Master Degree Thesis

Optimization of a high-performance tracking algorithm on GPUs for the Inner Tracking System of the ALICE experiment

Candidate
Franco Dessena

Supervisor
Prof. Stefania Bufalino
Co-supervisor
Prof. Massimo Masera

Dipartimento di Automatica e Informatica
Politecnico di Torino
Luglio 2018

Abstract

The subject of this thesis is the design and the optimization of a fast tracking algorithm that will be used for the innermost detector of the ALICE (A Large Ion Collider Experiment) experiment installed at the CERN Large Hadron Collider (LHC) in Geneva. In 2019, during the Long Shutdown 2 (LS2), the present Inner Tracking System (ITS) of ALICE will be replaced by a completely new detector, based on thin monolithic silicon pixel chips. This new detector has been designed to fulfill the Run 3 physics requirements, which consist first of all in an incremented collision rate that can reach 50 kHz of frequency for collisions between lead ions. Along with a hardware upgrade, also the software must be upgraded in order to match the new required performance.

The 7 layers, which compose the new detector, will be hit by the particles generated from the collision between proton-proton (p-p), proton-lead (p-Pb) or lead-lead (Pb-Pb). The points where the particles hit the pixel of each detector are called *clusters*. The tracking algorithm uses the clusters to reconstruct the trajectory of a particle and find the point where the collision happened. This point is called *interaction vertex*.

Currently, two different versions of the tracking algorithm have been developed: a first version is a serial implementation that can run on CPUs only and a second one based on the CUDA framework that can be executed only on NVIDIA GPUs.

The purpose of my thesis work is to develop a third version of the tracking algorithm, based on the OpenCL framework. OpenCL allows the program to run on different kind of devices such as CPUs, GPUs, FPGAs and others. To this purpose I designed and implemented three different versions of OpenCL porting, the first one based on the schema developed for the CUDA version, the others two based on different schema in order to better fit the algorithm to the features offered by OpenCL and to improve the overall performance.

The thesis is organized as follows: the first chapter is dedicated to the description of the features of both the present ITS and the upgraded detector. The second chapter is devoted to the description of the current tracking algorithm and of the performances achieved for the serial (on CPU) and parallel versions (on NVIDIA GPU). The third chapter describes the main features of the OpenCL framework and Intel Interception Layer, a tool adopted for the analysis of the execution flow of OpenCL program. The fourth chapter is focused on the OpenCL porting with the descriptions of three different versions I developed highlighting advantages, disadvantages and describing in detail the achieved performances to make a comparison between them, the serial and the CUDA parallel versions of the algorithm.

Contents

List of Figures

List of Tables

Chapter 1

ALICE experiment

A Large Ion Collider Experiment (ALICE) is a general-purpose detector installed at the CERN (Conseil Européen pour la Recherche Nucléaire) Large Hadron Collider (LHC) and built to study the strongly interacting matter produced at high energy densities and temperature, the so called QuarkGluon Plasm (QGP), by using the Pb-Pb collisions delivered by the LHC. Along with A Toroidal LHC Apparatus (ATLAS), Compact Muon Solenoid (CMS) and Large Hadron Collider beauty (LHCb), ALICE is one of the biggest experiment conducted at the CERN laboratory. Currently, more than 1500 physicists, engineers and technicians from 154 physics institutes from 37 countries work in the ALICE experiment.

1.1 Introduction

LHC is the world's largest and most powerful particle accelerator and it consists of a 27 km underground ring composed of superconducting magnets (kept at a temperature of -271.3 °C, using liquid helium, in order to reduce electric resistance or energy losses) capable to accelerate particles up to 99,9999991% of light speed with an energy of 13 TeV. The four experiments mentioned before are installed in four different interaction points along LHC, as shown in figure 1.1. The beams of proton (p) and lead ions (Pb) follow two different roads to reach LHC. Using an electric field, hydrogen particles loose their electrons, the derived protons reach LHC going through the Linear Accelerator (LINAC) 2, the Proton Synchrotron Booster (PSB), the Proton Synchrotron (PS) and finally through the Super Proton Synchrotron (SPS); at each step, the energy of the proton beam is increased up to 450 GeV when it reaches the LHC ring. The beam of lead ions starts from a container with vaporized lead inside and, through LINAC 3, Low Energy Ion Ring (LEIR), PS and SPS, it reaches the LHC. Inside the LHC, two beams run in opposite ways with a maximum energy of 6.5 TeV and then they are forced to collide where

the detectors of the experiments are located.

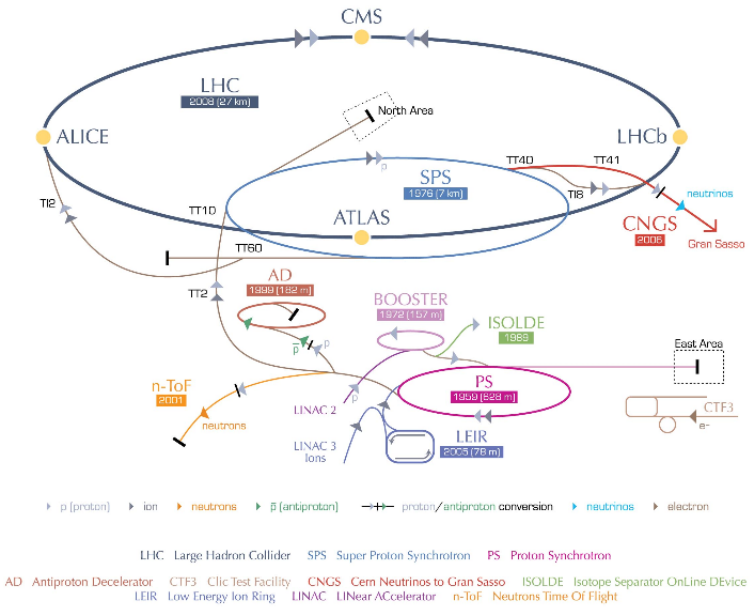

Figure 1.1: The main accelerators complex used to speed-up particles and to inject them into the LHC. [1]

1.2 ALICE design and layout

According to [2], the ALICE design was driven by the physics requirements, in particular by the extremely high particle multiplicity ($dN/d\eta$) expected in the heavy-ion collisions: the detector is optimized for a value of $dN/d\eta$=4000 but it is tested with Monte Carlo simulation up to twice this amount of particles; inside the LHC the beam interaction rate is quite low (about 10 kHz with Pb-Pb) and the radiation doses are moderate ($<$ 3000 Gy) and this allows using slow detectors with high-granularity, like the Time Projection Chamber (TPC) and the Silicon Strip Detector (SSD).

The coordinate systems of the ALICE detector are defined according to the LHC rules adopted by all the LHC experiments[3]. It is a right-handed orthogonal Cartesian system with point of origin $\{x,y,z\} = 0$ at the beams Interaction Point (IP) and the coordinates are defined as follow (according to figure 1.2):

- x axis: perpendicular to the beam direction and aligned with the local horizontal; positive x is from coordinates origin to the accelerator centre;

- y axis: perpendicular to the x axis; positive y is from the point of origin upward;
- z axis: parallel to the beam direction, points to the ATLAS experiment;
- azimuthal angle ϕ: observing from a positive z position, increases counter-clockwise from x (ϕ=0) to y (ϕ=$\pi/2$);
- polar angle θ: increases from positive z (θ=0) to the (x,y) plane (θ=$\pi/2$) to negative z (θ=ϕ).

The conversion between Cartesian and spherical coordinates is done as follow:

$$\begin{aligned} x &= \sin\theta\cos\phi \\ y &= \sin\theta\sin\phi \\ x &= r\cos\theta \end{aligned} \tag{1.1}$$

$$\begin{aligned} \rho &= \sqrt{x^2+y^2+z^2} \\ \theta &= \arccos z/\rho \\ \phi &= \arctan y/x \end{aligned} \tag{1.2}$$

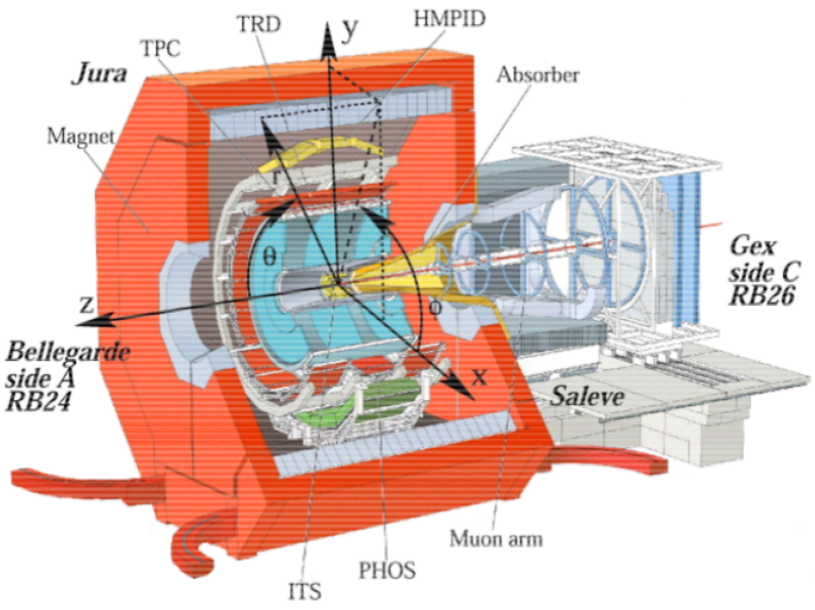

Figure 1.2: ALICE coordinate system: some labels (Bellagarde, Gex and Jura) are added to make the reading more clear. [3]

The overall ALICE dimensions are $16 \times 16 \times 26$ m^3 with a total weight of approximately 10000 t and it consists of a central barrel part (which measures hadrons, electrons, and photons) and a forward muon spectrometer; the central part is installed inside a large solenoid magnet and it covers polar angles from 45 ° to 135 °. Starting from the innermost detector the barrel contains[2]:

- ITS: an Inner Tracking System, composed by six layers of high-resolution silicon pixel (SPD), drift (SDD), and strip (SSD) detectors;
- TPC: a cylindrical Time-Projection Chamber;
- three particle identification arrays of Time of Flight (TOF);
- High Momentum Particle Identification (HMPID) and Transition Radiation Detector (TRD) detectors;
- two electromagnetic calorimeters (PHOS and EMCa);

In the following I will describe only the ITS detector because it is pivotal to better understand the different steps followed during my thesis work for the design and the optimization of the algorithm that will be used to reconstruct the trajectories of the particles traversing this detector and produced in the Pb-Pb collisions delivered by the LHC.

