reconstruction of high p_T tracks must be done synchronously to provide constraints for TPC calibration, so the principal requirement for ITS tracking code is speed.

In order to meet such a goal, a new ITS tracking algorithm based on Cellular Automaton (CA) has been proposed in [2]. Theoretical and actual performances of such algorithm are analyzed in detail on chapter 2.

Chapter 2

Track Reconstruction Algorithm

The main goal of this thesis work is to investigate optimisation possibilities offered by a total or partial migration of the ITS track reconstruction algorithm developed in [2] to a GPU architecture. However, the first necessary step in that direction is a serial optimisation of the existing code and a careful analysis of the main bottlenecks in the process.

The ITS Upgrade reconstruction algorithm proposed in [2] can be divided into six main steps:

- an *indexing phase*, when information about the reconstructed hits coming from the previous clusterisation step is organised into an index table, in order to speed up subsequent data recovery and filtering operations

- a *tracklet finding phase*, when couple of clusters laying on subsequent layers that satisfy some filtering criteria are combined into tracklets

- a *cell finding phase*, when subsequent tracklets that satisfy some filtering criteria are merged into cells

- a *neighbourhood construction phase*, when cells are ranked in terms of the number of compatible inner cells

- a *track reconstruction phase*, when neighbour cells are combined into track candidates

- a *fitting phase*, when track candidates are fitted using a Kalman Filter

All of these steps are further described in section 2.1 and for each one a mathematical model is presented, in order to discuss both the memory occupancy and the computational complexity in the worst case scenario.

Section 2.2 describes the tracking algorithm implementation I realised during the first part of my thesis work. After an initial presentation of the main features and a list of the major differences with the implementation developed in [2], some of the adopted design choices and optimisation strategies are described in detail.

In section 2.3 an analysis of the actual performances of the implementation realised for this thesis work is reported. This is useful to identify the most complex algorithm phases and to build up a good GPU migration strategy.

2.1 Algorithm flow

2.1.1 Indexing phase

In this phase output data coming from the previous clusterisation step are organised into an efficient data structure, an $n_z \times n_\phi$,index table to be rapidly accessed during the next phase. A two-way sorting is necessary to compile each bin of size $\left(\phi_{bin}, z_{bin_i}\right)$ of the index table:

- the primary sorting criterion is the value of azimuthal angle ϕ, normalised to $[0, 2\pi]$ range
$$\phi_{bin} = \frac{2\pi}{n_\phi} \tag{2.1}$$
- the secondary sorting criterion is the value of z, that for each layer L_i must be contained in $[z_{min_i}, z_{max_i}]$ range, given by the physical dimension of the i-th layer of the ITS detector
$$z_{bin_i} = \frac{z_{max_i} - z_{min_i}}{n_z} \tag{2.2}$$

Both sorting criteria are connected to a double constrained domain, so for each layer $L_{[1,6]}$ (no index table is needed for layer L_0) an index table can be easily set up and stretched to cover those domains. Assuming that clusters come already sorted in ϕ and z from the previous algorithm step, for each of them it is necessary to find bin coordinates, that for a given cluster $c = \left(z_c, \phi_c\right)$ on layer L_i are expressed by the following relations:

$$\left\{ \left\lfloor \frac{z_c - z_{min_i}}{z_{bin_i}} \right\rfloor , \left\lfloor \frac{\phi_c}{\phi_{bin}} \right\rfloor \right\} \tag{2.3}$$

Quantities 2.1 and 2.2 can be computed at compile time, as they only depend on z extension of each layer (reported in table 1.3) and on index table size $n_z \times n_\phi$, that is also known at compile time. As a result, if N is the number of input clusters, the computational complexity of this phase is:

$$T(N) = O(N) \tag{2.4}$$

Assuming that clusters are stored in an associative, sorted data structure, an index table can be built by simply storing in each bin the unique key c_{key} of its first cluster. According to this, since only 6 index tables are needed (one for each level but L_0) and each table has $n_z \times n_\phi$ bins, memory occupancy for this phase is completely independent from the input size and can be expressed by the following relation:

$$S(N) = 6 \cdot n_z \cdot n_\phi \cdot \text{sizeof}\left(c_{key}\right) = O(1) \tag{2.5}$$

2.1.2 Tracklet finding phase

In the first step of the algorithm each layer pair is processed to find a link between each cluster of the first layer and all the compatible clusters on the second layer, namely all those clusters that lay on a 2-dimensional window, called region of interest, opened on the second layer of the pair, as shown in figure 2.1. To quickly build the filtering window for each cluster, the index table described in section 2.1.1 is used.

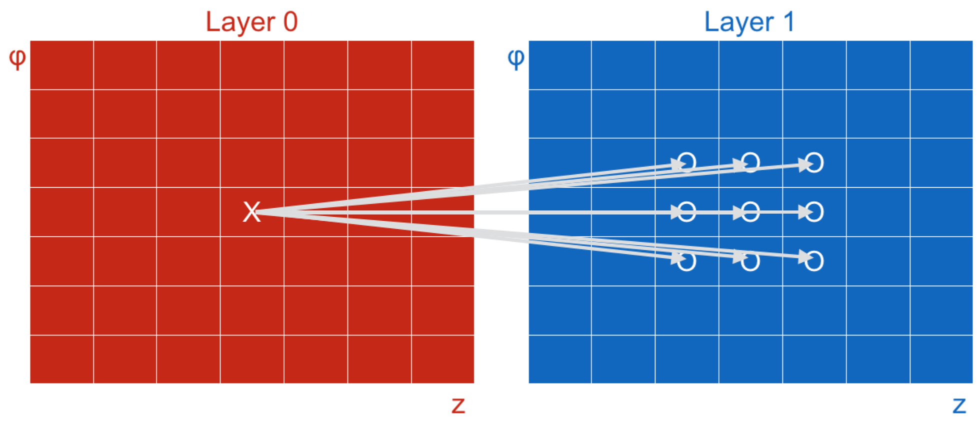


Figure 2.1: Example of the index tables for the first couple of layers, courtesy of [2]. Clusters are sorted according to ϕ and z coordinates and consequently it is possible to efficiently find, for each cluster in layer 0, all clusters contained in its region of interest on layer 1.

Considering a cluster $c = \left(z_c, \phi_c, r_c\right)$ on layer L_i, dimensions of its region of interest are computed with respect to ϕ_c and to the z coordinate of the intersection $\left(z_{L_{i+1}}, r_{L_{i+1}}\right)$ between layer L_{i+1} and the line passing through the cluster and the interaction vertex V of the colliding beams:

$$z_{L_{i+1}} = \left(\frac{z_c - z_V}{r_c}\right) \cdot \left(r_{L_i+1} - r_c\right) + z_c \tag{2.6}$$

For each possible cluster pair within the region of interest, only those couples $\left(c_1, c_2\right)$ that satisfy all the following filtering criteria can form a valid tracklet:

- the difference between azimuthal angles of the two cluster must be smaller than a threshold $\Delta\phi_{MAX}$ whose value is equal for all layer pairs

$$\left|\phi_{c_1} - \phi_{c_2}\right| < \Delta\phi_{MAX} \tag{2.7}$$

- the DCA_z, the distance of closest approach along the z axis, to the interaction vertex V, of the prolongation of the tracklet must be smaller than a threshold ΔDCA_z^{MAX}, whose value is layer dependent

$$\frac{z_{c_1} - z_V}{r_{c_1}} \cdot \left(r_{c_2} - r_{c_1}\right) - \left(z_{c_2} - z_{c_1}\right) < \Delta DCA_z^{MAX} \tag{2.8}$$

If a tracklet is not filtered out by the previous cuts, it is stored together with two quantities related to its direction:

- the segment inclination in the transverse plane, indicated by ϕ_T

$$\phi_T = \text{atan2}\left(y_{c_2} - y_{c_1}, x_{c_2} - x_{c_1}\right) \tag{2.9}$$

- the inclination of pseudo-plane rz, indicated by $\tan\lambda_T$

$$\tan\lambda_T = \frac{z_{c_2} - z_{c_1}}{r_{c_2} - r_{c_1}} \tag{2.10}$$

Considering a single layer pair, the first step of this phase is the bin selection for the current cluster, namely the construction of its region of interest. This operation consists in a simple iteration over a subset of the current index table bins. The worst case is when all bins must be kept: this leads to a computational complexity of

$$T(N) = \sum_{i=1}^{6} n_z \cdot n_\phi = O(1) \tag{2.11}$$

The second step is the tracklet filtering. The complexity of this step highly depends on the number of clusters in the filtering window, namely on the choice of an appropriate window size for each cluster of the first layer. The number of clusters in each layer can be represented by the following relation:

$$N_i = \alpha_i \cdot N, \qquad \sum_{i=0}^{6} \alpha_i = 1 \tag{2.12}$$

For each layer pair, all clusters on the first layer and a subset of clusters on the second layer must be taken into account. This subset can be quantified as the whole set of clusters on the second layer reduced by a filtering factor $\beta_{T_{i+1}} \geq 1$:

$$T(N) = \sum_{i=0}^{5} N_i \cdot \frac{N_{i+1}}{\beta_{T_{i+1}}} = \sum_{i=0}^{5} \frac{\alpha_i \cdot \alpha_{i+1}}{\beta_{T_{i+1}}} \cdot N^2 = O\left(N^2\right) \tag{2.13}$$

The same applies for the tracklets memory occupancy, with the addition of a filtering factor $\gamma_{T_i} \geq 1$ to take into account the effect of cuts expressed by equations 2.7 and 2.8:

$$S(N) = \sum_{i=0}^{5} \tau_i \cdot \text{sizeof}(tracklet) \cdot N^2 = O\left(N^2\right) \quad \text{where} \quad \tau_i = \frac{\alpha_i \cdot \alpha_{i+1}}{\beta_{T_{i+1}} \cdot \gamma_{T_i}} \tag{2.14}$$

2.1.3 Cell finding phase

In this step of the algorithm tracklets spanning on three consecutive layers and with the middle cluster in common are considered. Two consecutive tracklets are combined into cells if they have compatible directions, otherwise they are discarded.

Neglecting the effects of the multiple scattering, the three clusters of the cell should lay on a circle because of the presence of the ALICE magnetic field. Circle finding in a 2-dimensional space is computationally complex, so clusters are mapped on a paraboloid in a 3-dimensional space, with the minimum point laying on the interaction vertex and an axial symmetry along the w direction, using the following parametrisation:

$$S = \left\{x, y, w = r^2\right\} \tag{2.15}$$

In this new space, the equation of the circle with centre $\left\{x_c, y_c, z_c\right\}$ and radius ρ is similar to the equation of a plane

$$w - 2xx_c - 2yy_c + w_c - \rho^2 = 0 \tag{2.16}$$

A plane in space can be defined as the product of two different quantities:

- the unit vector $\vec{n}$ normal to the plane, that can be defined as the external product between vectors $\left\{\vec{s}_1, \vec{s}_2, \vec{s}_3\right\}$ connecting the clusters

$$\vec{n} = \left\{n_0, n_1, n_2\right\} = \frac{\left(\vec{s}_1 - \vec{s}_0\right) \wedge \left(\vec{s}_2 - \vec{s}_0\right)}{\left\|\left(\vec{s}_1 - \vec{s}_0\right) \wedge \left(\vec{s}_2 - \vec{s}_0\right)\right\|} \tag{2.17}$$

- the distance c of the plane from the origin, that can be defined as the projection of any one of the connecting vectors $\vec{s}_i$ on $\vec{n}$

$$c = -\vec{n} \cdot \vec{s}_i \tag{2.18}$$

By comparing the plane expression with 2.16, the following expressions are obtained for the circle center and radius, respectively:

$$\left\{x_c, y_c\right\} = -\frac{1}{2} \cdot \left\{\frac{n_0}{n_2}, \frac{n_1}{n_2}\right\} \qquad \text{and} \qquad \rho = \sqrt{\frac{1 - n_2^2 - 4 \cdot c \cdot n_2}{4 \cdot n_2^2}} \tag{2.19}$$

In this step of the track reconstruction phase, for each tracklet pair three selection criteria are applied to the resulting cell $\left\{c_i, c_{i+1}, c_{i+2}\right\}$:

- $\Delta \tan \lambda_T$ and $\Delta\phi_T$ must not exceed the corresponding threshold values
- value of DCA_z of the cell must not exceed a DCA_z^{MAX} threshold value, where DCA_z has the following expression:

$$DCA_z = \left|\frac{\tan \lambda_{T_1} + \tan \lambda_{T_2}}{2} \cdot r_{c_i} + \left(z_V - z_{c_i}\right)\right| \tag{2.20}$$

- value of DCA_{xy}, the projection of the distance of closest approach to the interaction vertex V on the xy plane, must not exceed a DCA_{xy}^{MAX} threshold value, where DCA_{xy} value has the following expression:

$$DCA_{xy} = \left|\rho - \sqrt{x_c^2 + y_c^2}\right| \tag{2.21}$$

Cell finding phase tries to combine only those tracklets that share a cluster. Considering the layer triplet $\left\{L_i, L_{i+1}, L_{i+2}\right\}$ and a single cluster c^* on the layer L_{i+1}, there are $\lambda_{L_i \to c^*} \leq \alpha_i \cdot N$ tracklets of type $\left\{x_i, c^*\right\}$ and $\lambda_{c^* \to L_{i+2}} \leq \alpha_{i+2} \cdot N$ tracklets of type $\left\{c^*, x_{i+2}\right\}$.

Applying this logic to all $\alpha_{i+1} \cdot N$ clusters in layer L_{i+1} and extending it to all the layer triplets, the following relation is obtained:

$$T(N) = \sum_{i=0}^{4} \sum_{c=1}^{\alpha_{i+1}\cdot N} \lambda_{L_i \to c} \cdot \lambda_{c \to L_{i+2}} \leq \sum_{i=0}^{4} \left(\alpha_{i+1} \cdot \alpha_i \cdot \alpha_{i+2}\right) \cdot N^3 = O\left(N^3\right) \tag{2.22}$$

Despite the theoretical cubic time complexity of the worst case, if the filtering operation in the tracklet finding step is efficient enough, only a small subset of tracklets $\lambda_{L_i \to c^*} \ll \alpha_i \cdot N$ can be retained, with a great performance improvement in this step.

The same goes for the cells memory occupancy, with the addition of a filtering factor $\gamma_{C_i} \geq 1$ to take into account the cuts described above:

$$S(N) = \sum_{i=0}^{4} \upsilon_i \cdot \text{sizeof}(cell) = O\left(N^3\right) \quad \text{where} \quad \upsilon_i = \frac{\sum_{c=1}^{\alpha_{i+1}\cdot N} \lambda_{L_i \to c} \cdot \lambda_{c \to L_{i+2}}}{\gamma_{C_i}} \tag{2.23}$$

In order to compose tracklets into cells, it is necessary to find, for each tracklet in the first layer pair, all tracklets in the second layer pair that start from its second cluster. Because of the iterative implementation of the tracklet finding step, if tracklets are stored into a data structure that preserves the insertion order, all tracklets that start from the same clusters are contiguous. As a result, the aforementioned search phase can be optimised by saving into an index table, for each cluster on a layer but L_0, the index of the first tracklet that starts from it. This operation reduces the compatible tracklets search operation to a single memory access, but leads to an additional memory occupancy of

$$S(N) = \sum_{i=1}^{5} \alpha_i \cdot N \cdot \text{sizeof}(index) = O(N) \tag{2.24}$$

2.1.4 Neighbourhood construction phase

In this step of the reconstruction algorithm a rank is assigned to each cell. If two cells spanning on 4 contiguous layers share a tracklet of the pair and satisfy some filtering criteria based on normal vectors $\vec{n}$ and the radii ρ, they are considered to be neighbours. An index equal to 1 is given to all the cells without any neighbours, while all the cells with one or more neighbours acquire an index equal to the maximum index among the neighbours plus one.

To find all the possible neighbours for a cell spanning a layer triplet $\left\{L_i, L_{i+1}, L_{i+2}\right\}$, it is necessary to iterate over all cells on layer triplet $\left\{L_{i+1}, L_{i+2}, L_{i+3}\right\}$ that share the tracklet laying on the layer pair $\left\{L_{i+1}, L_{i+2}\right\}$. Considering a tracklet $\left\{c_1, c_2\right\}$ on this layer pair, there are $\lambda_{L_i \to c_1} \leq \alpha_i \cdot N$ cells starting on layer L_i and $\lambda_{c_2 \to L_{i+3}} \leq \alpha_{i+3} \cdot N$ starting on layer L_{i+1} and containing the tracklet. By combining those values with equations 2.14 and 2.22, the following relation is obtained:

$$T(N) = \sum_{i=0}^{3} \sum_{t=1}^{\tau_i \cdot N^2} \lambda_{L_i \to c_1} \cdot \lambda_{c_2 \to L_{i+2}} = O\left(N^4\right) \tag{2.25}$$

If cells are stored into a dictionary[1], the neighbourhood relation between them can be represented with a list of cells identified by their unique key c_{key}.

By introducing an additional filter factor $\gamma_{N_i} \geq 1$, the memory occupancy for this phase

[1]An abstract data type storing items, or values. A value is accessed by an associated key. Basic operations are new, insert, find and delete. A dictionary defines a binary relation that maps keys to values. The keys of a dictionary are a *set*, namely an unordered collection of values where each value occurs at most once [25].

is represented by the following relation:

$$S(N) = \sum_{i=0}^{3} \frac{\sum_{t=1}^{\tau_i \cdot N^2} \lambda_{L_i \to c_1} \cdot \lambda_{c_2 \to L_{i+2}}}{\gamma_{N_i}} \cdot \text{sizeof}\left(c_{key}\right) = O\left(N^4\right) \tag{2.26}$$

2.1.5 Track reconstruction phase

During the tracks reconstruction phase all neighbour cells are combined each other recursively to construct complete roads, starting from cells spanning the outermost ITS layer triplet. Since each cell $c^*_{L_i}$ on layer L_i has $v_{c^*_{L_i}} \leq \alpha_{L_{i-1}} \cdot N$ neighbours, if T^* is the complexity to process that single cell, computational complexity of recursive algorithm starting from cell $c^*_{L_i}$ is

$$T'\left(c^*_{L_i}\right) = \begin{cases} T^*, & \text{if } i = 0 \\ T^* + \sum_{n=0}^{v_i} T'\left(c^n_{L_{i-1}}\right) & \text{otherwise} \end{cases} = O\left(N^i\right) \tag{2.27}$$

For tracks that span all seven ITS layers (namely those with a given index of 5), since from equation 2.23 there are $\lambda_5 \leq \upsilon_4$ reconstructed cells on the outermost layers, computational complexity of this phase is

$$T(N) = \sum_{c=0}^{\upsilon_4} T'\left(c_{L_4}\right) = O\left(N^7\right) \tag{2.28}$$

Since no other selection criteria are applied in this phase, memory occupancy is directly derived from computational complexity, because all processed roads are also stored in the memory:

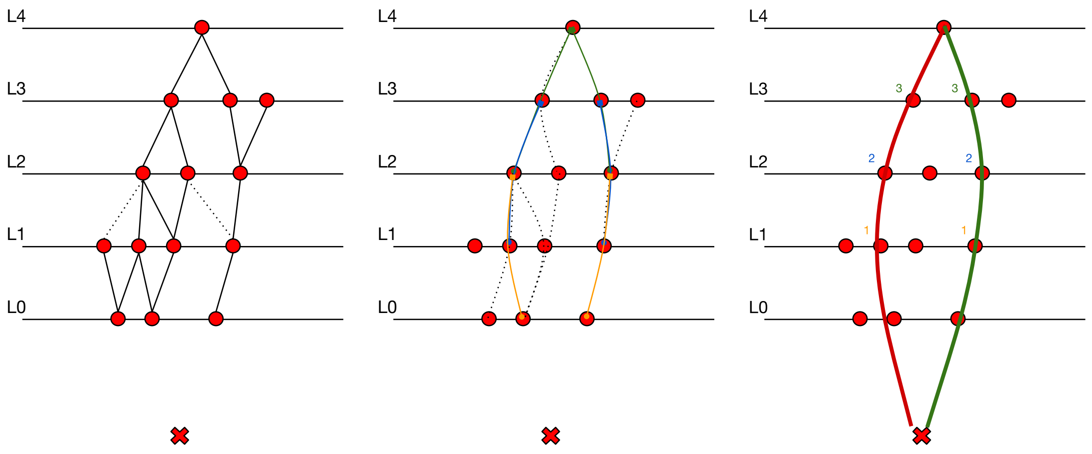
$$S(N) = O\left(N^7\right) \tag{2.29}$$

To speed up the search of compatible cells, tracklets are stored in an index table with the same logic described in the previous section. As a result, compatible cells search operation is reduced to a single memory access and memory occupancy is linear with respect to the number of tracklets:

$$S(N) = \sum_{i=1}^{4} \tau_i \cdot N^2 \cdot \text{sizeof}(index) = O\left(N^2\right) \tag{2.30}$$

2.1.6 Fitting phase

Complete roads reconstructed during the previous algorithm phase represent the track candidates. If two or more candidates share at least one cluster, a Kalman Filter fit is applied to them and only the one with the minimum χ^2 is kept. Since the implementation of the agorithm realised for the current thesis work does not implement the fitting phase, it will not be described any further.



(a) (b) (c)

Figure 2.2: Graphical representation of the reconstruction steps on the xy transverse plane of the Cellular Automata algorithm, courtesy of [2]. In particular, the tracklet finding phase (left), the cell finding phase (middle) and the track reconstruction phase (right) are represented. The red cross represent the reconstructed position of the interaction vertex, while the red dots are the reconstructed hits (clusters) on the different ITS layers.

Figure 2.2 graphically summarises all the track reconstruction phases described above for the first 5 layers of ITS detector. In particular, in figure 2.2c the fit is applied to both track candidates that share the outermost cluster, in order to retain only one of them (in this case the right one).

2.2 Serial CPU implementation

2.2.1 Main features overview

Starting from the implementation of the algorithm developed for [2] and integrated in the AliROOT framework, an optimised serial version has been realised for this thesis work. In this way, all those constraints explicitly required by the AliROOT context could be removed, so that the optimisation strategy has to be planned by taking into account only the minimum set of functionalities that must be provided by the reconstruction algorithm itself.

In order to distinguish between the two implementations, the version of the algorithm integrated in AliROOT will be hereinafter referred to as the *AliROOT* version, while the implementation realised for this thesis work will be referred to as the *standalone* version of the algorithm.

As a first, obvious step in the code migration process from the AliROOT version to the standalone one, all AliROOT specific dependencies were removed, including ROOT primitive type redefinitions, math libraries and data structures. In addition, whatever external library inclusion has been avoided, in favour of a pure C++14 Standard Template Library (STL) implementation. Such a design choice makes this code a good starting point for further optimisation and integration analyses, despite its minimal set of functionalities.

With respect to the AliROOT implementation, the standalone one lacks of some features that were considered irrelevant to the GPU migration process. Therefore, the last Kalman filter fitting phase of the algorithm (see section 2.1.6) hasn't been implemented, because it doesn't represent a bottleneck for the process and can thus be performed on CPU.

Moreover, in order to increase the tracking efficiency at low momenta, in the AliROOT implementation the whole algorithm flow described in section 2.1 is repeated a second time, with relaxed filtering criteria and by taking into account also shorter track candidates, spanning yet unused clusters, during the track reconstruction phase. Since both iterations share the same logical process, the double step refinement is reduntant for GPU migration analysis purposes, because the optimisation criteria used in a single step version of the algorithm are perfectly applicable also to a multi-step version. The same goes for the shorter track candidates inclusion.

The standalone version offers the possibility to manage events with multiple interaction vertices coming from the previous vertexing step. This feature, that was missing in the AliROOT implementation, is mandatory in order to handle pile-up events, that due to the increased readout rate of the ITS Upgrade will be sizeable. Additionally, the pile-up management capability introduces the possibility to study the algorithm behaviour with an increasing amount of input clusters per interaction vertex and to analyze the trend of both computation time and memory occupancy with respect to the input size.

2.2.2 Software architecture

The full source code written for this thesis work is available at [26]. In order to lay the groundwork for possible future extensions and to simplify a potential integration with the ALICE O^2 framework, the O^2 coding guidelines have been adopted [27].

The general classes architecture in the standalone implementation broadly follows the AliROOT implementation structure. Input data are read from a text file that, for the sake of simplicity, has been preferred to a more efficient binary one, and saved into a `CAEvent` object. All the necessary I/O utilities are included into the `CAIOUtils` namespace, in order to simplify the potential inclusion of additional suported file formats.

A `CATracker` object implements all the algorithm flow logic, in order to produce a list of track candidates from a `CAEvent` object when its `clustersToTracks` method is called. In the case of multiple interaction vertices associated to a single event, a list of `CARoad` objects is returned for each vertex. Therefore, the output has the form of `std::vector<std::vector<CARoads>>`.

Since all the program complexity is incapsulated into the `CATracker` class implementation, the usage of the program functionalities is really straightforward: in order to process a text file containing a certain number of events, the following code is sufficient.

```
std::vector<CAEvent> events =
  CAIOUtils::loadEventData(eventsFileName);

CATracker tracker{};

for (int iEvent = 0; iEvent < eventsNum; ++iEvent) {

  CAEvent& currentEvent = events[iEvent];
  std::vector<std::vector<CARoad>> roads =
    tracker.clustersToTracks(currentEvent);

  doSomethingWithRoads(roads);
}

```

The aforementioned `clustersToTracks` method internally calls a sequence of protected methods, each one implementing a single phase of the tracking algorithm. Input data and intermediate results are propagated from each phase to the following by a specific struct called `CAPrimaryVertexContext`. Since `CAPrimaryVertexContext` is a data member of `CATracker` class, the same object can be used to process a sequence of events by simply resetting the internal data structures, without the need to reallocate the entire memory if the currently allocate size is enough to store all required data. This comes with an important performance speedup, because memory allocation operations are quite costly in a standard CPU architeture and are really expensive in a GPU environment.

In addition to the basic functionalities described above, the standalone implementation realised for this thesis work contains several features to benchmark various aspects of the code, discussed in section 2.3. In particular, the `main` function is able to generate some text files that can be sent to a series of ROOT scripts, each of which can generate a report on a specific aspect of the code. Since ROOT offers the possibility to use C++ as a scripting language, these procedures can easily be launched from command line or directly into an Integrated Development Environment (IDE), without any need to compile them.

As pointed out in section 1.3.2, code portability is an important requirement for the O^2 framework. The fact that the standalone implementation only relies on C++14 STL utilities is an important step forward in that sense. Additionally, the CMake open-source tool [28] has been adopted in order to manage the code compilation process in a compiler independent way. More specifically, CMake version 3.0.2 or later is required to compile the program code, together with a C++14 compliant compiler.

2.2.3 Code optimisation

With respect to AliROOT implementation, some optimisations have been made to the tracking code. A noteworthy update is the usage of the `constexpr` specifier whenever possible, in order to enable the C++ compiler to evaluate the associated expression at compile time. In particular, all program constants, included in the `CAConstants` namespace, are declared as `constexpr`. This allows the system to optimise such values during the program execution, for example by placing it in read-only memory [29].

Another C++11 functionality largely used in the standalone implementation is the `emplace_back` method of the `std::vector` class, that allows the new object to be constructed in-place at the end of the container. In principle, with respect to the `push_back` method, that takes the object to be inserted and then copies (or moves) it into the caller memory, the `emplace_back` functionality should run faster, or at least it should never be less efficient. In practice, as discussed in [29], there are some cases when insertion functions run faster, but an emplace functionality almost certainly outperforms its insertion counterpart when all the following conditions are satisfied:

- the value being added to the container is constructed, not assigned;
- the argument type being passed differs from the type held by the container.
- The container doesn't reject the new value as a duplicate. This is always true for `std::vector`, because it does not require stored elements to be unique.

Another important speedup has been obtained by simply taking as much computation as possible out of loops, in order to reduce the number of repeated operation. Indeed, due to the heavily iterative nature of the tracking process, the code contains several loops, each one with an high number of iterations. In such a scenario, even the simplest operation, if taken out of a loop, can cause a noticeable performance improvement.

Finally, an important role in terms of performance optimisation is played by the memory preallocation of the data structures, described in detail in section 2.3.2. Memory resize of a `std::vector` container is an expensive process, because it requires to allocate a block of memory that is multiple of the container's current capacity (in most implementations, container capacity grows by a factor of two each time), copy all the elements from the old memory into the new one, destroy the objects in the old memory and deallocate the old memory [30]. Indeed, a proper initial memory allocation of the container, obtained by calling the `reserve` method of `std::vector`, leads to a considerable speedup of the program.

2.3 Performance analysis

2.3.1 Track reconstruction efficiency over transverse momentum

The track reconstruction efficiency over transverse momentum p_T has been analyzed in order to compare performances of the standalone implementation of the algorithm with the previous AliROOT version.

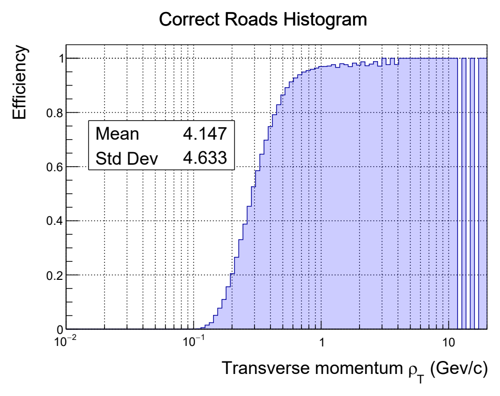
In order to be consistent with analyses reported in [2], the same sample of MC generated events has been used as input data and the same reconstructed tracks labelling system has been adopted. In order to classify the reconstructed track candidates, an equal numeric label has been assigned to all clusters that belong to the same MC simulated track; based on the criteria used to label the clusters of a track, each reconstructed track is assigned to one of the following subsets:

- a *correct* tracks subset, that contains all those tracks which clusters share the same label. In other words, a reconstructed track is considered to be correct only if there is a perfect correspondence with a MC simulated track;
- a *duplicated* tracks subset, that contains all those tracks that could be considered correct, but their label has yet been associated to another track. For each input label, only one correct track can be reconstructed;
- a *fake* tracks subset, that contains all those tracks that have at least one cluster with a different label.

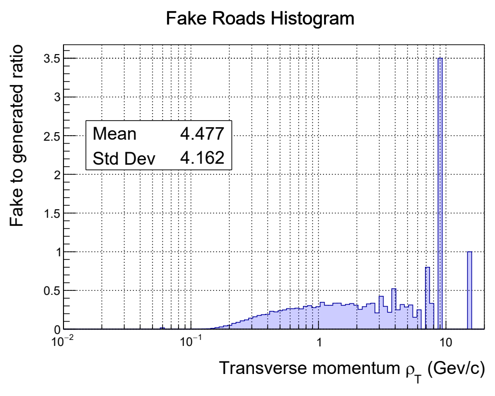
Finally, as in [2], only charged pions with at least one cluster per ITS layer are considered in this analysis.

Figure 2.3 shows the results obtained for the analysis of the tracking efficiency as a function of the transverse momentum for pion-related reconstructed tracks with 7 clusters. Looking at the histogram 2.3a, it appears that the tracking efficiency saturates at $p_T \approx 4 GeV/c$ and it rapidly drops for $p_T \leq 0.7 GeV/c$. Such a waveform is comparable to the tracking efficiency obtained for the AliROOT version of the algorithm, when a configuration with only one iteration and tight cuts was used: this constitutes a proof of correctness for the new implementation. Histogram 2.3b shows the amount of fake tracks reconstructed, with respect to the MC generated ones. This behaviour is actually very different from the fake tracks analysis presented in [2], because of the lack of a final fitting phase that would have removed a significant part of them; the same goes for duplicated tracks trend, reported in 2.3c. The almost flat waveform in histogram 2.3d shows instead that the relative amount of duplicated tracks reconstructed by the algorithm does not depend on the p_T value.

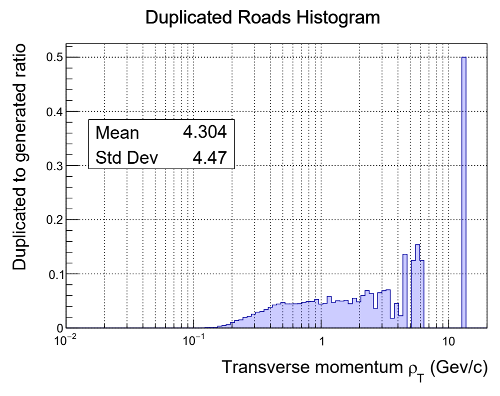
In addition to the comparative analysis described before, it is interesting to study the algorithm efficiency behaviour when multiple interaction vertices are reconstructed. Figure 2.4 shows four plots similar to those of figure 2.3, but in this last figure the histograms



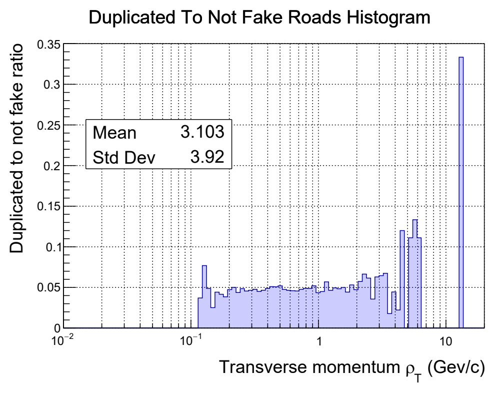
(a)



(b)



(c)



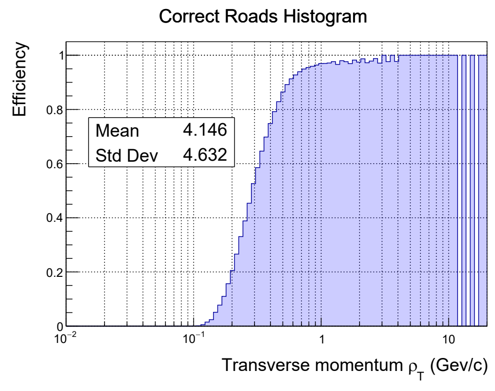
(d)

Figure 2.3: Analysis of the algorithm track reconstruction efficiency over transverse momentum for a sample of 100 central Pb–Pb events without pile-up, separately for correct (top left), fake (top right) and duplicated (bottom left) tracks subsets. The bottom right histogram shows the ratio of duplicated to not fake (correct + duplicated) tracks

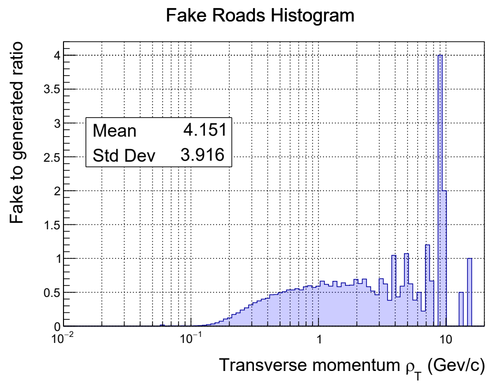
refers to events with four simulated interaction vertices. The probability to have such a pile-up in Pb–Pb central events is really low, but this analysis is nevertheless useful to investigate the algorithm behaviour in the worst case scenario. By comparing the two sets of histograms, it appears that tracking efficiencies are quite identical, while the amount of duplicated and fake tracks grows up considerably with the number of input vertices.

2.3.2 Memory occupancy benchmark

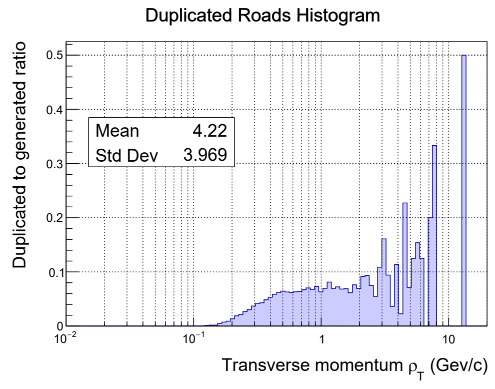
In order evaluate the memory occupancy theoretical model for tracklet and cell finding phases, analysed respectively in sections 2.1.2 and 2.1.3, actual memory occupancy reports have been generated for both steps. From equation 2.14 it follows that the actual number of reconstructed tracklets N_{T_i} spanning the layer couple $\{L_i, L_{i+1}\}$ should be equal to $\tau_i N^2$.