1.3 The ALICE Inner Tracking System

1.3.1 Layout of the present ITS

To better understand the reasons behind the upgrade of the ALICE ITS, a brief description of the ITS operating at present in the experiment will be given in the following. The present ITS is the innermost detector of ALICE and it is designed and built to determine the position of the primary vertex with a resolution better than 100 μm, reconstruct the secondary vertex of heavy flavour and strange particle decays, identify and track low-momentum particles (<200 MeV/c), improve momentum and angle resolution for particles reconstructed by the TPC and to reconstruct particles traversing dead regions of the TPC. The ITS surrounds, and supports, the beam pipe which is a 800 μm-thick beryllium cylinder of 6 cm diameter. As shown in figure 1.3 and in table 1.1, the detector is composed by six cylindrical layers, coaxial with beam pipe, with radius going from 3.9 cm (the innermost) to 43.0 cm (the outermost). Due to the high particle density (about 50 particles per cm^2 for the inner layer), the first two layers consist of Silicon Pixel Detector (SPD) and the following two of Silicon Drift Detector (SDD); the last two layers are equipped with

double-sided Silicon Strip Detector (SSD) because the expected particle density is reduced to less than 1 particle per cm^2. A specific cooling system for the outermost layer has been designed to match the requirements imposed by the TPC in terms of temperature stability and uniformity. In order to limit the noise over the signal ration, the silicon detectors used to measure ionisation density must have a thickness grater than 300μm. The ITS is able to detect simultaneously more than 15000 tracks, managing several millions of cells in each layer [2].

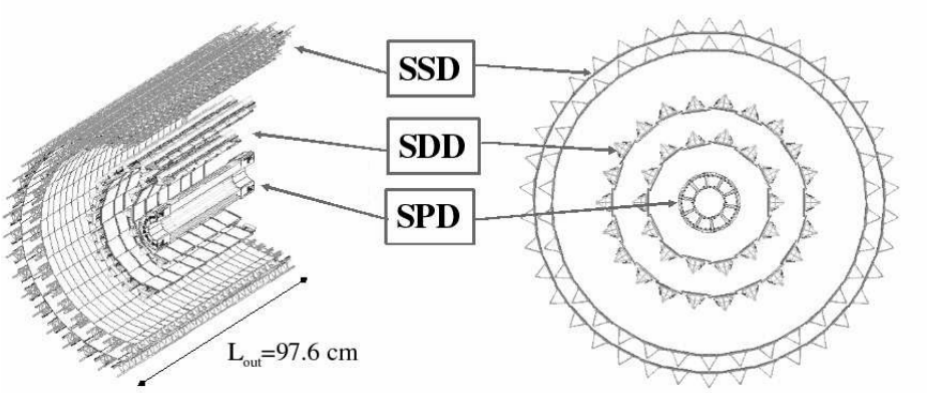

Figure 1.3: Layout of the present ALICE Inner Tracking System. [2]

Table 1.1: Dimensions of the six layers that compose the ITS. [2]

Layer	Type	r(mm)	±z(mm)	Area (m^2)
1	pixel	39	141	0.07
2	pixel	76	141	0.14
3	drift	150	222	0.42
4	drift	239	297	0.89
5	strip	380	431	2.20
6	strip	430	489	2.80
total				6.28

1.3.2 Upgraded ITS

After more than ten years of operation, the LHC is concluding a collisions campaign called Run 2 and at the beginning of 2019 a long pause named as Long Shutdown 2 (LS2) is scheduled and the accelerator complex will be stopped for 18 months; during this period the following changes/upgrades are planned for the ALICE apparatus [4]:

- a new beam pipe with smaller diameter (20 mm);
- a new high-resolution, low material ITS;
- a new TPC;
- upgrade of read-out electronics of the Transition Radiation Detector (TRD), the Time Of Flight (TOF) detector and the muon spectrometer;
- upgrade for the forward trigger detector;
- upgrade on the online systems and offline reconstruction and analysis framework;

Once the aforementioned upgrades will be completed, according to [5] the upgraded ITS should be able to achieve the following goals:

- improve spacial resolution by a factor of 3 on the plane r-ϕ at $p_T \approx 500$ MeV/c thanks to:
  - first layer closer to beam: from 39 mm to 22 mm
  - reduction of material budget for inner layers: X/X_0 */layer* from ≈1.14% to ≈0.3%
  - reduction of pixel dimension from 50μm x 425μm to 25μm x 25μm
- high tracking efficiency and resolution on transverse momentum $p_{\mathrm{T}} = p \sin\theta$ in stand-alone mode by :
  - increasing the granularity: 7 layers instead of 6 layers (with dimensions specified in 1.2)
  - increasing the radial extension, from 39-430 mm to 22-430 mm
- speed up of the reading of the interaction rate (currently limited to 1 kHz):
  - for Pb-Pb interaction: >50 kHz
  - for p-p interaction: >400 kHz
- quick access to ITS systems for maintenance: should be possible replace damaged parts during annual shutdown. For the present ITS this is not feasible.

Figure 1.4 shows the layout of ITS after the upgrade operations.

The result of Monte Carlo simulations, shown in figure 1.5, highlights how the efficiency in tracking the particles traversing the detector will be improved after the upgrade when the ITS is used in standalone mode for Pb-Pb collisions.

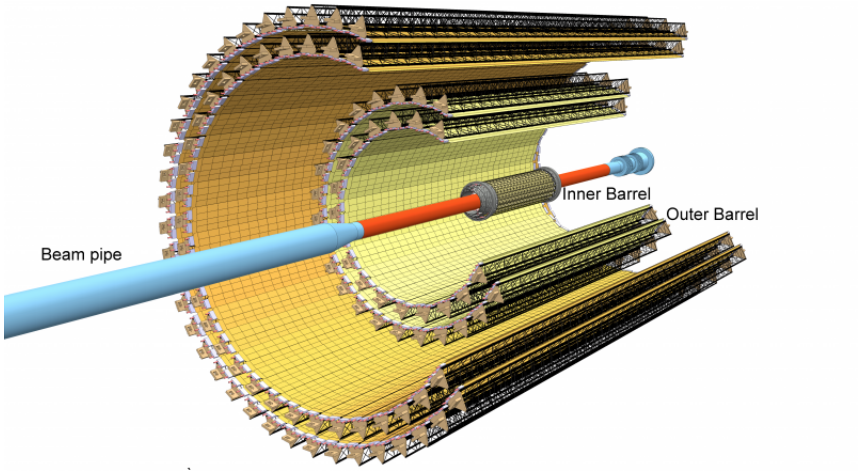

Figure 1.4: Schematic layout of the upgraded ITS thet will be installed in 2019 to be operation during the Run 3. The red cylinder represents the beam pipe were the collisions happen.

Table 1.2: Dimensions of the seven layers after ITS upgrade. [2]

Layer	r(mm)	±z(mm)
1	22	112
2	28	121
3	36	134
4	200	390
5	220	418
6	410	712
7	430	743

1.4 Event reconstruction

According to [6], the tracking procedures are performed in the central barrel of ALICE, in particular in the ITS, TPC and TRD detectors. The figure 1.6 shows the order followed to complete all the steps executed during the tracking phase. Starting form raw data collected via Monte Carlo (MC), the first step, called clusterization, reconstructs all the information about particles that have crossed subdetectors; in particular the space-time coordinates regarding the point where particles hit the active surface of subdetectors, which is called cluster; in addition information about the time of flight, the momentum and energy loss are collected in order to allow the identification of each particle emitted in the collision.
The second step is the first estimation of the primary vertex position using only the first two layers of the ITS (an accurate estimation can be obtained only using

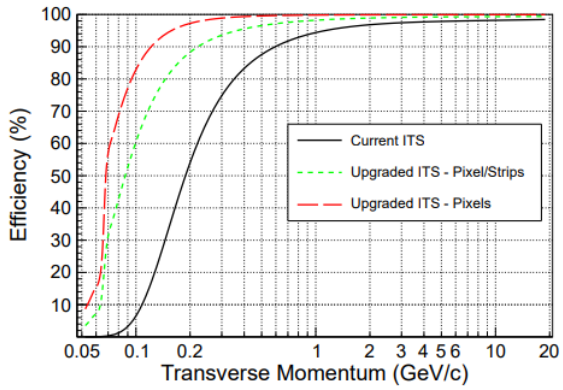

Figure 1.5: Tracking efficiency of charged pions for the current (black line) and upgraded ITS (red and green lines) in the ITS stand-alone tracking modes. The efficiency is defined as the number of correct refitted tracks over the number of total possible tracks. A possible track is a particle which is formed by, at least, one cluster on each layer of the ITS. [4]

full tracks, but this first estimation is necessary to speedup the following phases); the algorithm finds all the possible tracklets, pair of clusters (from two consecutive layers) which respect geometrical constraints in terms of azimuthal angle and maximum distance on z direction; the primary vertex is then localized where the largest number of tracklets converge (to estimate the position, at least two tracklets are needed); if more than one primary vertex are reconstructed it means that we are in the case of pile-up.

At this point, the actual reconstruction phase can start; at first the track seeds are build in the outer part of the TPC. A track seed is composed by two clusters and the primary vertex (then this operation will be done using three cluster only); each seed is propagated inward and, using a Kalman Filter[7], a compatible cluster is found and the track information are updated; all the tracks with more than 20 clusters (and a maximum of 159) and that miss no more than half of the expected clusters are accepted and propagated to the inner layers of the TPC.