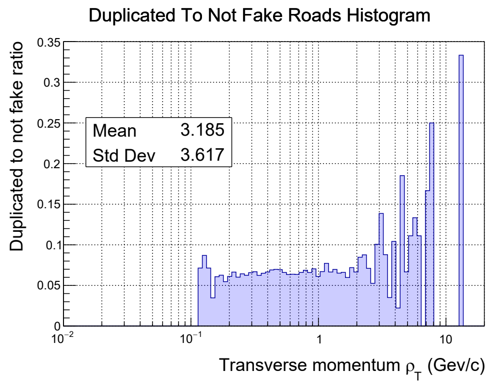
(a)



(b)



(c)



(d)

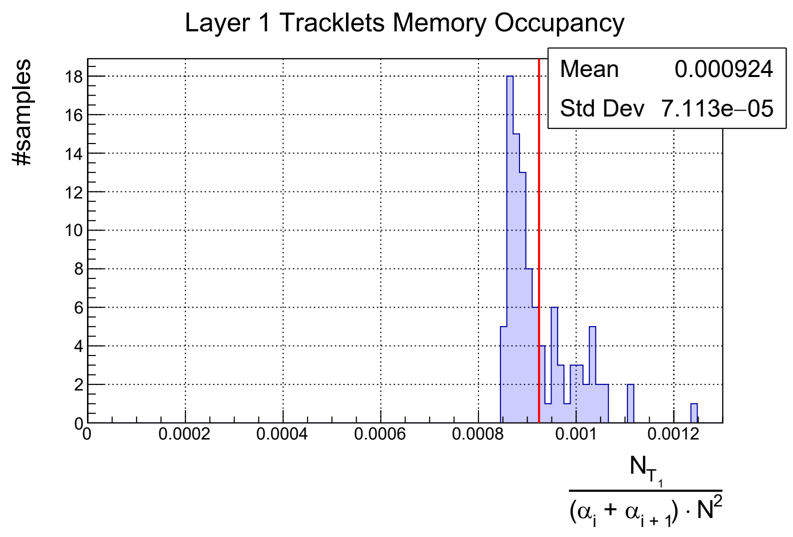
Figure 2.4: Analysis of the algorithm track reconstruction efficiency over transverse momentum for a sample of 100 central Pb–Pb events without four interaction vertices in input, separately for correct (top left), fake (top right) and duplicated (bottom left) tracks subsets. The bottom right histogram shows the ratio of duplicated to not fake (correct + duplicated) tracks.

Coefficient τ_i depends on the number of clusters on the two layers, that is independent from the algorithm implementation, and on the filtering factor

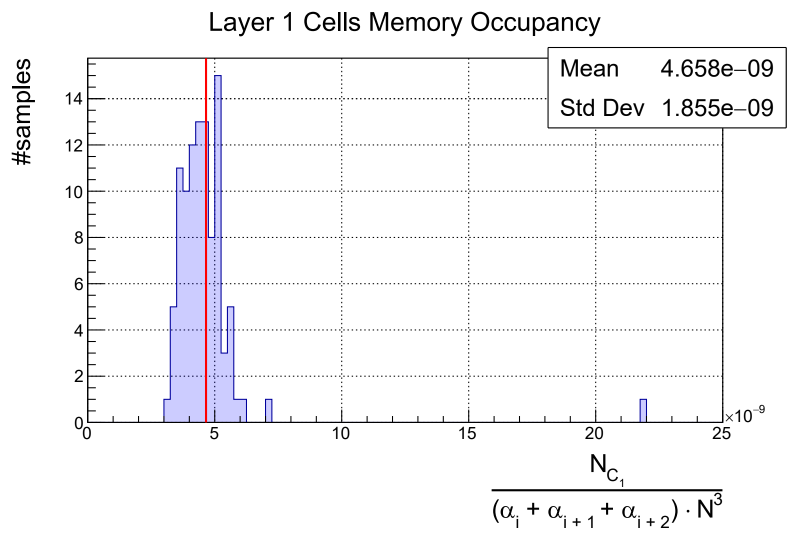
$$\beta_{T_{i+1}}\gamma_{T_i} = \frac{N_{T_i}}{\left(\alpha_i + \alpha_{i+1}\right) \cdot N^2} \tag{2.31}$$

that is instead dependent to the effectiveness of cuts, but it does not depend from the input size. The quantity expressed in eq. 2.31 and in particular its Root Mean Square (RMS) represents therefore a good indicator to evaluate the quality of the model. The same goes for the cell finding phase, but this time the filtering factor is represented by the following relation, where N_{C_i} is the number of cells spanning the layer triplet $\{L_i, L_{i+1}, L_{i+2}\}$:

$$\frac{N_{C_i}}{\left(\alpha_i + \alpha_{i+1} + \alpha_{i+2}\right) \cdot N^3} \tag{2.32}$$



(a)



(b)

Figure 2.5: Distribution of tracklet (top) and cell (bottom) filtering factors in a sample of 100 central Pb–Pb events without pile-up, reconstructed with the CA algorithm. The red vertical line indicates the mean value of the distribution.

Figure 2.5 shows the distribution of the values given by eqs. 2.31 and 2.32 for tracklets and cells starting from the layer L_1. Looking at the two histograms, it appears that cell distribution is much more sparse than the other one, with a single outlier that is really far from the mean value. This assertion is fully confirmed by the comparison of the Coefficients of Variation (CVs) of the two distributions:

$$c_{vT} \approx 7.7\%, \quad c_{vC} \approx 39.8\% \tag{2.33}$$

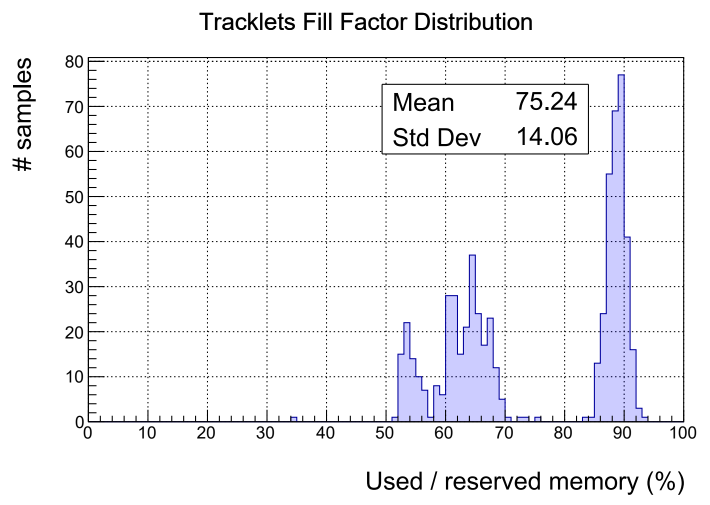
This results is mainly due to the fact that the actual number of retained tracklets is related to a single filtering phase, while the quantity of reconstructed cells depends on both the amount of the related tracklets coming from the previous algorithm phase and the application of the filtering criteria during the second phase. This double selection step introduces a significative divergence from the simple combinatorial model described in section 2.1.3. Memory occupancy distributions for outer layers are more stretched, and the same is obtained for events with multiple interaction vertices.

An important application of the memory occupancy model is the implementation of a memory preallocation strategy, that plays an important role in the algorithm optimisation. Indeed, the resize operations are quite expensive in a classical CPU programming model (as pointed out in section 2.2.3) and not allowed in GPU heterogeneous programming context. Relations 2.31 and 2.32 can be used to estimate the values of N_T and N_C for each layer and to reserve the appropriate amount of memory to the respective data structures.

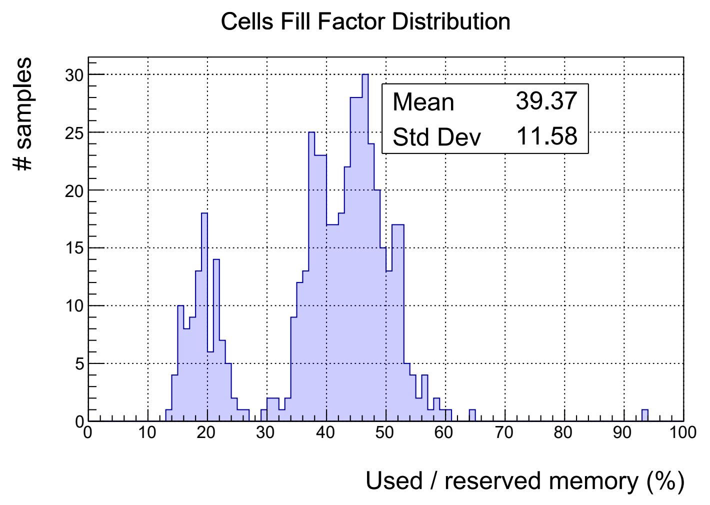
Since it is impossible to resize device-allocated structures, the risk of underallocation must be minimised, because an overflow would force the entire track reconstruction process to be restarted. This means that also outliers like the one visible in figure 2.5b must be taken into account. This is the reason why the preallocation factors are computed as the mean of the memory occupancy distribution plus ten times its distribution RMS.

Figure 2.6 shows the distribution of the fill factor, namely the ratio between the used and the allocated memory, for tracklets and cells data structures. Tracklets come with a 75% average fill factor, that is a quite good result, while cells data structures are considerably overallocated, with an average fill factor of less than 40%. It is therefore necessary to correctly handle the outlier visible on the right side of the histogram 2.6b.

Figure 2.7 shows the distribution of the ratio between real and theoretical memory occupancy for track candidates with 7 clusters. The histogram immediately reveals how this distribution is much less regular with respect to the ones shown before, with many outliers really far from the mean value. Therefore, it would be really difficult to make a suitable prediction for roads data structure occupancy based only on the input size. Appendix A reports all the memory occupancy benchmark results for a sample of 100 Pb–Pb central events simulated without pile-up.



(a)



(b)

Figure 2.6: Distribution of tracklets (top) and cells (bottom) data structures fill factor for all layers for 100 central Pb–Pb events without pile-up.

2.3.3 Computing time benchmark

In order to identify possible bottlenecks and to plan a good optimisation strategy, it is necessary to analyse the time spent by the algorithm during each step. The suite of Callgrind and KCachegrind tools [31] has been chosen to realise the sequential performance

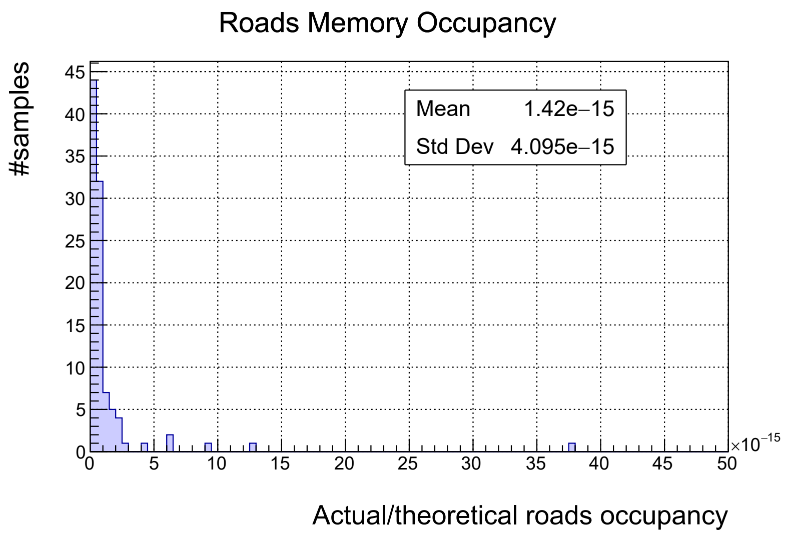
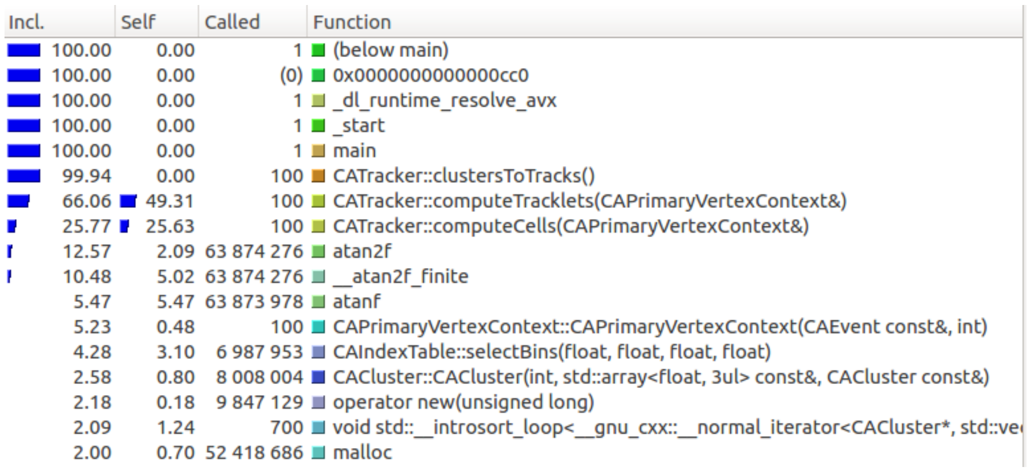


Figure 2.7: Distribution of the ratio between the real and the theoretical complete roads occupancy for a sample of 100 central Pb–Pb events without pile-up, where the denominator is the number of all possible combinations between clusters across all the 7 ITS Upgrade layers: $\prod_{i=1}^{7} \alpha_i N$

analysis. All benchmarks described in the current section have been realised with the following configuration:

- CPU: Intel i7-7700K (4.2GHz, 8MB Cache)
- RAM: Corsair DDR4 32GB (2×16) 2133MHz
- OS: Linux Ubuntu 16.04 LTS
- C++ Compiler: Clang 3.8.0-2ubuntu4, with -O3 optimisation flag

Figure 2.8 shows a list of the most time consuming algorithm functions, sorted in descending order, for a simulation with a sample of 100 central Pb–Pb events without pile-up. Looking at the leftmost column of the table, one can see that both the tracklet finding (7th column) and cell finding (8th column) phases represent by far the bottleneck of the entire process, with a total consumed CPU time greater than 91%. By observing also the second column from the left, one can observe that, while most of the computational time related to cell finding phase is spent into the related function itself, about 25% of the CPU time spent to find tracklets is related to external calls. In particular, looking at figure 2.9, that shows the callee map for tracklet finding phase, it appears that the computation of atan2 (see equation 2.9) has a significant impact on the computing time. As things stand,



Incl.	Self	Called	Function
100.00	0.00	1	(below main)
100.00	0.00	(0)	0x0000000000000cc0
100.00	0.00	1	_dl_runtime_resolve_avx
100.00	0.00	1	_start
100.00	0.00	1	main
99.94	0.00	100	CATracker::clustersToTracks()
66.06	49.31	100	CATracker::computeTracklets(CAPrimaryVertexContext&)
25.77	25.63	100	CATracker::computeCells(CAPrimaryVertexContext&)
12.57	2.09	63 874 276	atan2f
10.48	5.02	63 874 276	__atan2f_finite
5.47	5.47	63 873 978	atanf
5.23	0.48	100	CAPrimaryVertexContext::CAPrimaryVertexContext(CAEvent const&, int)
4.28	3.10	6 987 953	CAIndexTable::selectBins(float, float, float, float)
2.58	0.80	8 008 004	CACluster::CACluster(int, std::array<float, 3ul> const&, CACluster const&)
2.18	0.18	9 847 129	operator new(unsigned long)
2.09	1.24	700	void std::__introsort_loop<__gnu_cxx::__normal_iterator<CACluster*, std::ve
2.00	0.70	52 418 686	malloc

Figure 2.8: List of the most time consuming function calls displayed by KCachegrind, for a sample of 100 central Pb–Pb events without pile-up. The leftmost column refers to the percentage of time spent into a specific function, including also inner function calls, while the adjacent column reports only the time spent in the function itself, excluding inner calls.

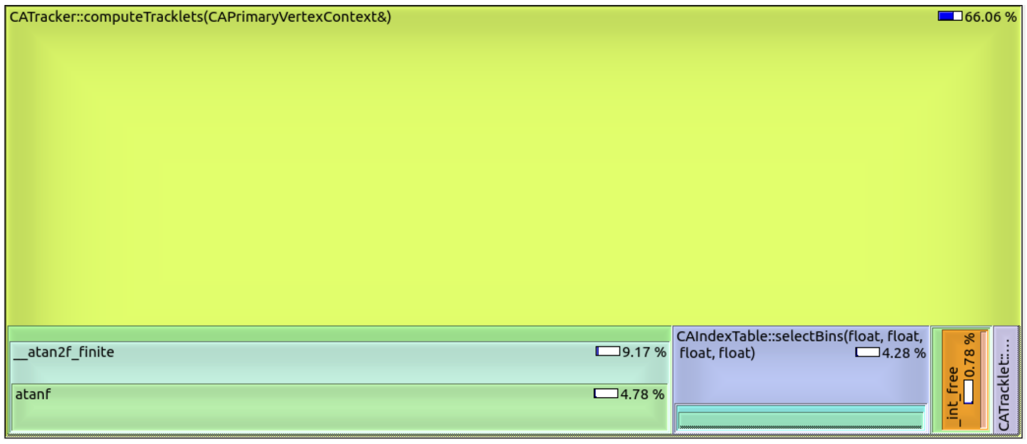


Figure 2.9: Callee map for tracklet finding phase, displayed from KCachegrind. Callees are drawn inside of the caller rectangle. The area size of a rectangle is proportional to the inclusive cost of the function this rectangle represents. [31]

the usage of a more efficient, maybe less precise mathematical library in place of STL utilities should result in a significant performance improvement.

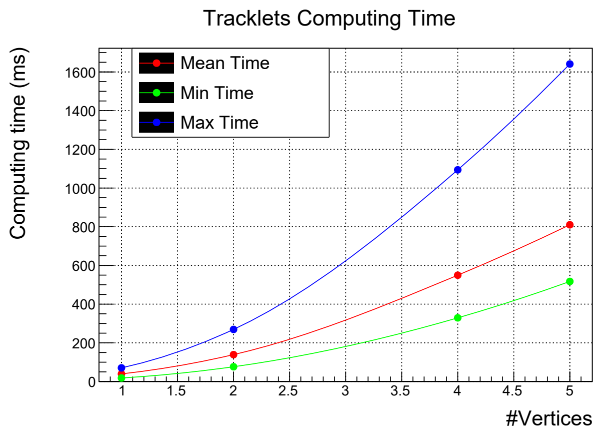
In addition to a comparative profiling of different algorithm phases, it is also important to benchmark the absolute computing time needed by the process. This value can be useful to compare the serial implementation of the tracker with the version reported in [2], in order to be sure that no important drawbacks have been introduced in terms of performance.

Furthermore a comparison with the optimised parallel version is useful to measure the obtained speedup.

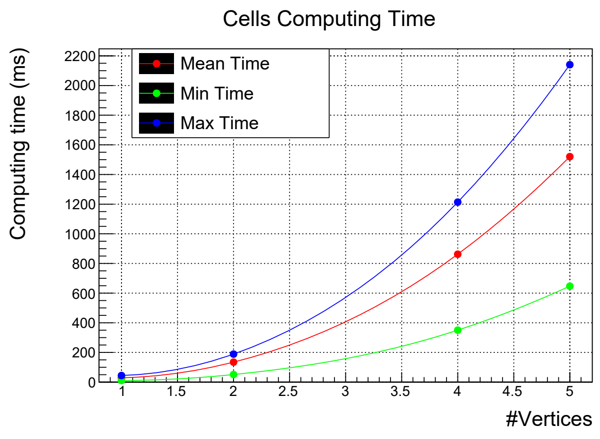
Table 2.1: Algorithm computing time measurements for tracklet and cell finding phases and for the whole process (in ms) analyzing 100 Pb–Pb central events having different amount of pile-up.

# Vertices		1	2	4	5
Tracklet finding	Min	18.2	76.4	329.1	517.3
	Mean	39.4	139.0	549.6	810.0
	Max	70.6	269.3	1094.0	1641.0
Cell finding	Min	10.2	50.9	350.3	646.9
	Mean	28.5	134.5	862.4	1520.3
	Max	44.4	188.7	1213.6	2140.7
Total	Min	33.4	135.0	696.2	1186.2
	Mean	75.0	285.4	1437.7	2362.8
	Max	110.3	446.5	2326.1	3810.5

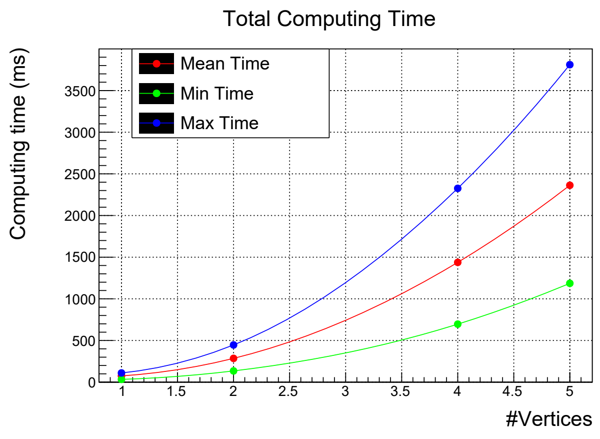
Table 2.1 reports measurement of minimum, mean and maximum computing time values for tracking and cell finding phases and for the whole process, with taking into account different amount of pile up in the events. It appears that, for large input sizes (4 or more interaction vertices) cell finding phase becomes more expensive than the previous one, in accordance with the theoretical model (equations 2.13 and 2.22). This is clearly noticeable in figure 2.10, that shows a graphical representation of the computing time trend for the values reported in table 2.1. By comparing the sum of tracklet and cell finding phases computing times with the total time spent by the process, it is clear how, also with multiple interaction vertices, these two phases continue to occupy the majority of the CPU resources, while subsequent steps, despite the higher theoretical computational complexity, remain always negligible.



(a)



(b)



(c)

Figure 2.10: Minimum (green), mean (red) and maximum (blue) computing time trends for the tracklet (top) and the cell (middle) finding phases and for the whole process (bottom) when 100 Pb–Pb central events having different amount of pile-up are analyzed.

Chapter 3

GPU Implementation Analysis

Modern GPUs are becoming an increasingly common tool to accelerate computational expensive but embarassignly parallel algorithms in the context of scientific computing. Indeed, these architectures provide thousands of low-power cores for highly parallelised performance and for offloading compilation from the CPU [24]. Moreover, if it was initially quite challenging to efficiently implement an algorithm using graphics specific device functionalities, the introduction of General Purpose GPU (GPGPU) architectures and drivers, like CUDA and OpenCL, made easier to fully exploit GPU computational power.

For all these reasons, reference [24] indicates both NVIDIA GPUs, based on CUDA, and AMD GPUs, based on OpenCL, as potential accelerator architectures for the O^2 facility, together with AMD Fusion System on Chip (SoC) and Xeon Phi board.

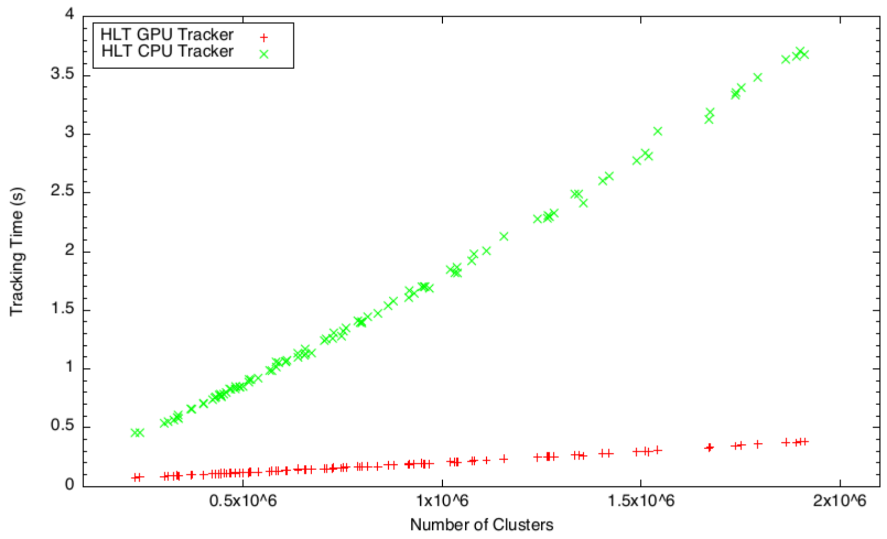


Figure 3.1: Tracking time of HLT TPC CA algorithm on Nehalem CPU (6 Cores) and NVIDIA Fermi GPU [24].

In the more specific context of CA based tracking reconstruction algorithms, GPU-oriented implementations have already been adopted by some experiments, like Antiproton Annihilation at Darmstadt (PANDA) [32] and ALICE itself for the TPC reconstruction phase. Regarding the latter, figure 3.1 shows a comparison of the algorithm execution times obtained with CPU (green line) and GPU (red line). It is clearly noticeable how substantial the speedup introduced by the GPU is, especially for an high number of clusters being processed.

Therefore, it is worth to analyze in depth the GPU migration possibilities for the ITS Upgrade tracking algorithm and this represents the main goal of my thesis project. Nevertheless, the proper tuning up of a program for a GPU execution is a complex operation, because it requires to adapt the code to the device memory hierarchy, that is radically different from the usual memory configurations, and finally to optimise the program at the instruction level.

A detailed description of the GPU programming model, in which CPU and GPU cooperate to complete the same task in a more efficient way, together with a general overview of the main GPU optimisation criteria, will be discussed in section 3.1. Since the program realised for this thesis work has been developed with NVIDIA CUDA framework, such environment will be described in the aforementioned section, but almost all concepts can be translated to OpenCL paradigm effortlessly (often there is a mere terminology mismatch between the two).

The actual application of GPU optimisation techniques to the ITS Upgrade tracking reconstruction algorithm, together with a detailed presentation of the adopted parallelisation strategy, can be found in section 3.2.

Finally, in section 3.3 an analysis of the parallel implementation performances is reported, together with some considerations for potential further optimisations.

3.1 CUDA programming model

3.1.1 Parallel architecture

The GPUs hardware is more specialised for highly parallel tasks with high arithmetic intensity with respect to the standard CPU architectures, because more transistors are devoted to data processing at the expense of data caching and flow control. These design choices have been initially driven by the high computing power required to run high-definition, real time graphics applications (i.e. 3D rendering, video codec or pattern recognition), where large sets of pixels are mapped to parallel independent threads.

Actually, GPUs have proved to be able to accelerate also algorithms outside the field of graphics, stressing the necessity to general purpose architectures and programming models. It is precisely for this reason that in November 2006 NVIDIA introduced CUDA, a general purpose parallel computing platform that comes with a software environment based on C programming language [33].

CUDA aims to allow programmers to develop software that transparently scales its parallelism according to the number of available processor cores, by introducing a level of abstraction over thread parallelism, barrier synchronisation and memory management. A programmer can easily distinguish between a coarse-grained data and task paralellism, useful to partition a problem in a set of independent subtasks, and a fine-grained thread parallelism, in which threads can cooperate to solve one of these subtasks in an efficient way.

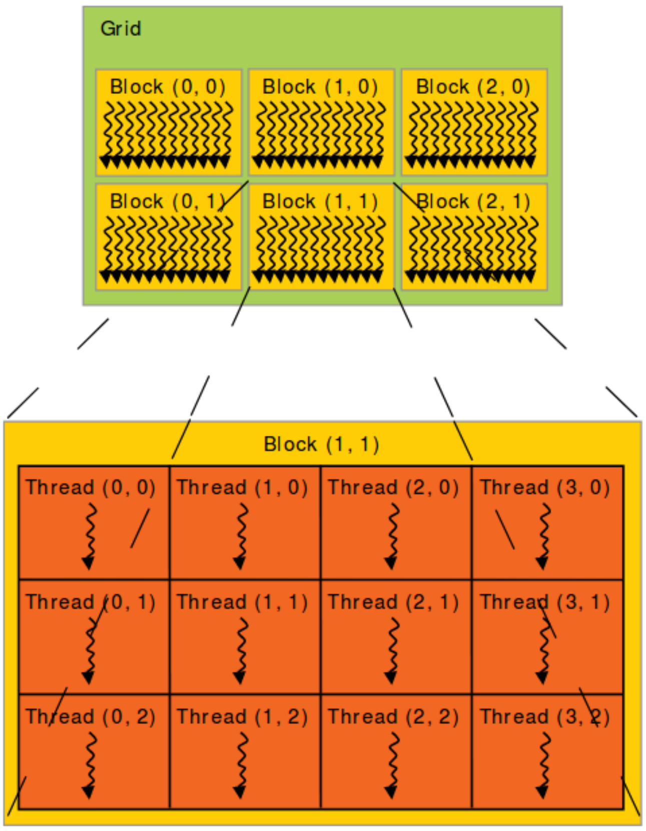


Figure 3.2: CUDA threads organisation in the context of a single kernel execution [33].

GPU parallelisation is managed by specific functions called *kernels* that, when called, are executed N times in parallel by N different CUDA threads. More specifically, CUDA introduces a specific C language extension to allow the programmer to explicitly set the execution configuration of each kernel acting on two values:

- the amount of threads per *block*. Each thread block can be a 1-dimensional, 2-dimensional or 3-dimensional set of threads that can cooperate by sharing data and synchronizing memory accesses;

- the amount of blocks. Blocks are organised into a 1-dimensional, 2-dimensional or 3-dimensional *grid* and are required to execute independently, because no control on their execution order is possible.

Figure 3.2 shows the threads organisation for a single kernel execution in CUDA programming model.

Each thread can be identified by a 1-dimensional, 2-dimensional or 3-dimensional *thread index* T_{B_i} that is unique inside a block. In the same way, each block is identified by a 1-dimensional, 2-dimensional or 3-dimensional *block index* B_i. For a 2-dimensional case, if each block has size $D = \{d_x, d_y\}$ and there are $G = \{g_x, g_y\}$ blocks into the current grid, the global index T_i of a thread with index $T_{B_i} = \{t_x, t_y\}$ belonging to a block with index $B_i = \{b_x, b_y\}$ can be expressed with the following relation:

$$T_i = \left(b_x + b_y g_x\right) \cdot d_x d_y + \left(t_x + t_y d_x\right) \tag{3.1}$$

NVIDIA GPU architecture is composed by a scalable array of multithreaded Streaming Multiprocessors (SMs) to which the parallel workload is equally distributed. Threads of a single block are forced to execute concurrently on the same SM, while multiple blocks are not necessarily executed on a single multiprocessor.

In order to allow each multiprocessor to concurrently execute hundreds of threads, a Single Instruction Multiple Thread (SIMT) architecture is employed. Threads are created, scheduled and managed in groups of 32, called *warps*, that start together at the same program address but have their own instruction counter and register state, so that they are able to branch independently. If different threads of a warp diverge because of a data-dependent conditional branch, all taken paths are serially executed so that threads that are not on the currently processed path are temporarily disabled and all active threads execute one common instruction at time. Therefore, full efficiency is realised when all 32 threads of a warp share the same execution path.

If T is the number of threads in a block, the number of warps N_W in that block can be expressed by the following relation:

$$N_W = \left\lceil \frac{T}{W_{size}} \right\rceil \quad \text{where} \quad W_{size} = 32 \tag{3.2}$$

This means that each block will always contain a number of threads that is a multiple of W_{size}, with a subset of them permanently disabled if the number of threads per block set in the kernel launch configuration is not a multiple of W_{size}. Therefore, the full efficiency is obtained by setting, for each kernel, a number of threads per block equal to a multiple of the warp size.

3.1.2 Heterogeneous programming

CUDA programming model assumes that kernels are launched from the *host*, namely the hardware that processes the main program flow, and executed on a phisically separated,

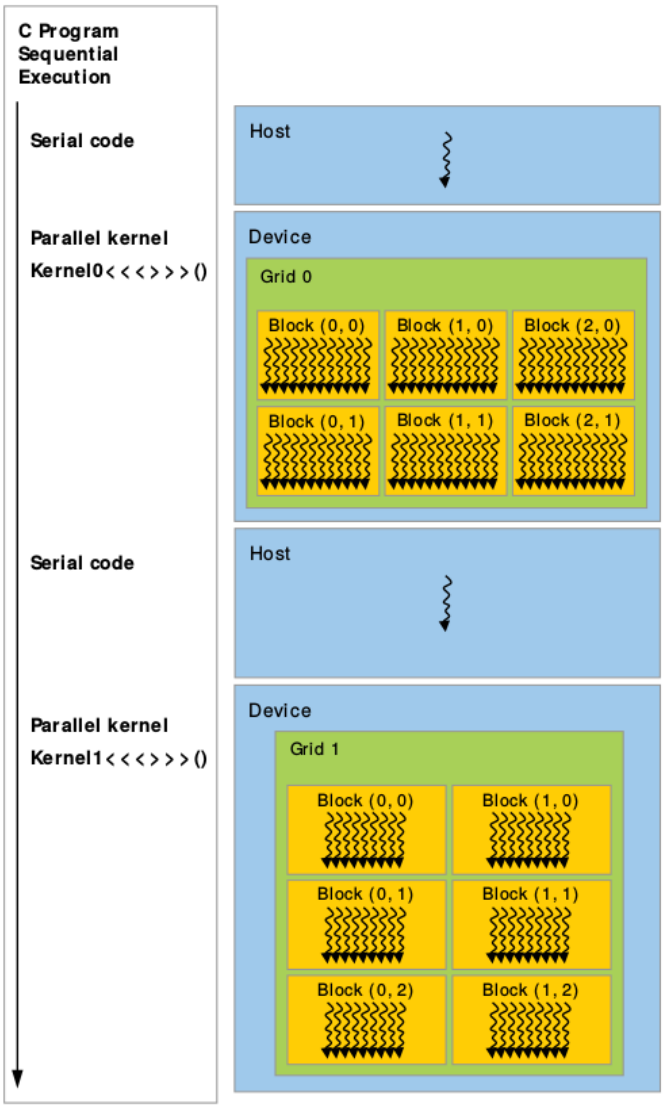


Figure 3.3: CUDA heterogeneous programming execution flow [33].

CUDA-enabled *device* that operates as a coprocessor. Moreover, both the host and the device maintain their own separate memory spaces, referred to as *host memory* and *device memory*, so data must be explicitly transferred from host to device in order to be processed by kernels and must be sent back to the host memory to be read by the subsequent program instructions. Such a programming model is called *heterogeneous programming* and is shown in a schematic way in figure 3.3.

Kernel launches are asynchronous, in the sense that the control is returned to the host before the device completes all the requested tasks. If at some point the host needs to await a kernel to finish, it must be implicitly or explicitly synchronised with the device.

In contrast, different kernels are normally executed following a precise order, notwithstanding the fact that some NVIDIA architectures are able to execute multiple kernels concurrently. CUDA framework provides *streams* to manage concurrency between different kernels. Each kernel can be associated to a specific stream by explicitly setting a stream identifier in its launch configuration: all kernels belonging to the same stream are executed in order, while kernels from different streams can be executed in any order. If no stream identifier is specified, a kernel is associated to the *default stream* and it is executed in order with respect to all other kernels without an explicit stream association. Therefore, to obtain a higher speedup it is important to specify different streams for all those device tasks which executions can be safely overlapped.

As mentioned above, host and device use independent memory spaces. Indeed, in order to allow a kernel to process program data, the host code must allocate some space on device memory and copy relevant data into it before launching the kernel itself. Then, in order to allow host code to process kernel output, data must be copied back to host memory and device memory must be freed when it is no longer required.

While device memory allocation and release are always synchronous operations, some NVIDIA hardware architectures are able to overlap data transfers between host and device with both kernel executions and other data transfers. Nevertheless, this feature requires the involved host memory to be marked as *page-locked* (or *pinned*), in order to prevent it to be swapped out. Additionally, this kind of memory provides an improved transfer bandwidth. According to [34], however, pinned memory should not be overused for the following reasons:

- an excessive use of page-locked memory can reduce the overall system performance, since it is a scarce resource;
- the pinning of system memory is an heavyweight operation compared to standard system memory allocations, so it should be used only when the amount of memory to be transferred in each step is big enough to hide this additional cost.

Dynamic in-kernel global memory allocations are supported by some CUDA-enabled devices, by calling `malloc` and `free` functions from device code. Memory allocated with `malloc` lives until the end of the program, or until it is explicitly released by calling `free` or by resetting device context. Nevertheless, dynamic allocations are very slow and can therefore cause an important performance degradation of the whole program execution, so they should be avoided whenever possible.