The preliminary information, about momentum and energy loss of the clusters, attached to each track, can be used for a first hypothesis about the specie of the particle being tracked: this first hypothesis is especially useful when the track is propagated inward from the TPC to the ITS. Starting from the outermost layer of ITS an algorithm similar to the one used for the TPC and based on a Kalman Filter is implemented; at each step to go to the inner layers, a compatible cluster is added to each track, the track parameters are updated and it is used as seed for the successive layer; if no cluster can be added to the track, a penalty factor to the χ^2 is added; at the end of this phase, a tree of hypothetical ITS tracks is built for each TPC track:

only the high quality track (the one with the minimum χ^2) of each tree is added to the reconstructed event, forming an ITS+TPC track. A further ITS standalone algorithm is necessary to reconstruct tracks using the clusters that do not form ITS+TPC tracks, for instance low momentum particles that do not reach TPC; the two innermost ITS layers and primary vertex are used to reconstruct helicoidal seeds. Later, seeds are propagated towards outer layers and clusters satisfying geometrical selections are added to them. This procedure is repeated few times with more relaxed constraints in order to improve the efficiency at low p_T. In parallel by using the ITS standalone tracking algorithm, the backward refit of the ITS+TPC track can be done adopting the Kalman Filter. During this phase the integrated track length and the expected time for different particle species are computed in order to execute a particle identification via time of flight measurement. When the refit is completed, a successive extrapolation step is necessary to compare each track with TRD tracks and TOF tracks; a similar procedure is adopted to match the track with signals on the ElectroMagnetic CALorimeter (EMCAL), the PHOton Spectrometer (PHOS) and the High Momentum Particle IDentification (HMPID) detector.
The final reconstructed tracks are used to find the primary vertex with a precision higher than the first estimation; at the end, vertices related to photon conversions and particle decays are used to locate the secondary vertices.

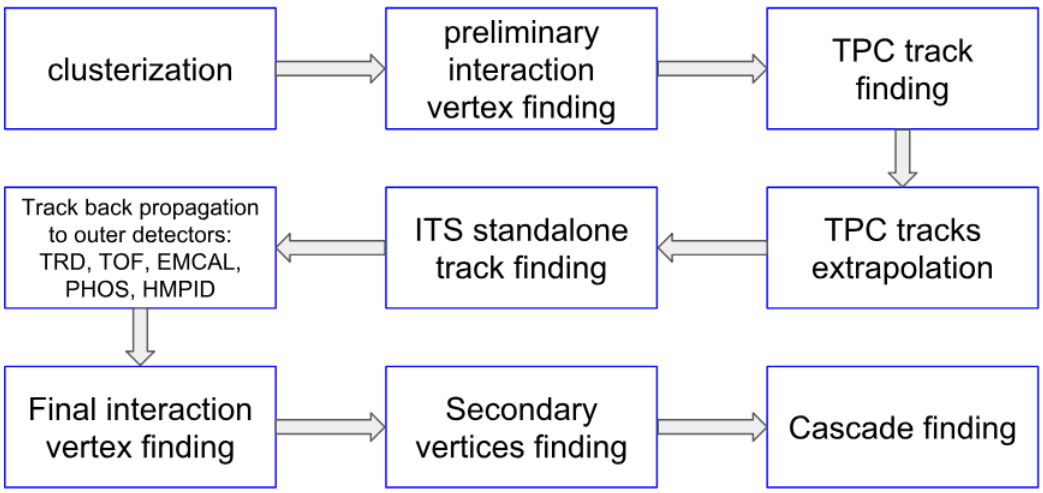

Figure 1.6: Flow of operations performed by the reconstruction algorithm used in the ALICE experiment.

1.5 AliROOT framework

The software framework used to reconstruct and analyze the real data collected by the ALICE experiment and to simulate and reconstruct Monte Carlo events is AliROOT and it is based on ROOT [8] .

ROOT is a scientific software framework which provide a set of ObjectOriented C++ frameworks with many advanced methods needed to handle and analyse large amounts of data in a very efficient way. The user can exploit the functionality of ROOT through a Graphic User Interface (GUI), command line or MACROs (C++ script).

AliROOT must be able to process both data collected in real experiments and simulations of heavy-ion collision events at the LHC energy, executed using external Monte Carlo tools like Heavy-Ion Jet Interaction Generator (HIJING) and String Fusion Mode (SFM). The last step is the reconstruction of tracks using the track reconstruction algorithm described at 1.4 and compare them with the simulated ones. In this way it is possible to analyze the correctness and performance of tracking algorithm used.

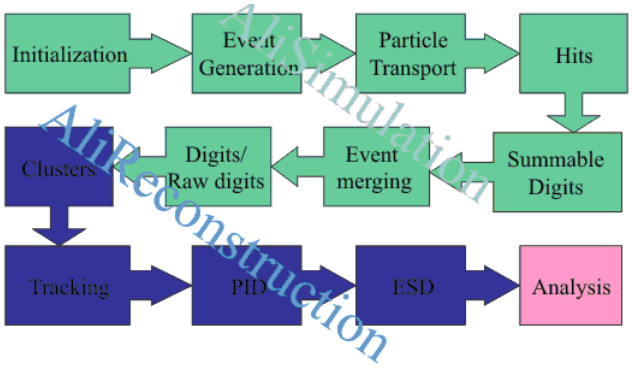

Figure 1.7: Flow of AliROOT operations over simulated events. Green blocks correspond to simulation phase, blue blocks correspond to reconstruction algorithm phase; the last pink block represent the comparison between simulated and reconstructed tracks. [9]

1.6 O^2 project

At the end of the LS2 a new campaign of data takings will start and this phase is called LHC Run 3. During the Run 3 the interaction rate for Pb-Pb events will reach up to 50 kHz[4] generating a data throughput from the detector greater than 1TB/s[10]. The online-offline (O^2) facility will be dedicated to the handling

of this amount of data generated by the ALICE detectors. O^2[11] is designed to be a high-throughput system which will include heterogeneous computing platforms. To ensure fast processing, some O^2 nodes will be equipped with specific hardware accelerators such as Graphic Processing Unit (GPU) and Field Programmable Gate Array (FPGA) and specific software will be developed taking into account the hardware used for each specific computing need. In this section I will summarize the main phase of the O^2 flow.

Data produced by the detectors are transferred to the facility through optical links

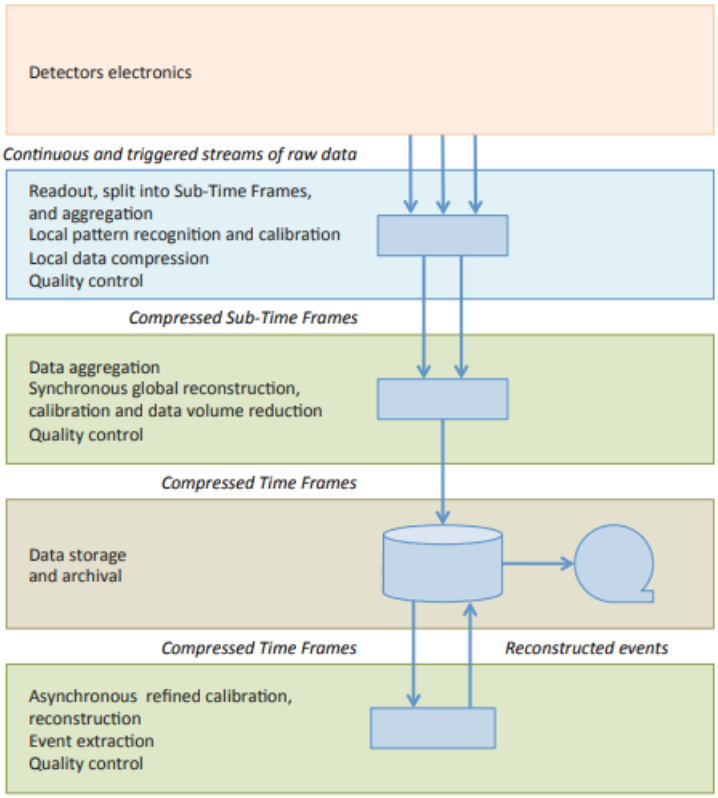

Figure 1.8: Functional flow of the O^2 computing system. [11]

in the form of data streams. Each First Layer Processor (FLP) collects data at 3.2 GB/s from up to 48 read-out links for a total of 1.1 TB/s over approximately 8300 read-out links. Data streams are aggregate and split in chunks of 20 ms called Time Frame (TF) with a compression factor of 2.5 [10]; TFs can be of two different kind: Sub-Time Frame (STF) that contains RAW data from a single First Level Processor (FLP) or Compressed Time Frame (CTF) which contains processed RAW data of all active detectors (after the creation, CTF chunks are read only). A time size of 20 ms means a frequency of 50 Hz and a size of 10 GB before compression for each

frame with a 0.5% of data loss at boundaries.

At this point, the STFs are dispatched to the Event Processing Nodes (EPN) for aggregation: the STFs related to the same time period are managed by the same EPN. The load over each EPN is balanced at run time using information provided by the same EPNs (as similar mechanism is performed during the previous step using information provided by FLP, in order to minimise data transport and to make results available as early as possible). Each EPN reconstructs data for each detector and reduces the data by an average factor of 8. The compressed TFs are stored in the O^2 using permanent archive as storage. Each EPN produces approximately 60MB/s of data for a total throughput of 90 GB/s directed to storage.

At least, an asynchronous data processing step will be done using resources from the Worldwide LHC Computing Grid (WLCG) before permanently store reconstructed events. In order to meet the ALICE Run 3 and O^2 constraints, a new Cellular Automata (CA) algorithm for tracking reconstruction has been proposed by [6] and it will be described in chapter 2.