3.1.3 Device memory hierarchy

As shown in figure 3.4, CUDA device memory is divided into different spaces, each one optimised for a specific functionality. Each thread has its private *local memory*, while each block can reserve a portion of *shared memory* that is common to all its threads and has

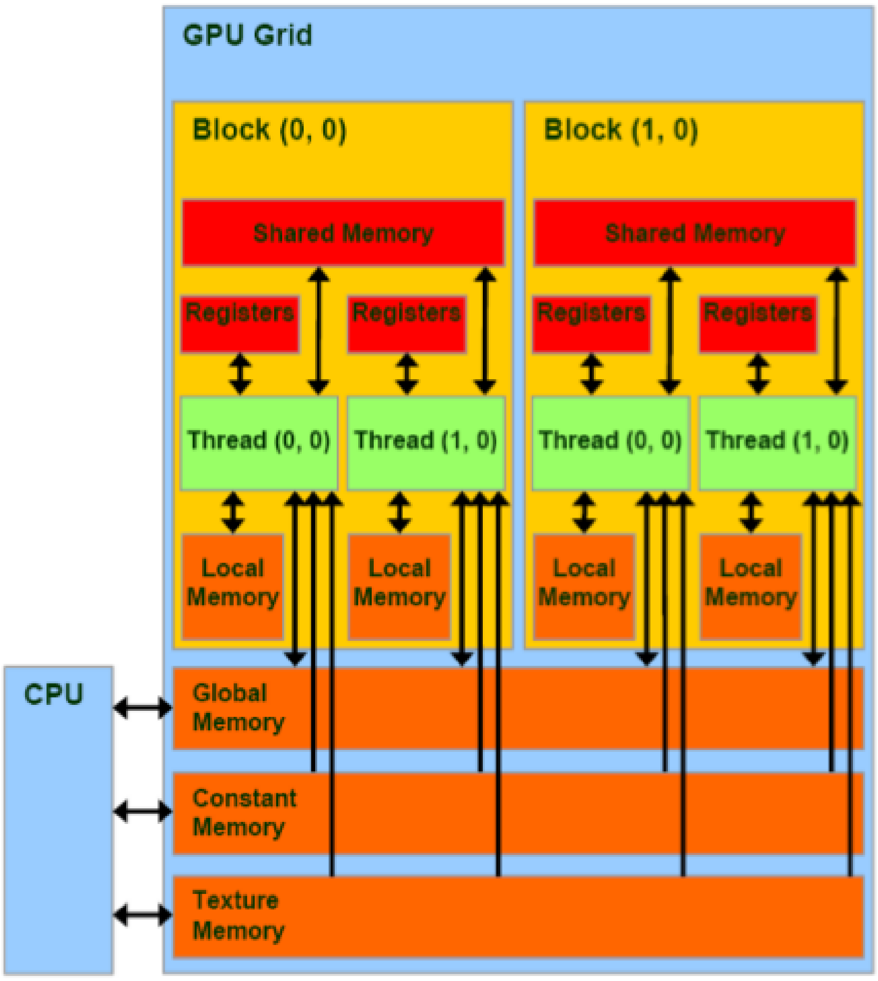


Figure 3.4: CUDA memory model [35].

the same lifetime of the block itself. All threads have access to the whole *global memory*, where host-driven allocations and transfers take place. *Constant memory* and *texture memory* are two additional read-only memory spaces that, as for global memory, are persistent across different kernel launches and can be accessed from the host application through dedicated CUDA functions. Finally, an important role is played by the 32-bit registers partitioned among active thread warps.

Global memory resides in device memory and is accessed through 32-byte, 64-byte or 128-byte transactions. Full efficiency is realized when such transactions are coalesced: when a warp executes an instruction that requires a global memory access, all memory accesses of the threads within the warp are coalesced into the minimum number of required memory operations. Indeed, to obtain the best global memory throughput it is important to maximise coalesing with the following strategies:

- following the best access patterns. The optimal access pattern depends on the architecture of the adopted device;
- using data types that meet the size and alignment requirements of the coalesing feature. More specifically, global memory instructions support read/write operations on naturally aligned words of size equal to 1, 2, 4, 8 or 16 bytes. If necessary, padding can be added to the data in order to force their alignment and to meet the requirements.

Local memory resides in device memory too, so accesses have the same high latency and low bandwidth of the global memory ones and should be coalesced whenever possible. Additionally, both global and local memory accesses are always cached by default (in both L1 and L2 caches for older architectures, only in L2 cache for newer ones). According to [33], the compiler is likely to place in local memory the following variables:

- arrays that are accessed with dynamic indices, namely by indices which values can't be figured out at compile time;
- large data structures that would heavily consume register space;
- if the available number of registers is reached, any remaining variable is placed in local memory.

Excluding delays due to read-after-write dependencies and register memory bank conflicts, accessing a register consumes zero extra clock cycles [34]. Therefore, registers should theoretically be always preferred to local memory, but the heavy usage of the registers can lead to performance drawbacks. Indeed, the set of registers (*register file*) in a multiprocessor must be shared between all the thread blocks. If then each thread uses too many registers, the number of blocks that can reside on each SM is reduced. The term *register pressure* is used to refer to a situation in which there are fewer registers available than would have been required for an optimal execution.

In other words, an excessively high number of registers used by each thread can lower the *occupancy* of the process, namely the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. A lower occupancy leads to a lower degree of parallelisation of the tasks and a less efficient ability to hide memory latency, thereby causing performance degradation.

According to [34], the shared memory has much higher bandwidth and lower latency than local and global memory, because it resides on a chip. Shared memory is divided into equally sized modules, called *banks*, in a way that any memory load or store of n consecutive addresses span n distinct memory banks and can therefore be serviced simultaneously. On the contrary, multiple accesses to the same memory bank must be serialised, resulting in performance degradation. If multiple threads of the same warp address the same shared memory location, a broadcast access is possible. Shared memory is particularly useful to:

- enable coalesced access to global memory in case of strided access patterns;
- eliminate or reduce redundant loads from global memory;
- avoid wasted bandwidth.

Constant memory space occupies 64 KB and it is cached, so that a device memory access only occurs in case of a cache miss. If all threads of a warp access the same location in constant cache, then constant memory can be as fast as a register access, so it is particularly useful when the threads of a warp only access few distinct locations.

Texture memory is optimised for 2D spatial locality and it is cached, therefore it is particularly useful when threads of the same warp access memory addresses that are close together. Since within a kernel call the texture cache is not kept coherent with respect to global memory writes, the threads can safely read only those values that have been updated before the kernel was launched.

3.1.4 Compilation workflow

CUDA framework comes with its own Instruction Set Architecture (ISA), called Parallel Thread Execution (PTX), but it is more convenient to use an high level programming language such as C to write code and then compile it into PTX. In any case, Nvidia CUDA Compiler (NVCC) must be used to compile kernels into binary code, in order to allow them to be executed on a CUDA-enabled device.

Heterogeneous programming model provides a mixture of conventional C/C++ host code and device specific functions, that must be distinguished during the NVCC compilation workflow. According to [36], CUDA compilation workflow separates the device functions from the host code, compiles device functions using NVCC and the host code using the available host compiler and embeds the compiled GPU functions as a fatbinary image in the host object file.

PTX code can also be loaded by an application at runtime and compiled further to binary code by device driver. This operation, called *just-in-time compilation*, increases the application load time, but it allows to benefit from any new compiler improvements of new device drivers and it is the only way to allow applications to be executed by devices that did not exist at the time the application was developed. Nevertheless, it is always better to generate binary code for a specific architecture whenever possible.

NVCC predefines a set of macros that allows programmers to activate some features only for some specific CUDA compilation workflow phases. The `__CUDACC__` macro, for example, is defined only when NVCC is compiling CUDA source files and it is particularly useful to allow the same code to be reused for both host and device contexts, by hiding all the CUDA specific language extensions to the host compiler .

Each CUDA-enabled GPU supports a specific list of features offered by the framework. Such set of features is identified by the *compute capability*, namely a version number X.Y composed by a major revision number X that denotes the core architecture (i.e. Kepler, Maxwell and so on) and a minor revision number Y that corresponds to an incremental improvement of the core architecture, with a possible inclusion of new features.

Binary code (*cubin*) is architecture specific, so binary compatibility is guaranteed from one minor revision to the next one, but not from one minor revision to the previous one or across major revisions. On the contrary, PTX code produced for some specific compute capability can always be compiled to binary code of greater or equal compute capability.

Prior to CUDA 5.0 a separate compilation was not supported for device code, so it was not possible to call device functions or to access variables across files [36]. From the CUDA 5.0 release on, separate compilation of device code can be explicitly activated with some dedicated options. This allows programmers to increase device code modularity and maintainability.

3.2 Parallel GPU implementation

3.2.1 Application assessment

The first necessary step to develop an efficient CUDA implementation of an algorithm is to define an effective parallelisation strategy, capable to obtain the maximum speedup without giving up code maintainability and cross-device compatibility and that takes into account the strong and weak points of the heterogeneous programming model.

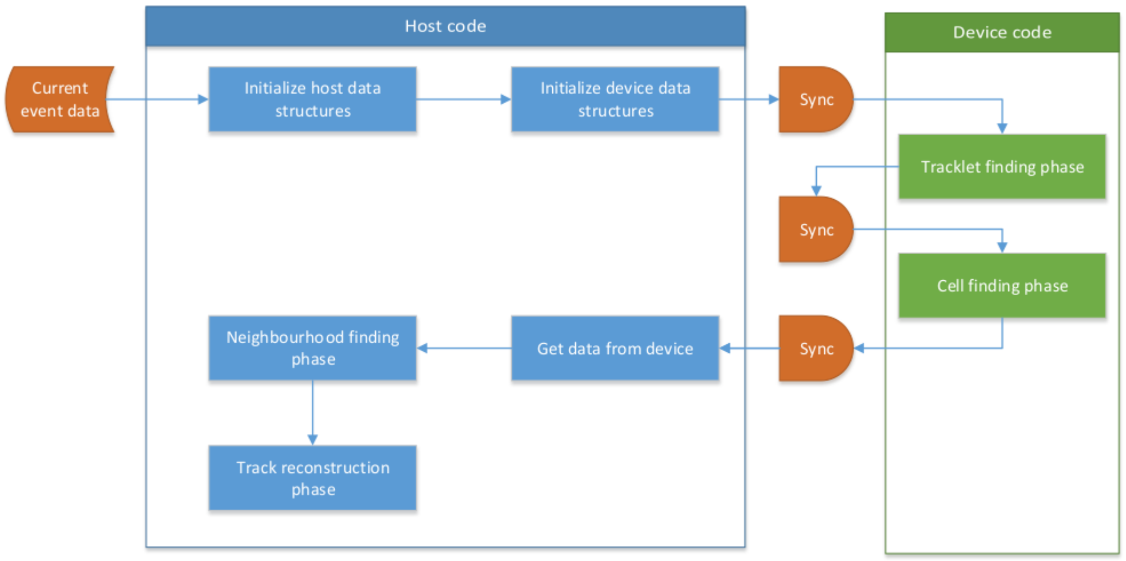

Host code
Current event data
Initialize host data structures
Initialize device data structures
Sync
Device code
Tracklet finding phase
Sync
Cell finding phase
Neighbourhood finding phase
Get data from device
Sync
Track reconstruction phase

Figure 3.5: Distribution of the algorithm steps in an heterogeneous programming model. Host code portions are blue filled, device ones are green filled and orange is used for external data and for synchronisation barriers.

As a first thing, it is necessary to identify which portions of the program should be executed on device and which ones should continue to be part of the host code. Performance results obtained for the serial implementation of the CA algorithm (and reported in section 2.3) motivated the choice to migrate on GPU only the tracklet finding and cell finding phases. Figure 3.5 shows the distribution of the algorithm steps between host and device with the chosen parallelisation strategy; as it can be seen, such implementation requires at minimum three synchronisation barriers between host and device environments.

The result that mainly drove the parallelisation strategy choice is the computing time percentage spent on each algorithm phase. By analysing the data reported in table 2.1, it is clear how, on average, more than 92% of the overall computing time is spent on tracklet and cell finding phases when input data contain a single interaction vertex and how such time percentage increases with the amount of pile-up. According to Amdahl's Law, the maximum speedup S that can be expected by parallelising a portion of a serial program can be expressed by the following relationship, where P is the fraction of the total serial execution time taken by the portion of code that should be parallelised and N is the number of processors over which such portion of the program runs [34]:

$$S = \frac{1}{(1-P)+\frac{P}{N}} \tag{3.3}$$

Therefore, the parallelisation of the slowest algorithm phases leads to the maximum theoretical speedup. In particular, since modern GPUs are highly parallel devices, the Amdahl's Law can be simplified by considering an infinite number of processors:

$$\lim_{N\to+\infty} S = \frac{1}{1-P} \tag{3.4}$$

In this case, the maximum theoretical speedup factor achievable with the chosen parallelisation strategy is around 13. Nevertheless, as discussed in section 3.1.2, data need to be transferred from host memory to device memory in order to be processed by kernels and viceversa. The overhead introduced by these operations and the fact that real applications rarely exhibit a perfectly linear scaling make it impossible to reach the theoretical limit.

Another important aspect that emerges from the results reported in section 2.3.2 is the greater memory occupancy predictability for tracklet and cell finding phases, with respect to subsequent algorithm steps. As pointed out in section 3.1.2 dynamic in-kernel memory allocation leads to a considerable performance degradation, so host-driven memory allocation should be preferred. Furthermore, this require a suitable memory occupancy prediction, because:

- an underallocation would have disastrous effects on performance, since it would be necessary to interrupt the kernel execution, free the old allocated memory, allocate a greater portion of memory and then launch the kernel again;
- since device memory allocation is a costly operation, an excessive overallocation would result in a slower initialisation phase for algorithm data structures.

By observing figure 2.7 it is clear how difficult would be to implement a memory pre-allocation strategy for the track candidates data structure, while tracklet and cell finding phases show a far more regular memory occupancy distribution.

Nevertheless, also tracklet and cell finding phases parallelisation presents some challenges [1]. In particular, a major issue comes with the index tables compilation process, described in section 2.1.3: while it is easy to know the position of the first element that starts from a particular subelement when all elements starting from the same subelement are processed one after another, it is a not so easy task when elements starting from different subelements are processed in parallel and stored out of order. On the other hand, without such data structures search, the operations would stop to have a constant computational complexity, introducing an heavy performance degradation in cell finding and neighbour construction phases.

Finally, also the simple `emplace_back` operation is costly in a SIMT architecture, because it must be atomic in order to avoid undefined behaviour; but if an atomic instruction executed by a warp reads, modifies and writes to the same location in global memory for more than one of the threads of the warp, all the operations have to be serialised [33].

3.2.2 Parallelisation strategy outline

A possible solution to the aforementioned parallelisation issues would be to have a single thread that processes each starting subelement and to run both tracklet and cell finding algorithms two times. In particular:

- the kernel is launched a first time to find, respectively, the number of valid tracklets associated to each cluster and the number of valid cells associated to each tracklet;
- the index table is compiled, by running a prefix sum [2] algorithm on the data structure containing the number of valid elements found for each subelement;
- the kernel is launched a second time to actually store found elements into the respective data structure, knowing exactly where to insert each one.

The related pseudocode is reported below:

```

void kernel(bool storeResult) {

```

[1] Since the following considerations are valid for both tracklet and cell finding phases, in order to increase readability, the term *element* will be used to refer to both tracklets and cells simultaneously. Furthermore, the term *subelement* referred to an element will mean a cluster stored into a tracklet or a tracklet stored into a cell.

[2] Let's consider the symbol $\circ$ to be an associative operation on a domain D. The prefix problem is to compute, for given $x_1, \dots, x_n \in D$, each of the products $x_1 \circ x_2 \circ \dots \circ x_k$, $1 \le k \le n$ [37].

```
  Subelement currentSubelement = subelements[threadId];
  std::vector<Subelement> involvedSubelements =
    getInvolvedSubelements(threadId);

  for (int iSubelement = 0; iSubelement < involvedSubelements.size();
   ++iSubelement) {

    if(isValidElement(currentSubelement, involvedSubelements[
   iSubelement])) {

      if(!storeResult) {

        ++validElements[threadId];

      } else {

        --validElements[threadId];
        elements[indexTable[threadId] + validElements[threadId]] =
          Element(currentSubelement, involvedSubelements[iSubelement]);
      }
    }
  }
}

void hostFunction() {

  kernel(false);
  compileIndexTable();
  kernel(true);
}

```

Such solution solves both issues, because all the elements spanning a particular subelement are stored adjacently into the data structure, as for the serial implementation, and there is no need for atomicity in `emplace_back` function because the insertion code knows exactly the position of each element. Nevertheless, this implementation is quite slow, because it requires to launch three kernels serially everytime a tracklet or cell finding step is performed.

A better approach is to group the two runs of the same phase in a single kernel launch, in order to limit the amount of introduced overhead and to avoid global device synchonisation barriers between different kernels. The idea here is to use CUDA *warp functions*, that allow the programmer to know, in a specific thread, the status of all other threads into the same warp and also to get the value that a particular variable assumes into another thread of the warp. In particular:

- during the first run, each thread computes the number of valid elements starting from a specific subelement and stores it into a local variable;

- CUDA warp functions are used to perform a prefix sum on such variable, so that each thread knows the number of elements found by itself and by all the previous threads in the warp;
- in this way the last thread in the warp, that knows the total amount of elements found into the current warp, is able to atomically reserve an adequate portion of the related data structure memory and to propagate the start index to all the other threads using CUDA warp functions [38];
- during the second run, each thread inserts each found element into its reserved bins.

The related pseudocode is reported below:

```

void kernel() {

  Subelement currentSubelement = subelements[threadId];
  std::vector<Subelement> involvedSubelements =
    getInvolvedSubelements(threadId);
  int validElements = 0, startIndex;

  for (int iSubelement = 0; iSubelement < involvedSubelements.size();
    ++iSubelement) {

    if(isValidElement(currentSubelement, involvedSubelements[
      iSubelement])) {

      ++validElements;
    }
  }

  validElements = warpPrefixSum(validElements);

  if(isLastWarpThread(threadId)) {

    startIndex = subelements.atomicReserve(validElements);
  }

  startIndex = warpGet(startIndex, lastWarpThreadId);

  for (int iSubelement = 0; iSubelement < involvedSubelements.size();
    ++iSubelement) {

    if(isValidElement(currentSubelement, involvedSubelements[
      iSubelement])) {

      --validElements;
      elements[startIndex + validElements] =
        Element(currentSubelement, involvedSubelements[iSubelement]);
    }
  }
}

void hostFunction() {

  kernel();
}

```

Such solution removes the need of launching two kernels, making each warp self consistent in the execution of its workload. Moreover, despite the reintroduction of atomic memory reservations, the collision probability is minimised because different warps do not share a SIMT context and the critical section is limited to a single sum operation. However, the double repetition of each phase creates a significant drawback to the overall performance, because validity checks are costly operations from a computational point of view.

An alternative strategy, that allows the second repetition of each phase to be avoided, consists in storing valid elements directly into their respective data structures as soon as they are found. In particular:

- a first kernel computes valid elements and stores them into their respective data structure with a warp atomic memory reservation. In addition, the number of valid elements found for each subelement is saved into another data structure;
- a second kernel compiles the index table by running a prefix sum algorithm on the data structure containing the number of valid elements found for each subelement;
- a third kernel deals with elements sorting. In the data structure thus obtained, all elements starting from the same subelement should be adjacently stored, but their exact order is not important. Since the size of each group of adjacent elements is known, because it has been stored during the first kernel run, a *counting sort* can be used to achieve this task with a $O(N)$ computational complexity. Such algorithm can be easily paralellised by assigning a subset of elements to each thread.

The related pseudocode is reported below:

```

void kernel() {

  Subelement currentSubelement = subelements[threadId];
  std::vector<Subelement> involvedSubelements =
    getInvolvedSubelements(threadId);

  for (int iSubelement = 0; iSubelement < involvedSubelements.size();
   ++iSubelement) {

    if(isValidElement(currentSubelement, involvedSubelements[
   iSubelement])) {

      ++validElements[threadId];
      int elementIndex = subelements.warpAtomicReserve();
      elements[elementIndex] = involvedSubelements[iSubelement];
    }
  }
}

void sortData() {

  int startIndex = threadId * elementsPerThread;
  int endIndex = startIndex + elementsPerThread;

  for(iElement = startIndex; iElement < endIndex; ++iElement) {

    Element element = elements[iElement];
    int currentSubelementIndex = element.getFirstSubelementIndex();
    int offset = atomicSub(validElements[currentSubelementIndex], 1);

    sortedElements[indexTable[currentSubelementIndex] + offset] =
      elements[iElement];
  }
}

void hostFunction() {

  kernel();
  compileIndexTable();
  sortData();
}

```

Despite the presence of three different kernel launches, the overhead introduced by this approach has turned out to be smaller than a second repetition of the entire phase, because both the index table compilation and the element sorting tasks can be performed in an efficient way.

3.2.3 Implementation details

As outlined before, a good parallelisation strategy is fundamental to obtain an efficient GPU implementation of the algorithm, but there are also other important elements, such as code maintainability and portability, that must be taken into account. With appropriate implementation choices, these goals can be achieved without undermining the main performance target.

As a first thing, since the software must be able to run also on architectures that do not include a CUDA-enabled GPU, all CUDA Toolkit dependencies must not be mandatory for the program compilation. At the same time, maintainability requirements impose that as much code as possible is common to all different execution targets, but CUDA framework introduces some specific C language extensions that only NVCC can understand.

Assuming that the target device is known at compile time, CMake tool can be used to handle conditional inclusion of CUDA libraries, but this is not enough to grant a full cross-compiler compatibility, because an host compiler would not recognise CUDA specific language extensions. Moreover, to use the same code for both host and device code in an heterogeneous programming model, these extensions must be activated only when the involved code is compiled by NVCC.

In order to achieve this goal, a specific `CADefinitions` file has been implemented. Such file uses the NVCC `__CUDACC__` macro definition to know if the currently used compiler is the host compiler or NVCC itself and to consequently toggle the presence of CUDA specifiers. In particular:

- the file defines a list of macro that assume an empty value when `__CUDACC__` macro is not defined (the host compiler is active), while are translated into a CUDA specifier when `__CUDACC__` macro is defined (NVCC is active);

- functions that must be reused both in host and in device code have such macros prepended to their definition.

Such solution requires that some files are included both in host and in device code. This may lead to multiple definitions of some functions and consequently to a compilation error. In order to correctly handle this situation, a dedicated CMake utility has been implemented to copy all .cxx files into a temporary folder, changing their extension to .cu, during the compilation process. In this way, all code can be compiled directly with NVCC, preventing the aforementioned error to be thrown.

Since [24] indicates both CUDA and OpenCL based devices as potential accelerator architectures for the O^2 facility, an OpenCL compatibile version of the code will be implemented soon. The proposed `CADefinitions` file is useful also to meet this requirement, because OpenCL and CUDA are based on similar programming models, but with a slightly different syntax for C language extensions. Furthermore, all CUDA Toolkit functions have been wrapped into a vendor agnostic `CAGPUtils` adapter, in order to hide CUDA specific syntax and make libraries from different vendors interchangeable at compile time.

In particular, the `CAGPUtils` Application Programming Interface (API) contains all the utilities that are necessary to manage an heterogeneous programming environment, such as memory transfers handling, kernel launch configurations, profiling and so on.

Moreover, there are also a couple of functions able to evaluate the best kernel launch configuration by taking into account the characteristics of the GPU device on which the kernel will be executed.

In order to allow a runtime evaluation of those parameters, a vendor-agnostic `CAGPUContext` has been implemented. More specifically, a Singleton pattern has been used to realise such context:

- when called for the first time, the class constructor initialises an array of device properties for all available devices;
- the method `getDeviceProperties()` is able to automatically identify the currently used device and to return the related property set;
- the method `getDeviceProperties(deviceIndex)` can be used to get the property set of a specific device.

The Singleton pattern is efficient, because each device is queried only once, but the current implementation is not able to handle an eventual hot swap of GPU devices. Furthermore, since the target device must be chosen at compile time, architectures with mixed configurations are not supported.

The fact that all device methods have to be explicitly marked in CUDA code has an important drawback: functions from external libraries that do not natively support CUDA environment (even STL) cannot be called in the device code. In order to cope with this lack of compatibility, two heterogeneous programming aware data structures, `CAGPUArray` and `CAGPUVector`, have been implemented along the line of their STL corresponding ones. These structures allow the programmer to easily handle device memory allocation and deallocation and atomic resize operations in a transparent way, following the Resource Acquisition Is Initialisation (RAII) paradigm. Moreover, using the aforementioned `CADefinitions` file, a compile time toggle between these structures and the STL ones is made possible, because type signatures are consistent between the two implementations.

Other implementation choices have been made to speedup the execution of some heterogeneous programming specific tasks. An important thing that must be taken into account is the fact that objects cannot be simply passed by reference to kernels, because host and device have two separate memory spaces (as described in section 3.1.3). Therefore, all objects must be either passed by value or copied into device memory before the kernel launch. In order to minimise the size of kernel parameters and avoid register pressure, a single object called `CAGPUPrimaryVertexContext` is allocated in device global memory and only a pointer to its memory location is passed to kernels. Such object is constructed from the `CAPrimaryVertexContext` data structure described in section 2.2.2, using the `CAGPUVector` and `CAGPUArray` custom containers described above.

Another important aspect to consider is that both tracklet and cell finding phases can process multiple layers in parallel, since there are no data dependencies between them. Therefore, in order to speed up the process, kernels related to different layers are launched in different CUDA streams. As all other CUDA specific elements, also streams have been wrapped into a vendor agnostic object called `CAGPUStream`, that hides both the Toolkit API calls and the stream lifecycle, following the RAII paradigm.

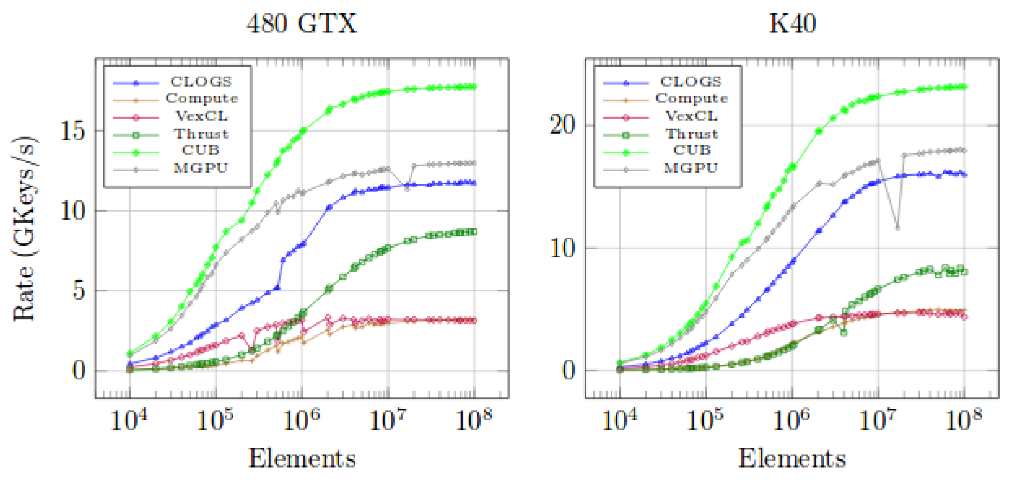


Figure 3.6: Scan performance benchmark for 32-bit elements for various CUDA based libraries [39].

Finally, since the prefix sum algorithm is widely used in GPU applications, many CUDA based, open-source libraries offer efficient and maintained implementations of it. In particular, CUB [40] library was chosen because, as shown in figure 3.6, it offers the highest performance for all ranges of input data size.

3.3 Performance analysis

3.3.1 Serial implementation comparison

The analysis of the transverse momentum efficiency and computing time described in section 2.3 for the the serial implementation of the algorithm have been repeated for the parallel implementation, in order to compare the two versions and to better quantify the performance gap. On the contrary, the memory occupancy benchmark was not repeated, because the memory model used in an heterogeneous programming model is totally different from the classic one, so a direct comparison would not be possible.

The architecture used for such analyses, reported below, contains the same hardware used to benchmark the serial implementation, with the addition of a CUDA-enabled GPU and the CUDA compiling tools:

- CPU: Intel i7-7700K (4.2GHz, 8MB Cache)

- RAM: Corsair DDR4 32GB (2×16) 2133MHz
- GPU: NVIDIA GeForce GTX 970 (SM 52, 1664 CUDA cores)
- OS: Linux Ubuntu 16.04 LTS
- C++ Host Compiler: Clang 3.8.0-2ubuntu4, with -O3 optimisation flag
- Device Toolkit: CUDA Toolkit V8.0.61

Also the sample of events used for GPU benchmarks is the same used for the serial implementation of the program. On the contrary, the Callgrind and KCachegrind suite could not be used for GPU profiling purposes. Nevertheless, CUDA Toolkit comes with it is own profiling tool, called NVIDIA Visual Profiler (NVVP), that allows programmers to deeply dive into the CUDA code performance tuning.

Figure 3.7 shows the results of tracking efficiency over transverse momentum analysis for pion-related reconstructed tracks with 7 clusters. By comparing these charts with those reported in figure 2.3, it is clear how the two implementations of the algorithm share the same portions of correct, duplicate and fake reconstructed tracks. This coherence represents a proof of correctness for the parallel implementation, since it is able to obtain the same results as the serial one. Moreover, during the data acquisition phase the NVCC flag `-use_fast_math`, that tells the compiler to substitute all CUDA math function calls with their less precise but more efficient implementations, was enabled. The fact that no significant efficiency degradation has occurred means that it is safe to leave it enabled in a production environment.

Table 3.1 reports measurement of minimum, mean and maximum computing time values for context initialisation, tracking and cell finding phases and for the whole process, with different amount of pile up in each event. In addition, for comparison purposes, the values from table 2.1 have been reported in brackets. The GPU implementation of the algorithm introduced a significant speedup in both tracklet and cell finding phases. Considering the mean values and the simulations with a single interaction vertex, a speedup of 12.7 times for the former and 5.7 times for the latter has been obtained, that lead to a speedup of 4.5 times on the whole process. Moreover, such performance gain becomes even more evident in simulations with multiple interaction vertices: the GPU implementation of the algorithm runs 10.7 times faster than the serial one when the processed event have 5 interaction vertices.

Figure 3.8 shows the computing time distribution between the different algorithm phases. An important difference with respect to the serial CPU implementation is the fact that most of the time is not spent on tracklet and cell finding phases, that together occupy less than 50% of the total computing time needed to process an event without pile-up (figure 3.8a). Instead, the context data structures initialisation phase occupies an important amount of the total time also for an event with 4 interaction vertices, as shown in figure 3.8b.

An important optimisation for this step is brought by the smart memory management implemented into contexts constructors, in order to avoid useless memory allocations (as

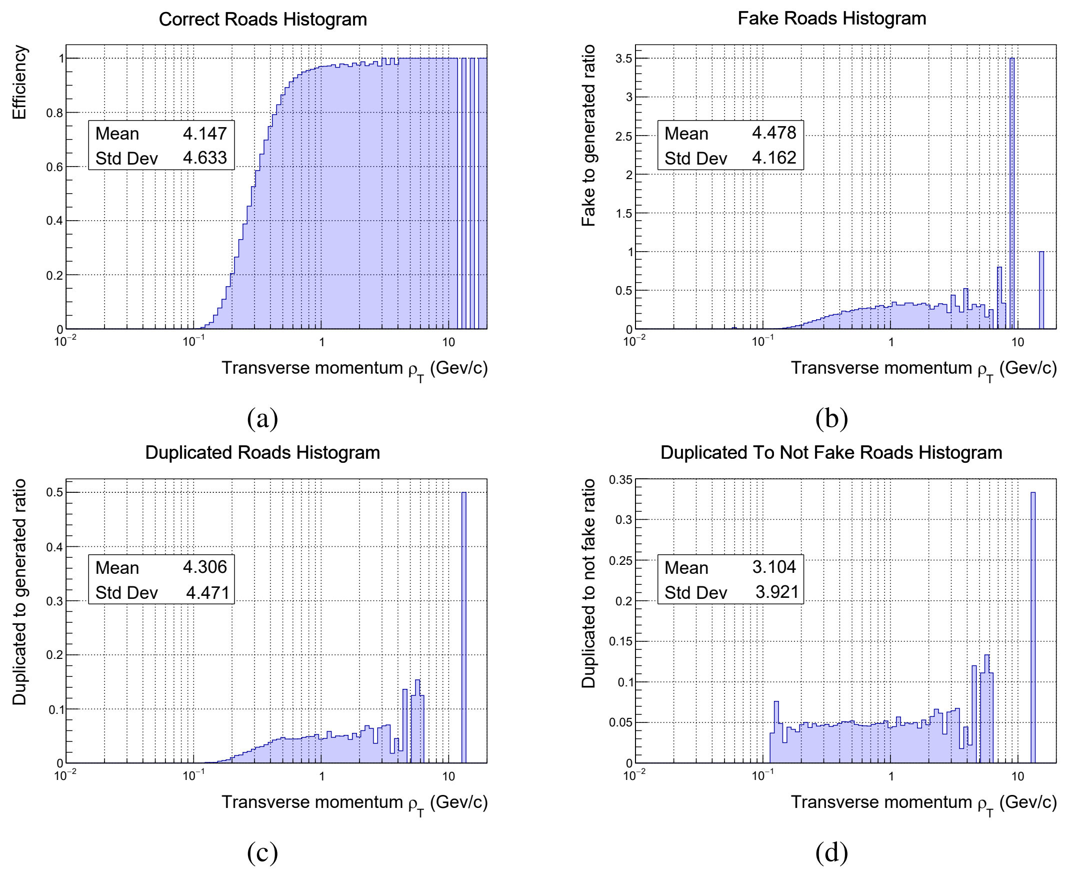


Figure 3.7: Analysis of the algorithm track reconstruction efficiency over transverse momentum for a sample of 100 central Pb–Pb events without pile-up, for correct (top left), fake (top right) and duplicated (bottom left) tracks subsets. The bottom right histogram shows the ratio of duplicated to not fake (correct + duplicated) tracks.

described in section 2.2.2). Such optimisation is even more effective in GPU implementation, because device allocations are synchronous operations and are quite expensive in terms of computing time. Nevertheless, since no initial memory allocation strategies are applied, the actual speedup strictly depends on the order in which events are processed. Figure 3.9 shows the time spent for context initialisation in each event of the data sample, with spikes corresponding to memory allocations. When the system reaches its steady state, the density of such spikes should tend to zero, with the exception of potential events with an high pile-up factor. However, it should be worth to find an appropriate memory preallocation strategy to maximise straight away the performance gain, continuing with the work described in section 2.3.2.

Another possible memory management optimisation could be the use of page-locked host memory for device transfers. Allocating pinned host memory is a costly operation,

Table 3.1: Algorithm computing time measurements for context initialisation, tracklet and cell finding phases and for the whole process (in ms) for a sample of 100 Pb–Pb central events with different amount of pile-up. Values in brackets are related to the serial implementation of the algorithm.