Chapter 2

The Tracking Algorithm

In this chapter I will illustrate the tracking algorithm described in [6] and from which I started to carry out my thesis project to perform the optimization of the code on a parallel architecture. The goal of that tracking algorithm is to reconstruct all possible tracks generated in the LHC heavy-ion collisions using the data coming from the previous clusterization step. The algorithm can be split into the following phases:

- Indexing
- Tracklet computing
- Cell computing
- Finding of neighbourhood cells
- Track reconstruction
- Fitting phase

2.1 Algorithm flow

2.1.1 Indexing phase

This step is useful to speed-up the following phases and especially the step in which the algorithm needs to compute the tracklets; each clusterized input c is organised in an index matrix n_z x n_ϕ using a bin index calculated with z_c coordinate and azimuth angle ϕ_{c}

$$index_{\mathrm{c}} = \left\{ \left\lfloor \frac{z_{\mathrm{c}} - z_{\mathrm{min_i}}}{z_{\mathrm{bin_i}}} \right\rfloor, \left\lfloor \frac{\phi_{\mathrm{c}}}{\phi_{\mathrm{bin}}} \right\rfloor \right\} \tag{2.1}$$

The quantity z_{bin_i} reported in 2.1 represents the ratio between the extension along the z coordinate of the i-layer and the size of index table (2.3), while ϕ_{bin} is determined as shown in 2.2.

$$\phi_{\text{bin}} = \frac{2\pi}{n_\phi} \tag{2.2}$$

$$z_{\text{bin}} = \frac{z_{\text{max}_\text{i}} - z_{\text{min}_\text{i}}}{n_\text{z}} \tag{2.3}$$

At this point of the execution flow, the input data are already sorted in base of ϕ_{bin} and z_{bin}, so the computational complexity of indexing cluster is linear to number of input clusters N:

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

2.1.2 Tracklet computing phase

The goal of this phase is to find all possible tracklets; a tracklet is a connection between 2 clusters of adjacent layers. Therefore this computational step is executed on pairs of subsequent layers. As shown in fig. 2.1, for each cluster of the first layer is possible to find its region of interest, which is defined as a 2 dimensional window that contains all possible compatible clusters. Using the index table described above this operation can be speeded-up. At this point we obtain a set of pairs of clusters

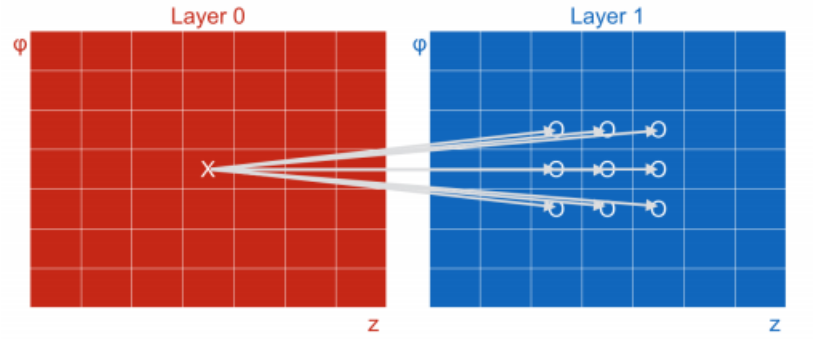

Figure 2.1: Example of calculation of the region of interest for a given cluster: the use of index table speed-up this operation thanks to sorting criterion (ϕ and z) used to arrange cluster. Red and blue squares represent a portion of index table of the first two layers.

(c_i,c_{i+1}) but not all of them can be considered correct tracklet (fig.2.2): a filtering operation is needed to select only useful ones in order to limit workload of following phases as explained later. In particular the possible tracklets are filtered using $\Delta\phi$

and Δz calculated as indicated in 2.5 and 2.6 considering a cluster of i-layer defined as $c_i = (z_i, r_i, \phi_i)$

$$\Delta\phi = |\phi_{\mathrm{i}} - \phi_{\mathrm{i+1}}| \tag{2.5}$$

$$\Delta z = |\tan\lambda_{\mathrm{i}} \cdot (r_{\mathrm{i+1}} - r_{\mathrm{i}}) - (z_{\mathrm{i}} - z_{\mathrm{i+1}})| \tag{2.6}$$

The quantity λ_i in 2.6 can be calculated with respect to the z coordinate of the interaction vertex z_V as shown in 2.7

$$\lambda_{\mathrm{i}} = \frac{z_{\mathrm{i}} - z_{\mathrm{v}}}{r_{\mathrm{i}}} \tag{2.7}$$

In order to consider a pair of clusters as valid tracklet the following two criteria must be verified:

- the difference between azimuthal angle of the two clusters ($\Delta\phi$) must be less than a threshold equal for all layers.

$$\Delta\phi < \Delta\phi^{\mathrm{MAX}} \tag{2.8}$$

- Δz, the difference between the propagation of interaction vertex passing for first cluster on z axis and the second cluster must be less than a threshold dependent on the layer

$$\Delta z < \Delta z_{\mathrm{i}}{}^{\mathrm{MAX}} \tag{2.9}$$

The values of the threshold can be found in 2.1.

Table 2.1: Threshold values used inside tracklet finding phase to consider a couple of clusters as a valid tracklet.

Layer	1	2	3	4	5	6
Δz^{MAX}	0.1	0.1	0.3	0.3	0.3	0.3
$\Delta\phi^{\mathrm{MAX}}$	0.3					

The first part of this phase, the selection of bin for each cluster is, at worst, a simple iteration over all the bins of the index table, so the computational complexity depends only on the index table dimension (which can be considered constant):

$$T(N) = \sum_{i=0}^{6} n_{\mathrm{x}} \cdot n_{\phi} = O(1) \tag{2.10}$$

The filtering part depends on the number of cluster on each layers; we must consider all the possible pairs and assuming a filtering factor K_i less than 1 and $\mathrm{N_i}$ as the number of clusters on the layer i, we obtain:

$$T(N) = \sum_{i=0}^{5} N_{\mathrm{i}} \cdot K_{\mathrm{i}} \cdot N_{\mathrm{i+1}} = \sum_{i=0}^{5} K_{\mathrm{i}} \cdot N^2 = O(N^2) \tag{2.11}$$

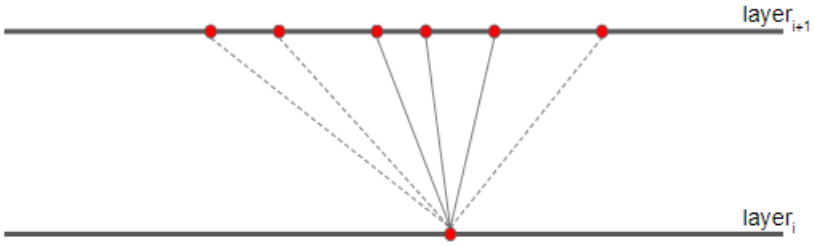

Figure 2.2: Example of the tracklet reconstruction phase: in this example three tracklets are considered as valid and the other three, shown with dotted lines, are discarded.

2.1.3 Cells computing phase

In this phase two consecutive tracklets are combined to form a cell if they fulfill some geometrical criteria; hence we must consider three consecutive layers.
The presence of a magnetic field inside ALICE, causes the clusters of a cell to lay on a circle; to improve the computational performance, clusters are mapped on a paraboloid with the minimum point positioned on the interaction vertex.
With this mapping, a cell can be identified by its center $\{x_c,\ y_c\}$ and its trajectory radius ρ. Each cell, defined in terms of the free clusters $\{c_i,c_{i+1},c_{i+2}\}$ or in terms of the tracklets which formed it $\{t_i,t_{i+1}\}$, needs to satisfy the following constraints to be considered valid:

- $\Delta\phi$ and $\Delta\tan\lambda$ of the two tracklets have to be lower than their respective threshold values.
- The DCA_z, distance of closest approach along z axis, defined in 2.12, must be smaller than ${DCA_z}^{MAX}$

$$DCA_{\mathrm{z}} = \left| \frac{\tan\lambda_{\mathrm{T_1}} + \tan\lambda_{\mathrm{T_1}}}{2} \cdot r_{\mathrm{c_i}} + (z_{\mathrm{v}} - z_{\mathrm{c_i}}) \right| \tag{2.12}$$

- The $DCA_{x,y}$, the projection of the distance of closest approach to the interaction vertex V on the xy plane, defined in 2.13, must not exceed ${DCA_{xy}}^{MAX}$ threshold value.

$$DCA_{\mathrm{xy}} = \left|\rho - \sqrt{{x_{\mathrm{c}}}^2 + {y_{\mathrm{c}}}^2}\right| \tag{2.13}$$

The values of threshold mentioned before can be found in 2.2.

Table 2.2: Threshold values used inside cell finding phase to consider a couple of tracklets as a valid cells.

Layer	1	2	3	4	5
$\Delta\phi^{\mathrm{MAX}}$	0.14				
$\Delta \tan\lambda^{\mathrm{MAX}}$	0.025				
${\mathrm{DCA_z}}^{\mathrm{MAX}}$	0.2	0.4	0.5	0.6	3.0
${\mathrm{DCA_{xy}}}^{\mathrm{MAX}}$	0.05	0.04	0.05	0.2	0.4

As shown in fig. 2.3, cell finding phase tries to combine the cluster of three layers and in particular try to combine tracklets of successive layers which share a cluster, the middle one of the cell generated, with a computation complexity of:

$$T(N) = O(N^3) \tag{2.14}$$

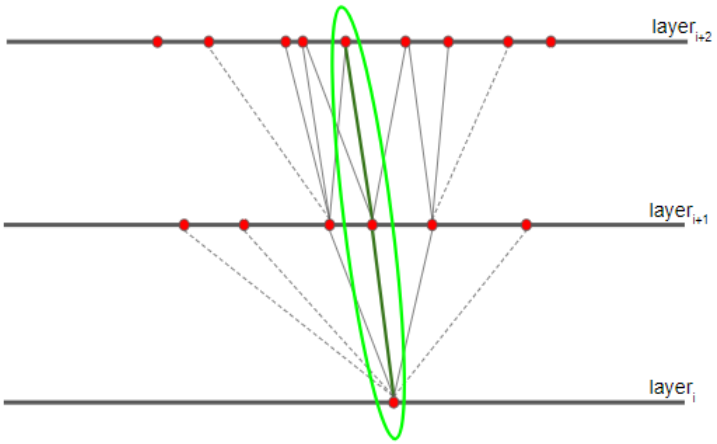

Figure 2.3: Example of cell reconstruction phase: two tracklets, represented in green, are combined to create a cell.

2.1.4 Cell neighbourhood finding phase

Each cell is ranked using a neighbourhood index: two contiguous cells are considered neighbourhood and receive a rank if they share a tracklet and satisfy some criteria based on the difference of the curvature of cells and their coordinates. A cell receives a rank equal to the maximum rank of its neighbourhood plus one; the cells without any neighbourhood receive a rank equal to one. This phase has a computational complexity that only depends on cells number N:

$$T(N) = O(N^4) \tag{2.15}$$

2.1.5 Track reconstruction phase

All neighbourhood cells are combined each other in order to reconstruct complete roads. A road is the union of, at least, 5 clusters, and represents a track candidate. Starting from the most external cells the algorithm recursively analyses each cell until the interaction vertex and chooses the appropriate road. The complexity of this phase is

$$T(N) = O(N^7) \tag{2.16}$$

2.1.6 Fitting phase

The output of the previous phase is a set of all the tracks candidate. A Kalman Filter is applied to the roads which share one or more clusters: only the road with the minimum value of χ^2 is kept. Figure 2.4 shows three steps of the tracking algorithm, in particular the figure 2.4c represents schematically the fitting phase applied to two track candidates which share the outermost cluster.

2.2 CUDA implementation

As already anticipated the goal of the present thesis project is the OpenCL parallel implementation of the upgraded ITS tracking algorithm. The achieved results will be also compared with those available and obtained using Compute Unified Device Architecture (CUDA)[12]. To this purpose in this section I will briefly describe the main achievements reached with a parallel version of the algorithm using the CUDA framework [10]. The CUDA version has been designed as a standalone package starting from the algorithm presented in [6] which is integrated in the AliROOT framework. In order to guarantee the maximum compatibility the whole code is based on C++14 Standard Template Library (STL) and the external libraries needed by AliROOT are removed. The last phase of the algorithm, consisting in the Kalman Filter fitting, is not implemented in the CUDA version because it does not represent

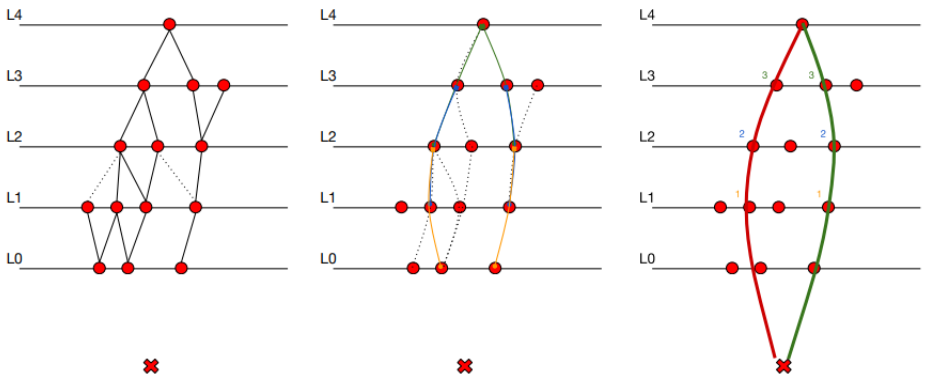

Figure 2.4: Three steps of the tracking algorithm: the red cross represents the interaction point, the red dots represent the clusters, the red line is a track candidate, the green line is a track candidate that passed the fitting phase.

a performance limitation if executed on CPU. One of the main results of the CUDA version of the algorithm is that, contrary to the AliROOT version, it can manage events with multiple interaction vertices for a single event (pile-up).

The Amdahl's law gives the maximum theoretical speedup S that can be reached parallelising a portion of a serial program and is reported in the following:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.17}$$

where P is the portion of serial program that should be made parallel, N is the number of compute units where the parallel code can run. For a GPU hardware, the number of compute units is very high, so the Amdahal's law can be rewritten considering N as infinite:

$$S = \lim_{N\to\infty} \frac{1}{1-P} \tag{2.18}$$

According to figure 2.5, about 90% of total execution time is spent for tracklet and cell finding phase, so parallelising these two the theoretical maximum speedup is equal to

$$S = \lim_{N\to\infty} \frac{1}{1-0.9} = 10 \tag{2.19}$$

Nevertheless, the overheads introduced by running a program on physical hardware (e.g. the copy of memory objects between host end device memory) make this limit impossible to achieve. Figure 2.6 summarizes the main step of CUDA implementation in particular splitting the operations in function of device where they will be executed.

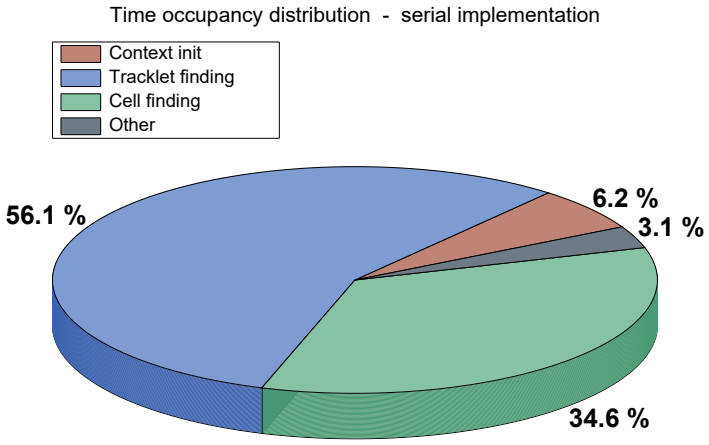

Figure 2.5: Time occupancy distribution for serial implementation.

2.2.1 Software architecture

The structure of classes of the CUDA version follows the AliROOT implementation structure and the O^2 coding guidelines[13] for a future integration on ALICE O^2 framework.

The event are read from a text input file and stored in `CAEvent` objects. The complete algorithm flow is implemented by using a `CATracker` object which, using the `clusterToTracks` method, is able to find a list of roads (`CARoad`) starting from the list of `CAEvent`.

So, the real core of the application is the `clusterToTracks` method: it calls a sequence of methods that implement the steps of tracking algorithm; all the intermediate results are stored in `CAPrimaryVertexContext` that is a data member of `CATracker` class. The information related to CUDA devices, memory objects and kernels are stored inside a `PrimaryVertexContext` object that is a data member of `CATracker`.

The final output of the program consists of three different list of tracks classified in function of their cluster's label assigned using Monte Carlo simulation and it can be:

- correct: all clusters share the same Monte Carlo label;
- duplicated: a correct track but its label has yet been associated to an other track (each input label corresponds to only one track);
- fake: at least one cluster has different label;

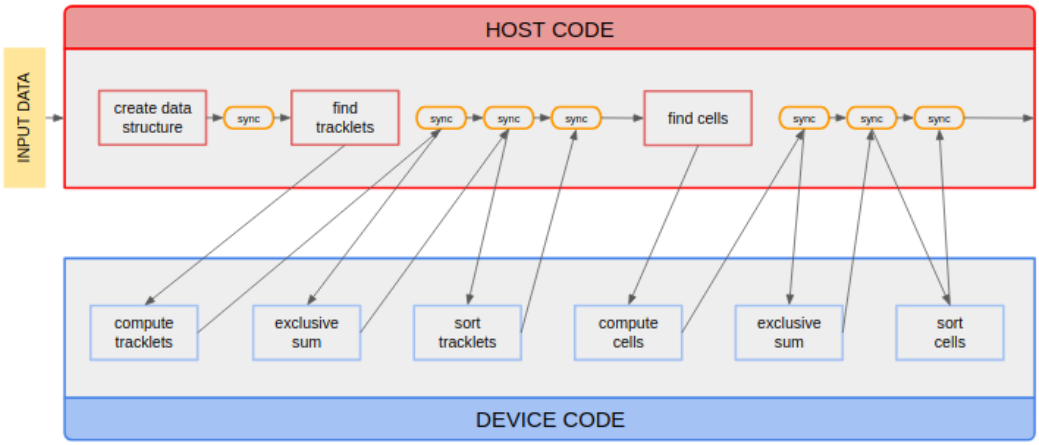

Figure 2.6: Host-device paradigm: host execution, red filled, and device execution, blue filled, must be properly synchronized.

The aforementioned classification is produced using an additional textual input file that stored all the tracks generated using a Monte Carlo simulation.

2.2.2 Initialization phase

After reading the input data from a textual file, all the structures used during the algorithm are created, indicating with N the number of ITS layers, as follows :

- *N* clusters vectors;
- *N-1* (empty) tracklets vector;
- *N-2* (empty) lookup tables for tracklets;
- *N-2* (empty) cells vector;
- *N-3* (empty) lookup tables for cells;
- the index table, properly filled with cluster index;
- other structure not used during tracklets/cells computation phase;

All this structures are necessary for any implementation version.

The initialization phase has a very high impact on the algorithm computing performance and in particular the creation or the reallocation of memory is very expensive in terms of execution time.
For this reason the `CAPrimaryVertexContext` object, which contains all the allocated

structures, can be used for all the events simply resetting the internal structures with the new data, and resizing these structures only if necessary (the new size is greater than the previous one).

2.2.3 Tracklet finding phase

As seen in the section 2.1.2, the goal of this phase is to find all possible valid tracklets between two consecutive layers; the input data consist of vectors of clusters and the index table. The algorithm can be split in three consecutive phases that must be executed in series and a synchronization mechanism is needed to guarantee that:

- all the tracklets are found and stored in the respective vector: each cluster is managed in parallel so the vector obtained is unsorted; in order to put each tracklet in the correct position of the vector the use of atomic variables is necessary to avoid any overlapping between different tracklets; during this phase the tracklet lookup table is filled with the number of tracklets found of each cluster, for example, if 13 tracklets with the 42^{nd} cluster as first cluster (index of index table = 41) are found, the 42^{nd} element of tracklet lookup table (relative to the layer where the cluster is found) is filled with number 13;

- after the end of the previous phase the second step can start: an exclusive sum, also known as prefix scan, is performed over all the tracklets lookup tables; this operation is necessary for the cell computing phase; fig. 2.7 shows how the scan works.

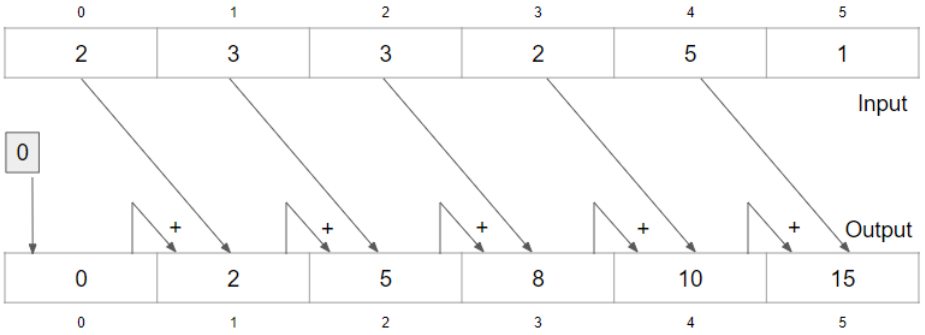

Figure 2.7: Exclusive sum: each i-th element of the output vector is filled summing from the first element up to the (i-1)-th element of the input vector

- the last step of this phase consists in the sorting of the tracklet vectors as function of the cluster index.

2.2.4 Cell finding phase

The cell finding phase implementation is very similar to the previous one: using the tracklets vector as the cluster vectors we can find all the possible valid cells. Also in this case the algorithm can be split in three consecutive phases:

- all the valid cells are calculated and stored in the relative cell vector; the cell lookup table are filled with proper values.
- Exclusive sum is applied to the cell lookup tables.
- Cells are sorted on the basis of the tracklet index.