# Vertices		1	2	4	5
	Min	5.3	9.1	19.3	24.9
Context init	Mean	6.7	13.0	25.1	32.5
	Max	11.2	22.2	63.1	94.3
	Min	1.8 (18.2)	6.3 (76.4)	27.0 (329.1)	44.2 (517.3)
Tracklet finding	Mean	3.1 (39.4)	9.0 (139.0)	35.2 (549.6)	55.1 (810.0)
	Max	5.3 (70.6)	13.3 (269.3)	47.2 (1094.0)	70.5 (1641.0)
	Min	2.0 (10.2)	6.5 (50.9)	35.4 (350.3)	60.6 (646.9)
Cell finding	Mean	5.0 (28.5)	15.0 (134.5)	72.5 (862.4)	123.1 (1520.3)
	Max	7.9 (44.4)	20.9 (188.7)	104.4 (1213.6)	166.0 (2140.7)
	Min	9.6 (33.4)	23.4 (135.0)	84.2 (696.2)	133.3 (1186.2)
Total	Mean	16.7 (75.0)	40.2 (285.4)	139.5 (1437.7)	219.9 (2362.8)
	Max	27.0 (110.3)	59.9 (446.5)	220.7 (2326.1)	317.9 (3810.5)

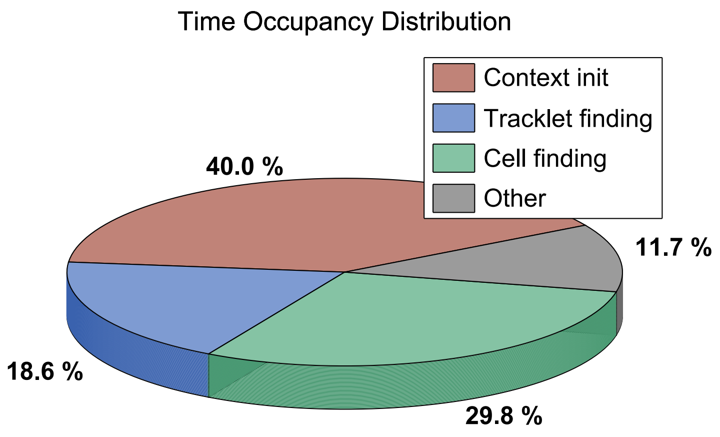
but since transfers are far more frequent than reallocations in steady state, the use of pinned memory should further optimise the context initialisation performances. Indeed, page-locked memory comes with both an higher transfer bandwidth and the possibility to parallelise multiple memory transfers, if the CUDA device has more than one copy engine.

3.3.2 GPU profiling

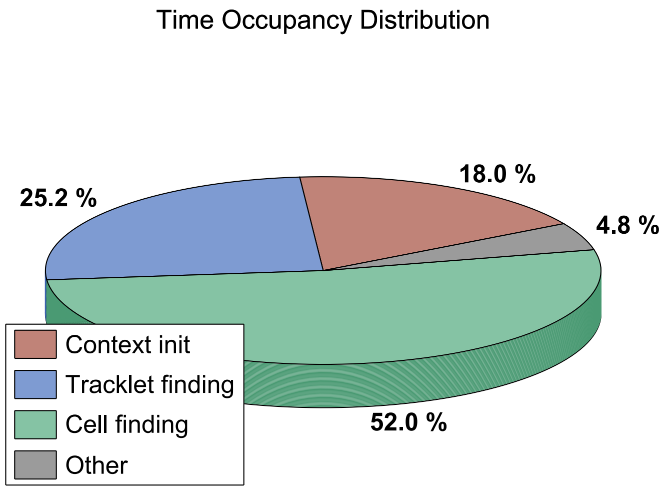
In order to further investigate the effective algorithm efficiency with respect to a modern GPU potential, it is useful to run NVVP and take a look at some of the output reports. All results in this section refer to a program running on a sample of 100 Pb–Pb central events without pile-up.

Figure 3.10 reports the NVVP execution map for all device activities related to a single event processing. By observing different lines, it is clear that tracklet (indigo bars) and cell (liliac bars) finding phases related kernels are the most time consuming. Nevertheless, also initial and final memory transfers and the tracklet sorting (fuchsia bars) need a considerable amount of time to be performed.

An important parameter is the number of registers necessary to each kernel to run. The cell finding phase kernel occupies 70 registers, while the tracklet finding phase kernel



(a)



(b)

Figure 3.8: Computing time distribution between the different algorithm phases, for a simulation with 100 Pb–Pb central events without pile-up (top) and with 4 interaction vertices (bottom)

needs 56 registers. These values are quite high and they limit the device occupancy, resulting in a potential waste of some computational power. Indeed, since each block contains 192 threads for the tracklet finding phase and 224 for the cell finding phase, no more than 6 blocks (36 warps) of the former and 4 blocks (28 warps) of the latter can run concurrently on a single SM, with a theoretical device limit of 64. This leads to a maximum theoretical

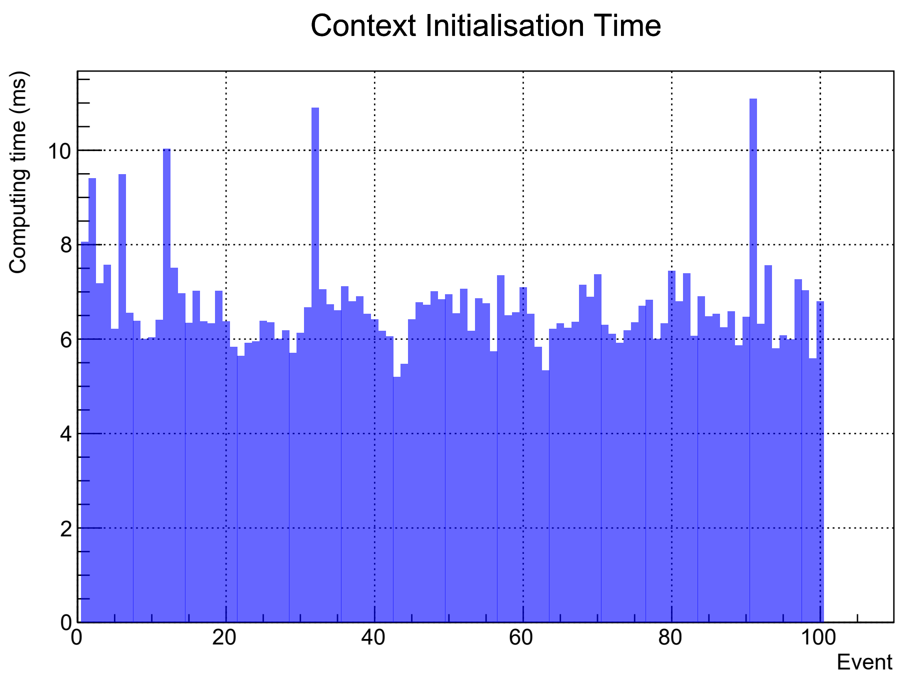


Figure 3.9: Context initialisation time for each event analyzing 100 Pb–Pb central events without pile-up. Spikes correspond to memory reallocations.

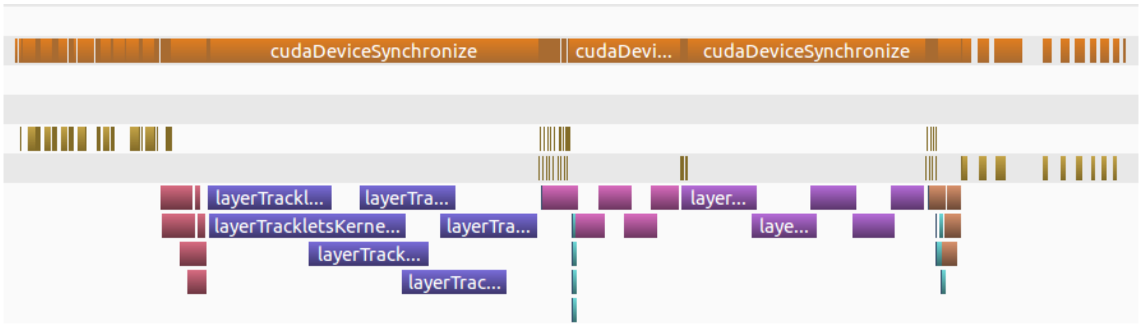


Figure 3.10: NVVP kernel execution map for a Pb–Pb central event without pile-up. The orange row reports runtime API calls executed from host code. The two golden rows report memory transfers, from host to device and from device to host respectively. Other coloured bars refer to device kernel execution. Superimposed bars are executed in parallel.

device occupancy of 56.2% and 43.8%, respectively. Nevertheless, reducing the number of occupied registers would result in a higher amount of local memory accesses and in a consequent performance degradation.

Using shared memory to store some big data structures, like clusters or tracklets, should

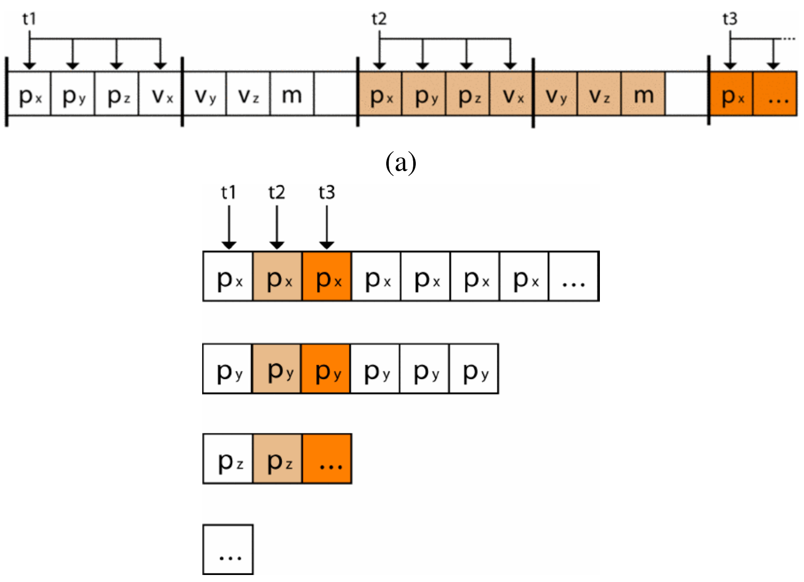


Figure 3.11: AoS (top) and SoA (bottom) memory layouts, with respective thread access patterns [41].

be a good strategy to reduce the number of occupied registers without a performance degradation. However, shared memory is optimised to have the largest bandwidth when all threads in a warp access contiguous 32-bit (the default) or 64-bit (if explicitly configured) words, but the Array of Structures (AoS) memory layout adopted for the current implementation doesn't fit well with this feature. Figure 3.11 compares the aforementioned AoS pattern with the more effective Structure of Arrays (SoA) memory configuration, which guarantees that all threads that require consecutive elements will access consecutive values in memory. Moreover, [41] presents also more complex memory layouts, that could lead to a even better memory access performance improvement. The implementation of such layouts should be considered as a promising optimisation strategy for future works, because both global and shared memory bandwidth can be increased in this way.

Another factor that limits the actual performance of the algorithm is the fact that the workload is not perfectly balanced across different threads. This is particularly evident for the tracklet finding phase, where each thread processes all tracklets starting from a single cluster. Since the number of cluster couples to process varies from one cluster to the other, some threads of a warp will remain inactive until the longest one completes its task, resulting in a lower efficiency. Figure 3.12 shows how this phenomenon leads to an unbalanced usage of the different device SMs. Nevertheless, such a different workload from one cluster to the other is a natural consequence of the fact that filtering cuts depend on the position of each peculiar cluster, so it can't be easily overcome.

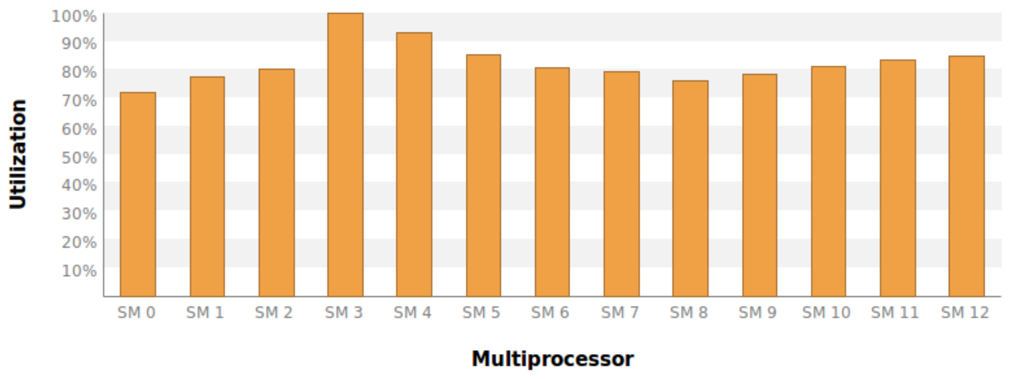


Figure 3.12: Multiprocessor utilisation for a tracklet finding phase kernel processing a Pb–Pb central event without pile-up.

Finally, observing the orange line in figure 3.10 it is clear how CPU spends a considerable amount of time waiting for the device, by calling the `cudaDeviceSynchronise` function. A multi-thread implementation of the host code, with some kind of pipeline logic, should minimise this waste of time and introduce an additional speedup in the overall process. Furthermore, also neighbourhood construction and tracklet reconstruction phases should benefit from an appropriate parallel implementation, together with the context initialisation phase, particularly if page-locked memory is used and parallel memory transfers are allowed. Nevertheless, since context switching between host threads is a costly operation, the implementation of a multi-threaded version of host code must be carefully analysed to be included in a future working plan.

Conclusions

The main goal of this thesis work was to investigate the possibilities to realise a GPU version of the CA track reconstruction algorithm developed in [2] for the ALICE ITS Upgrade.

The present ITS is the innermost detector of the ALICE detector, composed of six high-resolution cylindrical silicon detectors. It contributes to practically all the physics topics addressed by the ALICE experiment because it is crucial to determine the point where the collisions happen. A major limitation of the present running condition of the ITS detector is the poor maximum readout rate of 1 kHz, irrespective of detector occupancy. This limitation due to the hardware limits ALICE to capture only a small fraction of the full Pb–Pb collision rate of 8 kHz delivered by the present LHC and would outrageously limit the use of 50 kHz Pb–Pb collision rate provided in Run 3 (the LHC running phase that will start in 2019).

The idea for the design of an Upgraded ITS is to entirely replace the existing ITS detector with a new one, characterised by an additional layer, a smaller distance between the innermost layer and the beam pipe, a reduction of the material budget and an highly increased readout rate, up to 50 kHz for Pb–Pb collisions and 2 MHz for p–p collisions.

Such interaction rate will result in an estimated data throughput from the detector greater than 1 TB/s for Pb–Pb events during Run 3. It is therefore necessary to achieve a maximal reduction of the readout, as early as possible during the data flow, in order to minimize the cost of the computing system for both data processing and storage. The O^2 facility, the Online-Offline computing system that will assist the ALICE experiment during Run 3, has been designed to reach such a challenging goal.

The main difference between O^2 and the current AliROOT framework is the presence of synchronous calibration and reconstruction phases, that are necessary to guarantee a considerable reduction of permanently stored data. As a result, contrary to the strategy adopted at present (LHC Run 2 phase), where most of the tracks in the ITS are the result of a prolongation of the TPC tracks, during Run 3 at least a partial reconstruction of high p_T tracks must be done synchronously to provide constraints for the TPC calibration. Therefore, the principal requirement for ITS tracking code is the speed. In order to meet such a goal, a new ITS tracking algorithm based on CA has been proposed in [2].

Highly parallel implementations of similar algorithms have been previously realised for other experiments, like PANDA, and for ALICE TPC detector. Since those implementations suggest that a noticeable speedup can be obtained with GPUs, it is interesting to evaluate if CUDA-enabled devices are able to cope with the 50 kHz Pb–Pb readout rate foreseen for ITS detector when the LHC Run 3 will start (after 2019).

In order to implement an effective parallelisation strategy, a preliminary, standalone CPU version of the code has been realised, removing all AliROOT specific dependencies from the pre-existing one. Moreover, a mathematical model for each algorithm step has been elaborated, in order to analyse both memory occupancy and computational complexity in a worst case scenario. Comparing theoretical results with code profiling data, an effective preallocation strategy has been developed, in order to both optimise performances and allow data structures to be allocated on a GPU device.

Table 4.1: Algorithm computing time measurements for context initialisation, tracklet and cell finding phases and for the whole process (in ms) running on a sample of 100 Pb–Pb central events with different amount of pile-up. Values in brackets are related to the serial implementation of the algorithm.

# Vertices		1	2	4	5
Context init	Min	5.3	9.1	19.3	24.9
	Mean	6.7	13.0	25.1	32.5
	Max	11.2	22.2	63.1	94.3
Tracklet finding	Min	1.8 (18.2)	6.3 (76.4)	27.0 (329.1)	44.2 (517.3)
	Mean	3.1 (39.4)	9.0 (139.0)	35.2 (549.6)	55.1 (810.0)
	Max	5.3 (70.6)	13.3 (269.3)	47.2 (1094.0)	70.5 (1641.0)
Cell finding	Min	2.0 (10.2)	6.5 (50.9)	35.4 (350.3)	60.6 (646.9)
	Mean	5.0 (28.5)	15.0 (134.5)	72.5 (862.4)	123.1 (1520.3)
	Max	7.9 (44.4)	20.9 (188.7)	104.4 (1213.6)	166.0 (2140.7)
Total	Min	9.6 (33.4)	23.4 (135.0)	84.2 (696.2)	133.3 (1186.2)
	Mean	16.7 (75.0)	40.2 (285.4)	139.5 (1437.7)	219.9 (2362.8)
	Max	27.0 (110.3)	59.9 (446.5)	220.7 (2326.1)	317.9 (3810.5)

Finally, the code has been adapted to the CUDA heterogeneous programming model, without sacrificing portability and maintainability. Table 4.1 reports execution time for the most computationally expensive algorithm phases. In addition, for comparison purposes, values related to the serial CPU implementation have been reported in brackets. Considering the mean values and the simulations with a single interaction vertex, a speedup of 12.7 times for the former and 5.7 times for the latter has been obtained, that lead to a speedup of 4.5 times on the whole process. Moreover, such performance gain becomes even more evident in simulations with multiple interaction vertices: the GPU implementation of the algorithm runs 10.7 times faster than the serial one when the processed event has 5 interaction vertices.

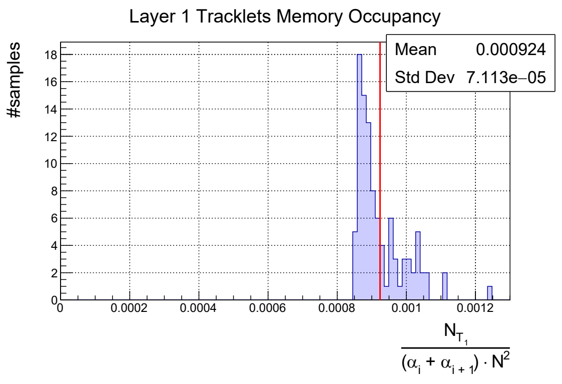
Despite the noticeable performance gain, there is still room for other performance improvements, in particular concerning the use of page locked memory for host allocation and the adoption of a more efficient memory layout for data structures, like AoS, in order to optimise access patterns. Furthermore, a parallelised version of the host code could perhaps lead to an additional performance gain.