2.3 Performance analysis

In this chapter I will show the results of the performance analysis of both the serial and the CUDA implementation of the tracking algorithm. All benchmarks reported below have been obtained by using the following configuration:

- CPU: Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz
- GPU: NVIDIA TITAN Xp 12 GB GDDRX5 1405 MHz
- RAM: 16 GB (2x8GB) 2666 MHz DDR4 ECC RDIMM
- OS: Ubuntu 18.04 LTS
- C++ Compiler (serial): gcc version 7.3.0, with -O3
- C++ Compiler (CUDA): NVCC 9.2
- CUDA Version 9.2.88

To better compare the OpenCL implementation with the current implementations, also the performance of serial program is analyzed. The serial version follows the same guidelines reported in 2.2.1. The input file used to evaluate the performances contains 100 Pb-Pb events, because the central events are those with the highest numbers of particle emitted per collision. The first analysis concerns the efficiency over transverse momentum p_T. As shown in 2.8, the efficiency saturates around a p_T=4 GeV/c and quickly decrease for $p_T \leq 0.5$ GeV/c. From the point of view of the time occupancy, figs. 2.5 and 2.9 show how the total execution time is divided among the different phases: it is evident that for serial implementation, tracklets and cell finding occupy more than half of total time; on the other hand, successive phases, cells neighbours finding and track reconstruction, take a very

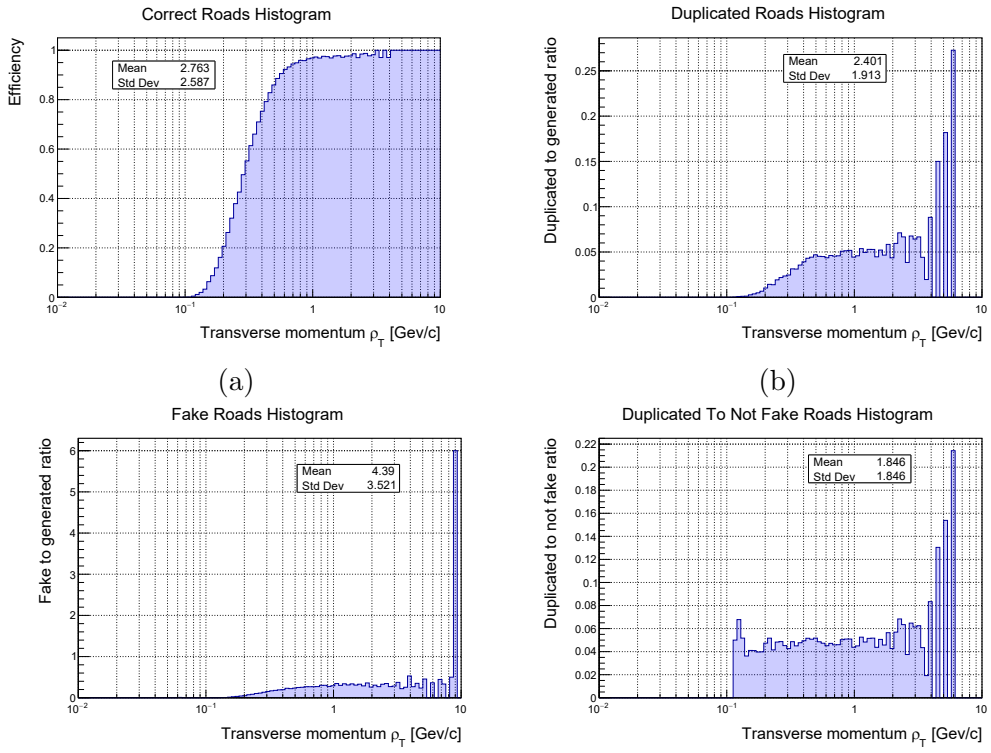

Figure 2.8: Efficiency evaluation of CPU algorithm as a function of the transverse momentum for a sample of 100 central Pb–Pb events without pile-up obtained dividing the number of reconstructed roads over the total number of generated roads.

small portion of the time; this means that the parallelization of these phases would not infer an important improvement on the algorithm performance; on the CUDA version, the initialization phase has the higher impact because of the parallel implementation of the successive phases, so the operations of tracklet and cell finding take only 36.7% of total execution time.

Table 2.3 and 2.4 summarize the execution time of the most important phases for the serial and the CUDA implementation and it also shows that the pile-up has an important impact on the performance.

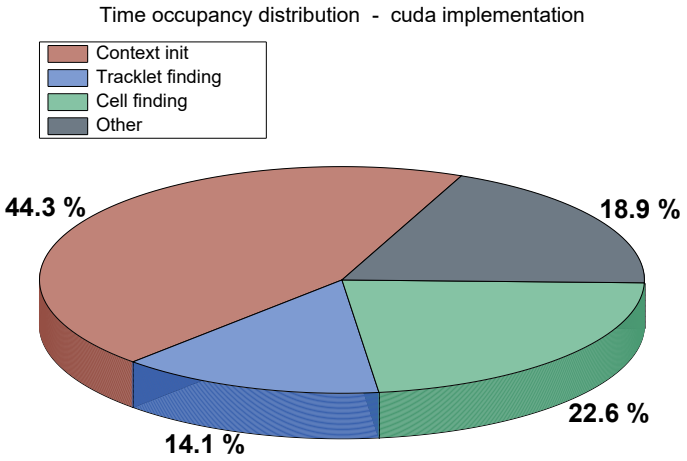

Figure 2.9: Time occupancy distribution for CUDA parallel implementation.

Table 2.3: Serial implementation: min, mean and max time (in ms) for execution of 100 Pb-Pb events with different amount of pile-up.

#Vertices		1	2	4	5
Context init	min	4.5	7.6	17.9	24.3
	mean	5.7	10.1	22.5	29.4
	max	7.6	14.3	35.5	45.9
Tracklet finding	min	23.6	102.8	435.3	693.6
	mean	51.6	180.2	677.3	1027.6
	max	74.1	227.2	831.1	1259.6
Cell finding	min	11.5	58.1	399.6	745.1
	mean	31.8	151.3	943.7	1698.1
	max	49.5	211.0	1297.7	2324.1
Total	min	40.0	169.8	855.4	1466.6
	mean	92.0	346.2	1652.9	2768.0
	max	134.7	455.4	2149.3	3602.3

Table 2.4: CUDA implementation: min, mean and max time (in ms) for execution of 100 Pb-Pb events with different amount of pile-up.

#Vertices		1	2	4	5
Context init	min	5.7	9.3	22.2	28.5
	mean	7.2	13.2	28.8	38.2
	max	10.4	19.4	49.8	66.7
Tracklet finding	min	1.7	4.2	20.0	32.3
	mean	2.3	5.2	23.9	36.9
	max	3.1	7.7	29.6	44.2
Cell finding	min	1.4	4.9	31.7	50.1
	mean	3.7	13.3	62.4	102.9
	max	5.8	18.1	84.0	139.8
Total	min	9.3	19.5	78.3	117.8
	mean	16.2	37.0	124.7	191.1
	max	23.9	48.1	173.1	259.1

Chapter 3

OpenCL

3.1 Overview

OpenCL[14] is an open source framework for parallel programming based on ANSI C/C++ and a host-device architecture paradigm. The program generated using OpenCL can be executed on heterogeneous platforms including CPU, GPU, FPGA, DSPs and other hardware accelerators. The operations are performed thorough kernels that are programs executed by devices and written with OpenCL C/C++ programming language which is based on C99.

The OpenCL framework allows writing parallel code that scales over the number of compute unit available in a transparent way from the programmer point of view.

The ability to execute programs over so many different type of devices is the real strength of this framework, indeed it is possible to run programs on very different devices which span from smartphones to notebooks to supercomputers. This is also the main important advantage with respect to its main competitor, namely NVIDIA CUDA, which specifically targets NVIDIA GPU hardware; on the other hand, CUDA performances are generally better than OpenCL ones[15].

3.2 History

The first proposal of OpenCL was stipulated by Apple in 2008, and after a re-definition in collaboration with AMD[16], NVDIA[17], Qualcoom[18], IBM[19] and INTEL[20], the proposal was submitted to Khronos Group [21] which approved the first public version on December 8, 2008. Khronos Group is a non-profit american consortium founded in 2000 with the goal to provide a cooperation platform for industry players to develop open standards for cross-platform technology.

3.2.1 OpenCL 1.x

The first version of OpenCL was released by Apple in August 2009 along with Mac OSX Snow Leopard. In the following months other companies (like AMD, NVDIA and IBM) announced the intention to support OpenCL standard for their future devices. The successive versions, 1.1 and 1.2, were released in June 2010 and November 2011 respectively, and they were characterized by many important features to increase control of parallelism and performance (i.e. more OpenCL built-in C function, new data types, operations on buffer regions, capacity to force IEEE 754 for single precision math operations, the possibility to include OpenCL program inside external library).

3.2.2 OpenCL 2.x

The 2.0 version was released by Khronos Group in November 2013 and it included new features like generic address space, nested parallelism, atomic functions and shared virtual memory. With the 2.1 version, released in November 2015, along with new features and performance improvements, it was added the ability to use C++14 to write OpenCL kernel. The 2.2 version is the last OpenCL version released (on May 16^{th} 2017) and it includes features aimed to optimize the generated code, the ability to use C++ inside OpenCL library functions in order to increase safety and to improve performances; at present the kernel language is a subset of C++14 language, including classes, lambda, template and others functionality.

3.2.3 Future

After the release of the version 2.2 , Khronos Group announced that OpenCL program will be merged into the Vulkan project[22] which is also managed by the Khronos Group.

3.3 The OpenCL Architecture

To properly describe the OpenCL architecture it is convenient to split it in different models that are strictly linked together[23]:

- Platform model
- Execution model
- Memory model
- Programming model

3.3.1 Platform model

The platform model describes how the OpenCL framework sees the hardware configuration of the system where the OpenCL programs will be run. It is always composed by a single host and one or many OpenCL devices. The host is the part of the system where the OpenCL execution starts and it is in charge of the management and the interaction with OpenCL *context* (which includes *devices*, *command queues*, memory and all the operations that involve them) through OpenCL API; each device is composed by more compute units and each compute unit can be split in Processing Elements (PEs) which are the real calculation units. Figure 3.1 shows a possible subdivision of the hardware system from the platform model point of view.