Both CPU and CUDA versions of the code realised for this work will be migrated into the ALICE O^2 framework as soon as the OpenCL version, currently under development, is ready. Some vendor-agnostic APIs have been realised to simplify the OpenCL extension of at least some parts of the existing CUDA code, if the use of an OpenCL C++ wrapper is allowed.

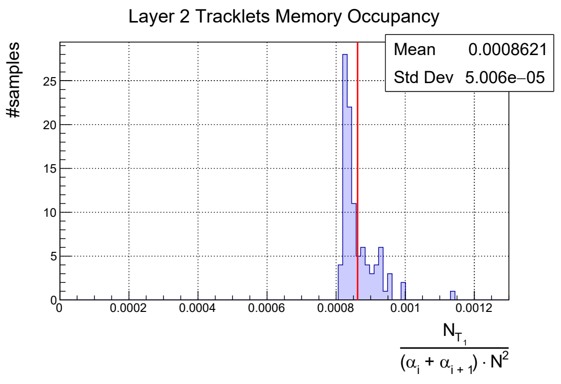
Appendix A

List of memory occupancy distributions

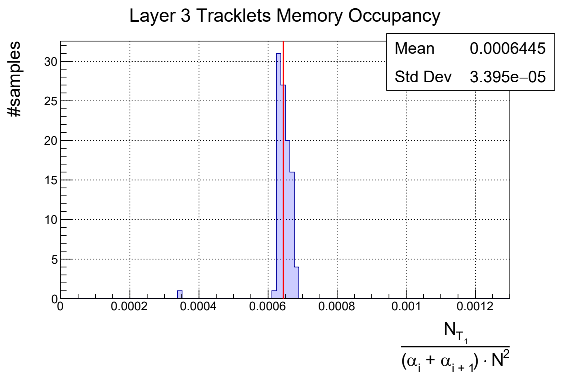
In this section are reported all memory occupancy benchmark results for the serial implementation of the CA tracker. Plots are related to simulations with 100 Pb–Pb central events without pile-up.



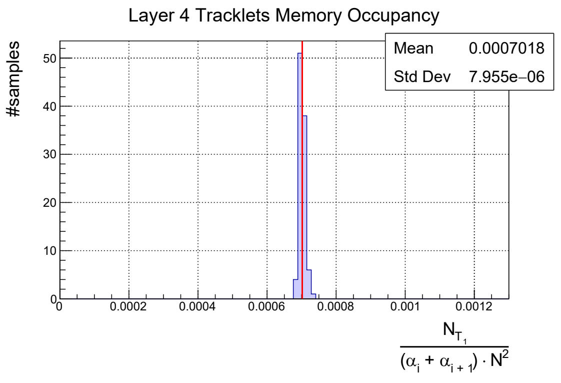
(a)



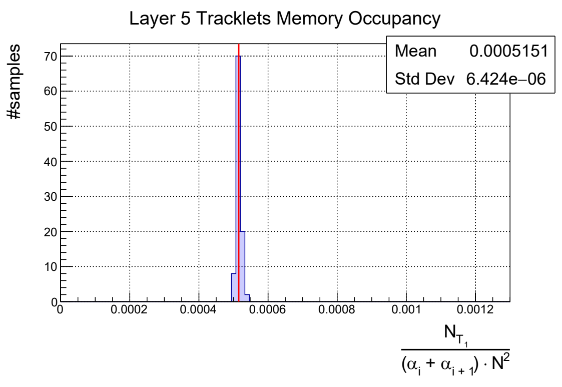
(b)



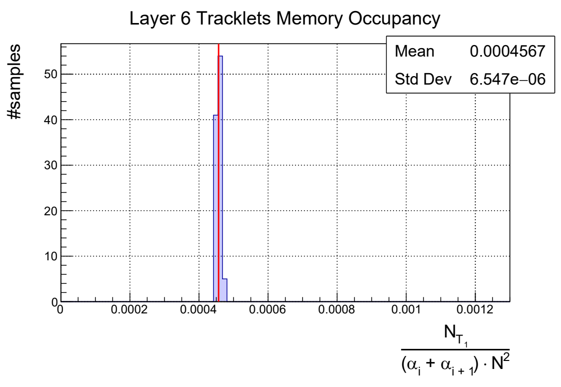
(c)



(d)

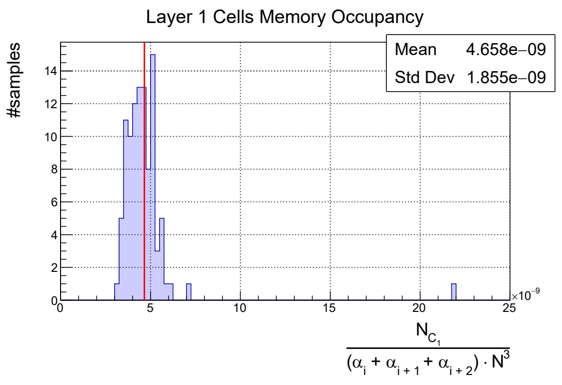


(e)

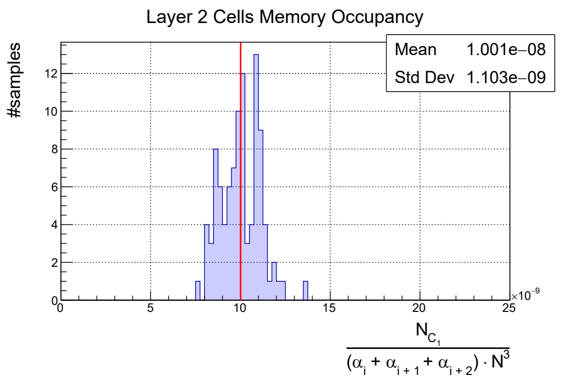


(f)

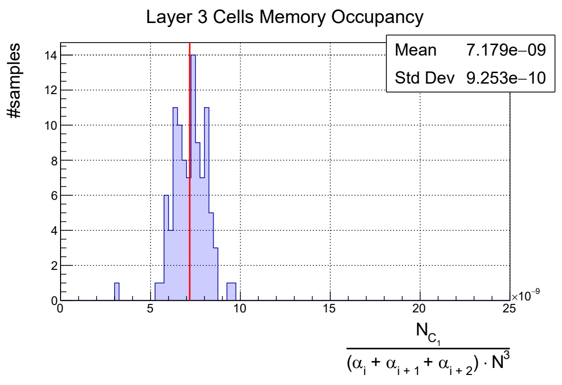
Figure A.1: Tracklet data structures memory occupancy for all 7 ITS layers. By comparing different plots, memory occupancy distribution appears to be more sparse for inner layers.



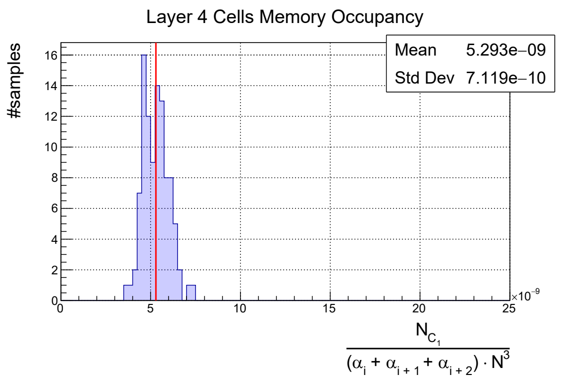
(a)



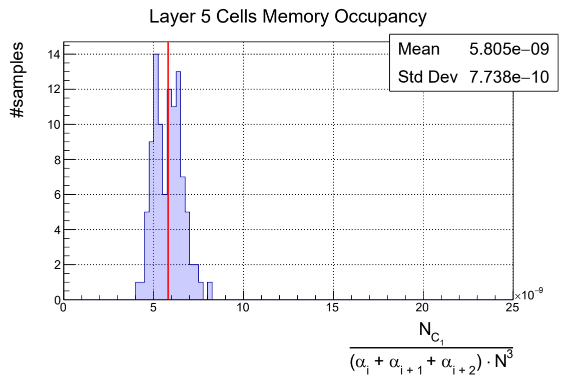
(b)



(c)



(d)



(e)

Figure A.2: Cell data structures memory occupancy for all 7 ITS layers. Again, memory occupancy distribution appears to be more sparse for inner ITS layers.

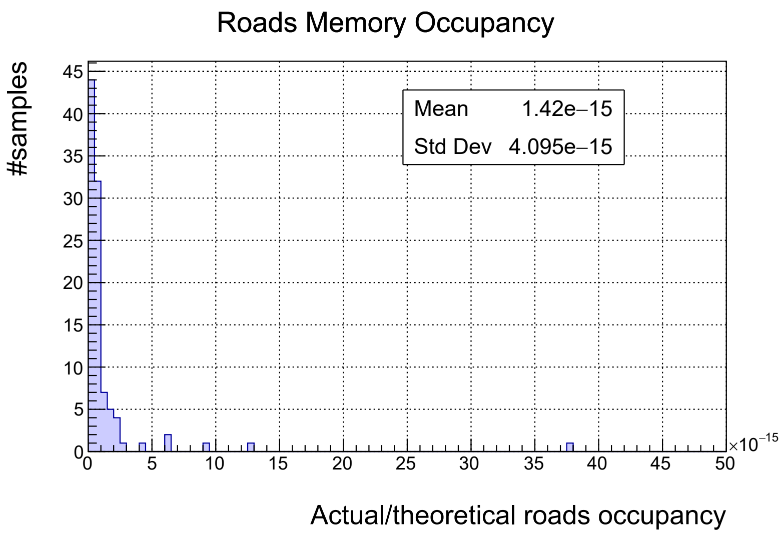


Figure A.3: Distribution of the actual to theoretical complete roads occupancy ratio for a sample of 100 central Pb–Pb events without pile-up, where the denominator is the number of all possible combinations among clusters across all 7 ITS Upgrade layers: $\prod_{i=1}^{7} \alpha_i N$

List of Acronyms

ACORDE	ALICE Cosmic Rays Detector
ALICE	A Large Ion Collider Experiment
AoS	Array of Structures
API	Application Programming Interface
ATLAS	A Toroidal LHC Apparatus
CA	Cellular Automaton
CMS	Compact Muon Solenoid
CPU	Central Processing Unit
CTF	Compressed Time Frame
CUDA	Compute Unified Device Architecture
CV	Coefficient of Variation
DAQ	Data Acquisition
EMCAL	Electromagnetic Calorimeter
EPN	Event Processing Node
FLP	First Level Processor
FMD	Forward Multiplicity Detectors
GEM	Gas Electron Multiplier
GPGPU	General Purpose GPU
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HIJING	Heavy-Ion Jet Interaction Generator
HL-LHC	LHC High Luminosity
HMPID	High Momentum Particle Identification
IDE	Integrated Development Environment

ISA	Instruction Set Architecture
ITS	Inner Tracking System
LEIR	Low Energy Ion Ring
LHC	Large Hadron Collider
LHCb	Large Hadron Collider beauty
LINAC	Linear Accelerator
LS2	Long Shutdown 2
MAPS	Monolithic Active Pixel Sensor
MC	Monte Carlo
MCH	Muon Tracking Chamber
MTR	Muon Trigger System
NVCC	Nvidia CUDA Compiler
NVVP	NVIDIA Visual Profiler
OO	object oriented
PANDA	Anti-proton Annihilation at Darmstadt
PHOS	Photon Spectrometer
PID	Particle Identification
PS	Proton Synchrotron
PSB	Proton Synchrotron Booster
PTX	Parallel Thread Execution
QGP	Quark-Gluon Plasma
RAII	Resource Acquisition Is Initialisation
RICH	Ring Imaging Cherenkov
RMS	Root Mean Square
SDD	Silicon Drift Detector
SFM	String-Fusion Model
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SNR	signal-to-noise ratio
SoA	Structure of Arrays
SoC	System on Chip
SPD	Silicon Pixel Detector
SPS	Super Proton Synchrotron

SSD	Silicon Strip Detector
STF	Sub-Time Frame
STL	Standard Template Library
TF	Time Frame
TOF	Time Of Flight
TPC	Time Projection Chamber
TRD	Transition Radiation Detector
VMC	Virtual Monte Carlo
WLCG	Worlwide LHC Computing Grid

Bibliography

[1] G. Cerati, P. Elmer, S. Lantz, I. MacNeill, K. McDermott, D. Riley, M. Tadel, P. Wittich, F. Würthwein, and A. Yagil, “Traditional tracking with kalman filter on parallel architectures,” *Journal of Physics: Conference Series*, vol. 608, no. 1, 2015.

[2] M. Puccio, *Study of the production of nuclei and anti-nuclei at the LHC with the ALICE experiment*. PhD thesis, Università degli studi di Torino, Dipartimento di Fisica, May 2017.

[3] C. De Melis, “The cern accelerator complex. complexe des accélérateurs du cern.” https://cds.cern.ch/record/2197559, July 2016. Accessed: 2017-07-23.

[4] “The accelerator complex.” http://cds.cern.ch/record/1997193, Jan. 2012. Accessed: 2017-07-23.

[5] L. Betev and P. Chochula, “Definition of the alice coordinate system and basic rules for sub-detector components numbering.” https://edms.cern.ch/document/406391, 2003. Accessed: 2017-07-24.

[6] K. Aamodt *et al.*, “The alice experiment at the cern lhc,” *JINST*, 2008.

[7] F. Carminati *et al.*, “Alice: Physics performance report, volume i,” Tech. Rep. CERN-LHCC-2003-049, CERN, Geneva, Nov. 2003.

[8] B. Abelev *et al.*, “Performance of the alice experiment at the cern lhc,” *Int. J. Mod. Phys. A*, vol. 29, Feb. 2014.

[9] Y. Belikov, M. Ivanov, K. Safarik, and J. Bracinik, “Tpc tracking and particle identification in high density environment,” *eConf*, vol. C0303241, p. TULT011, 2003.

[10] C. Collaboration, “Measurement of tracking efficiency,” Tech. Rep. CMS-PAS-TRK-10-002, CERN, Geneva, 2010.

[11] M. Bernardini and K. Foraz, “Long shutdown 2 @ lhc,” in *Chamonix 2014: LHC Performance Workshop*, pp. 290–293, 2015.

[12] G. Apollinari *et al.*, *High-Luminosity Large Hadron Collider (HL-LHC): Preliminary Design Report*. CERN Yellow Reports: Monographs, Geneva: CERN, 2015.

[13] L. Musa, “Upgrade of the inner tracking system conceptual design report. conceptual design report for the upgrade of the alice its,” Tech. Rep. CERN-LHCC-2012-013. LHCC-P-005, CERN, Geneva, Aug. 2012.

[14] https://github.com/alisw/AliRoot. Accessed: 2017-07-30.

[15] http://root.cern.ch. Accessed: 2017-07-30.

[16] X.-N. Wang and M. Gyulassy, "Hijing: A monte carlo model for multiple jet production in p p, p a and a a collisions," *Phys. Rev.*, vol. D44, pp. 3501–3516, 199.

[17] http://ntc0.lbl.gov/~xnwang/hijing/. Accessed: 2017-07-30.

[18] J. Ranft, "Dual parton model at cosmic ray energies," *Phys. Rev. D*, vol. 51, pp. 64–84, Jan. 1995.

[19] N. Armesto and C. Pajares, "Central rapidity densities of charged particles at rhic and lhc," *Int. J. Mod. Phys.*, vol. A15, pp. 2019–2052, 2000.

[20] I. Hrivnacova *et al.*, "The virtual monte carlo," *CoRR*, vol. cs.SE/0306005, 2003.

[21] R. Brun, F. Bruyant, M. Maire, A. C. McPherson, and P. Zanarini, *GEANT 3: user's guide Geant 3.10, Geant 3.11; rev. version*. Geneva: CERN, 1987.

[22] S. Agostinelli *et al.*, "Geant4: A simulation toolkit," *Nucl. Instrum. Meth.*, vol. A506, pp. 250–303, 2003.

[23] A. Fassò *et al.*, "The fluka code: present applications and future developments," Tech. Rep. physics/0306162, Milano Univ. INFN, Milano, June 2003.

[24] P. Buncic, M. Krzewicki, and P. Vande Vyvre, "Technical design report for the upgrade of the online-offline computing system," Tech. Rep. CERN-LHCC-2015-006. ALICE-TDR-019, CERN, Apr. 2015.

[25] V. Pieterse and P. E. B. eds., "Dictionary of algorithms and data structures." https://www.nist.gov/dads/. Accessed: 2017-07-08.

[26] https://github.com/GlassOfWhiskey/tracking-itsu. Accessed: 2017-08-17.

[27] https://github.com/AliceO2Group/CodingGuidelines. Accessed: 2017-08-17.

[28] https://cmake.org/. Accessed: 2017-08-18.

[29] S. Meyers, *Effective modern C++: 42 specific ways to improve your use of C++11 and C++14*. Sebastopol, CA: O'Reilly, 2015.

[30] S. Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Indianapolis, IN: Addison-Wesley, 2001.

[31] J. Weidendorfer, "Sequential performance analysis with callgrind and kcachegrind," in *Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pp. 93–113, 2008.

[32] A. Herten, *GPU-based Online Track Reconstruction for PANDA and Application to the Analysis of $D \rightarrow K\pi\pi$*. Dr., Ruhr-Universität Bochum, Bochum, 2015.

[33] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2017. Version 8.0.

[34] NVIDIA Corporation, "NVIDIA CUDA C best practices guide," 2017. Version 8.0.

[35] A. Corana, "Architectural evolution of nvidia gpus for high-performance computing," tech. rep., IEIIT-CNR, Feb. 2015.

[36] NVIDIA Corporation, "NVIDIA CUDA C compiler driver nvcc," 2017. Version 8.0.

[37] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, pp. 831–838, Oct. 1980.

[38] A. V. Adinetz, "Cuda pro tip: Optimized filtering with warp-aggregated atomics." https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/. Accessed: 2017-09-10.

[39] B. Merry, "A performance comparison of sort and scan libraries for gpus," *CoRR*, vol. abs/1601.03144, 2016.

[40] https://nvlabs.github.io/cub/. Accessed: 2017-09-12.

[41] J. Siegel, J. Ributzka, and X. Li, "Cuda memory optimizations for large data-structures in the gravit simulator," in *2009 International Conference on Parallel Processing Workshops*, pp. 174–181, Sept 2009.