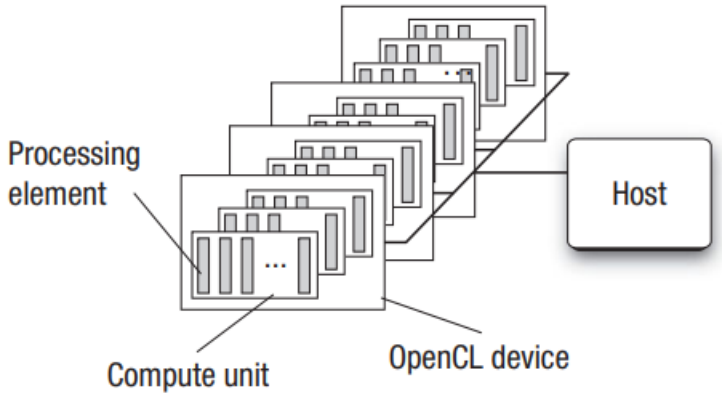

Figure 3.1: OpenCL Platform model: one host and four devices, each one split in compute units and their corresponding processing elements[24]

3.3.2 Execution model

The normal execution of an OpenCL program can be split into two parts: the host program executed on the host and the kernel executed by one or more devices. After the creation of context, the host can create one or more command queues associated to a particular device in order to launch a kernel; through the command queue the host can manage kernel execution commands, memory commands and synchronization commands. Based on a specific command queue configuration, each command can be executed following the order of submission (In-order Execution) or in any order (Out-order Execution) constrained only by the used synchronization mechanism. When the kernel is submitted, an index space, associated to it, is defined; the index space, called NDRange, is divided into work-groups and each work-group is composed by many work-items: each work-item represents a single

point of index space and it executes an instance of kernel. It is possible to identify a work-item globally (based on its global ID) or with respect to its work-group (based on group ID and local ID). Figure 3.2 summarizes how a single work-item can be identified. NDRange kernel is defined by an integer array of length N (with N from 1 to 3) that indicates the global size for each dimension (total number of work-items).

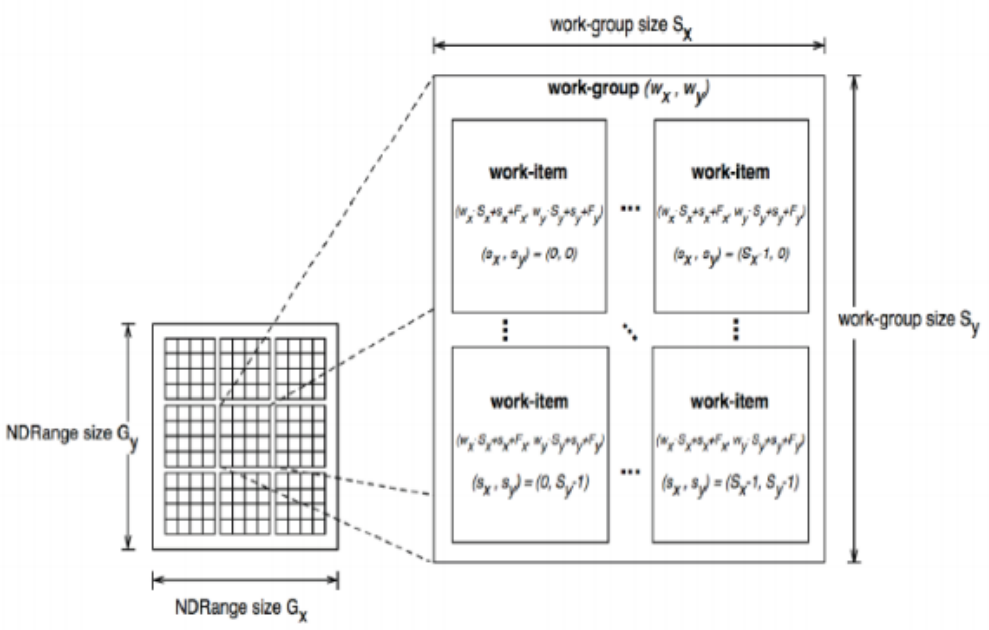

Figure 3.2: Example of NDRange partition with N=3, work groups of same size and offset equal to 0[24].

3.3.3 Memory model

The OpenCL device memory can be divided in the following types:

- **Host Memory:** only the host can access (read/write) this part of the memory;
- **Global Memory:** each work-item can read/write this memory;
- **Constant Memory:** this region remains constant for all the executions and the work-item can only read it;
- **Local Memory:** memory shared within a work-group; it can be used to allocate objects shared among all the work-items inside the same work-group;

- **Private Memory:** this region is private for each work-item and no other work-item can read/write this region;

A schematic representation of memory model is shown in fig.3.3.

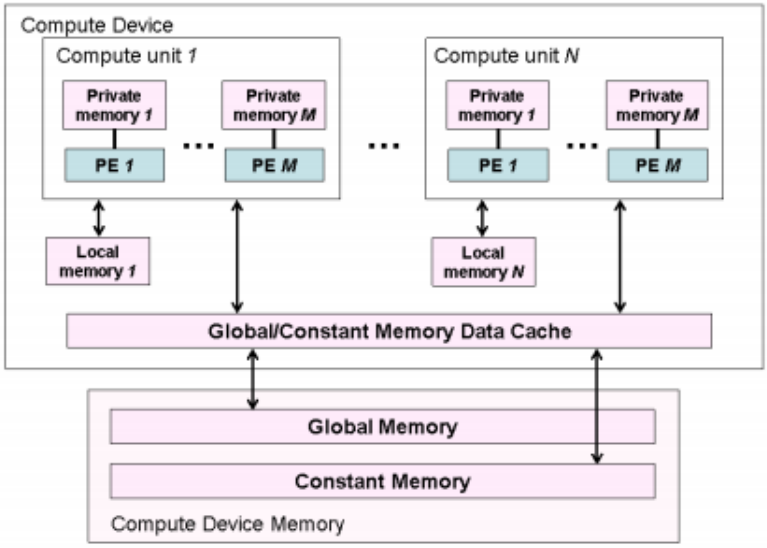

Figure 3.3: Representation of memory model[24].

3.3.4 Programming model

OpenCL supports task parallelism, data parallelism and a hybrid version combining both. The choice of the model to follow depends on the algorithm and on the used data structure.
In particular, the **data parallelism model** is strictly linked to the data employed: there is a mapping between a work-item and the element in a memory object over which a kernel can be executed in parallel. The key of this model is the NDRange definition where the programmer can define the total number of work-items and how they are divided in work-groups (or leave OpenCL implementation free to find the better work-group size). If different work-items of the same work-group should access the same memory a synchronisation mechanism is needed to prevent a wrong memory access. It is important to notice that there is no synchronisation mechanism between work-items from different work-groups. OpenCL defines a task as a kernel executed by a single work-item: this operation is equivalent to launch a work-group with a single work-item inside. For the implementation of the **task parallelism**, the host has to deal with the synchronisation among many kernels and each of them executes a different operations on the data.

3.4 Memory objects

One of the main expensive phase of any OpenCL program is the transmission of data to and from the device: to speed up the execution of the program it is very important to set a limit to the number of read/write operations. OpenCL provides different types of memory objects that can remain on the device reducing data exchanges between host and device. An OpenCL memory object can be one of the following types:

- buffer: one dimensional array of data;
- sub-buffer: one dimensional section of buffer;
- image: a specialized type of memory object that is used for accessing 2D and 3D image data;

Buffer and sub-buffer are strictly linked together and share the large part of properties and methods applicable over them, but all three categories of object listed before have some common properties. Memory objects are allocated over an OpenCL context and all the devices associated to that context can access the memory object; host can perform read/write operations submitting command to a specific queue (created using the same context used for memory object) and these operations can be blocking or not. In the case of not blocking operation, host must provide a synchronization method (`clFinish()` or event handler) to avoid the use of buffer before the end of the read/write operation.

Image objects are objects to store images and to exploit the high performance texturing hardware of a GPU. An image object includes some information about the image, such as dimensions and format, and easily allows performing some operations like clamping and filtering.
Buffers and sub-buffers are the most used objects and they are described in detail in the next section.

3.4.1 Buffers and sub-Buffers

OpenCL buffers are the common memory object used to store scalar, vector or custom data types. All the considerations about buffer are also valid for sub-buffers. The C++ API functions used to create buffer and sub-buffer are the following:

```
cl::Buffer::Buffer(
    const Context& context,
    cl_mem_flags flags,
    ::size_t size,
    void * host_ptr,
```

```
    cl_int * err);
```

```
cl::Buffer cl::Buffer::createSubBuffer(
    cl_mem_flags flags,
    cl_buffer_create_type buffer_create_type,
    const void * buffer_create_info,
    cl_int * err = NULL);
```

The *context* is an OpenCL context over which the memory will be allocated; the *size* specifies the number of bytes to allocate; if *err* is not NULL, the returned error code will be stored on it; *host_ptr* is a pointer to allocated data, in particular the use of this parameter depends on the *flags* specified and if it is different from *NULL*, the allocation size of *host_ptr* has to be greater or equal to *size*.
The most important parameter to be specified is *flags* that indicates the type of operations doable over a memory object and the allocation method used to create it:

- CL_MEM_READ_WRITE: kernel can read and write buffer memory; if *flags* is not specified, this is the default option;
- CL_MEM_WRITE_ONLY: only write operation can be performed by the kernel; reading operation may lead to undefined results;
- CL_MEM_READ_ONLY: only read operation can be performed by the kernel; writing operation may lead to undefined results;
- CL_MEM_USE_HOST_PTR: memory referenced by *host_ptr* (that cannot be *NULL*) is used to store data;
- CL_MEM_ALLOC_HOST_PTR: memory is allocated in host-memory;
- CL_MEM_COPY_HOST_PTR: the OpenCL implementation allocates memory for memory object and copy data by *host_ptr*;

These parameters can be OR combined but CL_MEM_READ_WRITE, CL_MEM_WRITE_ONLY and CL_MEM_READ_ONLY are mutually exclusive, and the same is valid for CL_MEM_USE_HOST_PTR and CL_MEM_ALLOC_HOST_PTR.

3.4.2 Buffer operations

The memory of buffer objects can be read or written by the host submitting the relative command to a command queue. In particular, instead of coping data at buffer creation moment, the buffer can be filled with the `clEnqueueWriteBuffer()`

function specifying the data to be written, the size of the data and if the operation is blocking or not. With `clEnqueueReadBuffer()` a read operation is performed and also in this case one has to specify the size of the data to read and the pointer to the memory where those data will be stored.
OpenCL also provides two additional functions to access memory objects.
The `clEnqueueMapBuffer()` function maps a buffer region into the host address space and it returns a pointer to this region; the host can access and modify this region and after that operation, by using `clEnqueueUnmapMemObject()`, all the modifications are removed from the device memory.

3.5 Example of program flow

In most of the applications, an OpenCL program starts asking for the OpenCL supported platforms present in the system and selects a device on which to launch kernels; in particular one can ask for CPU, GPU, accelerator or any of these kind of device.

```
/* Simple program for sum two integer arrays */
/* get list of all platforms */
cl::Platform::get(&platformList);

/* select the first platform */
cl::Platform platform=platformList[0];

/* get list of all GPU devices associated to the selected platform */
platform.getDevices(CL_DEVICE_TYPE_GPU, &deviceList);

/* select the first device found */
cl::Device device=deviceList[0];

/* create the context for selected device */
cl::Context context({device});
```

After obtaining context and device, the next step is to create a command queue used to submit operations on the selected device; as seen before, the creation options of command queue determines if operations will be executed following the submission order or not.

```
/* Create a in-order execution command queue; specifying
   CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE we obtain an out-of-order
   command queue */
cl::CommandQueue queue(context,device);
```

The next phase is the creation of buffers to hold all the data used by the kernel; after this phase one can decide if the buffer memory can be read, written or both and the size of the memory to allocate; in order to read/write to/from this memory the host must use OpenCL API functions like `clEnqueueReadBuffer()` or `clEnqueueWriteBuffer()`. These functions submit the relative operation (read or write) to the command queue. It is also possible to specify the input data that fill the buffer at the creation moment.

```
/* Create 3 buffer for host input and output array: in this case the
   buffer memory is read/write because no specific option is specified */
int v1[]={0, 1, 2, 3, 4};
int v2[]={5, 6, 7, 8, 9};
int v3[N]; /* N = 5 */
cl::Buffer bInputV1(context,CL_MEM_USE_HOST_PTR,N*sizeof(int), v1, NULL);
cl::Buffer bInputV2(context,CL_MEM_USE_HOST_PTR,N*sizeof(int), v2, NULL);
cl::Buffer bOutput(context,CL_MEM_USE_HOST_PTR,N*sizeof(int), v3, NULL);
```

The following step is to create the program one wants to execute on device; the source code can be read from an external file or from a local string variable; after this step, the program must be build and compiled: this operation can be done at runtime (not device dependent but more latency is added for loading the source code) or off-line (pre-compiled, no latency is added but the code can be device dependent). Specific OpenCL compile options can be specified during build operation.

```
/* kernel source code */
    std::string source_code=
        "kernel void sum2Vector(                    "
        "   global const int* v1,                   "
        "   global const int* v2,                   "
        "   global int* output){                    "
        "       int index=get_global_id(0);             "
        "       output[index]=v1[index]+v2[index];  "
        "   }                                       ";

    cl::Program::Sources sources;
    sources.push_back({source_code.c_str(),source_code.length()});
    cl::Program program(context,sources);
    program.build({device});
    cl::Kernel kernel=cl::Kernel(program,"sum2Vector");
```

The program flow foresees at this point the submission and execution of kernel: one adds the argument to the kernel (in this case input and output buffers) and then submits the kernel to the command queue. In this phase it is possible to specify

the dimensions of NDRange that depend on the selected device and on the program granularity:

- **work dim:** the number of dimensions N of NDRange with N greater than 0 and less than or equal to CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS; if not specified the default value is 1.
- **global work offset:** it specifies the offset used to calculate work-item global ID.
- **global work size:** it is strictly linked to the program granularity, for example, in the case of the present thesis work, the number of work-item should be equal to the array size.
- **local work size:** if this value is NULL OpenCL implementation will determine an appropriate work group size; it must be less than or equal to CL_DEVICE_MAX_WORK_GROUP_SIZE.

```
kernel.setArg(0,bInputV1);
kernel.setArg(1,bInputV2);
kernel.setArg(2,bOutput);
queue.enqueueNDRangeKernel(
   kernel,           /* kernel to execute */
   cl::NullRange,    /* global work offset */
   N,                /* global work size */
   cl::NullRange);   /* local work size */
```

The last phase is to read back the result of the kernel execution; it is important to notice that the function `enqueueNDRangeKernel()` submits the command to execute the kernel but it is not a blocking function so it does not give any information about the start and the end of the execution; it is necessary, therefore, to use some synchronization mechanism to ensure that the execution is ended before reading back output: OpenCL `finish()` is a blocking function that returns only when all operation on the command queue is completed. After that operation it is possible to read the result from the buffer. Figure 3.4 represents how each kernel instance uses its global index to access the correct element of input vector and store the result in the correct position of the output vector. Figure 3.5 describes the complete flow of a common OpenCL program.

```
queue.finish();
queue.enqueueReadBuffer(bOutput,CL_TRUE,0,sizeof(int)*N,v3);
/* v3 = { 5, 7, 9, 11, 13 } */
```

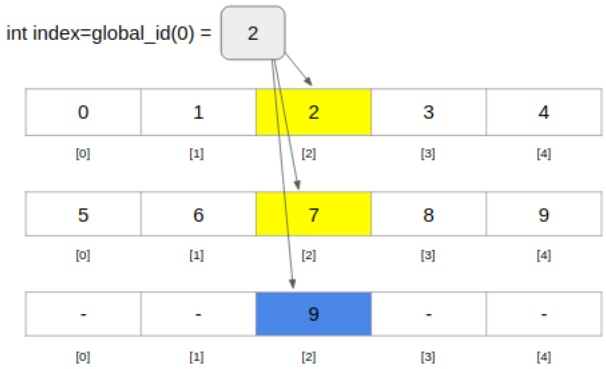

Figure 3.4: Example of how an OpenCL kernel access data using its global index. Yellow blocks indicate the input value read by the kernel from input data and correspond to the position indicates by global index. The result, the blue block, is stored in the output vector at the same index position.

3.6 Performance analysis tool

Using the Intel software Intercept Layer for OpenCL Applications[25] it is possible to analyse the performance without changing the program code. The software generates a log file, where all the OpenCL calls are reported, and a JSON file that can be opened by using the Google Chrome browser. For example, opening the JSON file generated from the program reported before, it is possible to identify each phase of execution. The device selected is a CPU and the command queue is configured for in-order execution.

As shown in figure 3.6 the programs takes 92.6 ms to finish but most of this time is used to get the platform, create the context and build the kernel (about 83 ms); the real programs core is executed in 88 μs and include buffer creation, kernel's argument setting, launch of kernel, waiting time before the end of the kernel and the copy of the result back to the host vector. About 10 ms are used to release all OpenCL objects (command queue, context and device(s)). It is important to notice how `finish()` function is necessary to prevent the reading of the output buffer (made by API function `enqueueNDRangeKernel()` called inside `enqueueReadBuffer()`) before the end of the kernel execution (details can be seen in figure 3.7).

Intercept Layer for OpenCL Applications is particularly useful when the program uses more than one command queue; small changes to the previous program is necessary to show this kind of application:

- creation of 2 command queues and 2 output buffers. To obtain a better visualization of the input/output, the vector size is increased to N = 500000 and the work-group size is set to 1).

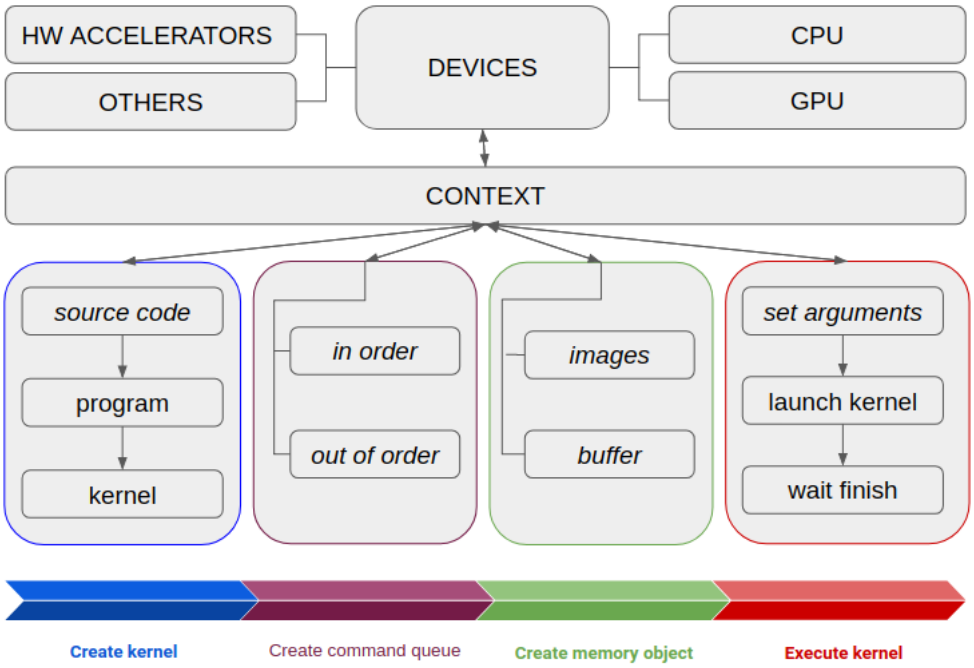

Figure 3.5: OpenCL flow diagram from kernel creation to kernel execution

```
cl::CommandQueue queue1(context,device);
cl::CommandQueue queue2(context,device);
cl::Buffer bOutput1(context,CL_MEM_USE_HOST_PTR,N*sizeof(int), v3,
    NULL); /* output buffer for first kernel */
cl::Buffer bOutput2(context,CL_MEM_USE_HOST_PTR,N*sizeof(int), v4,
    NULL); /* output buffer for second kernel */
```

- the launch of 2 parallel kernels and the use of finish() only after putting a second kernel in the queue.

```
kernel.setArg(0,bInputV1);
kernel.setArg(1,bInputV2);
kernel.setArg(2,bOutput1);

queue1.enqueueNDRangeKernel(kernel,
      cl::NullRange,
      cl::NDRange(N),
      cl::NDRange(1));

kernel.setArg(0,bInputV1);
```

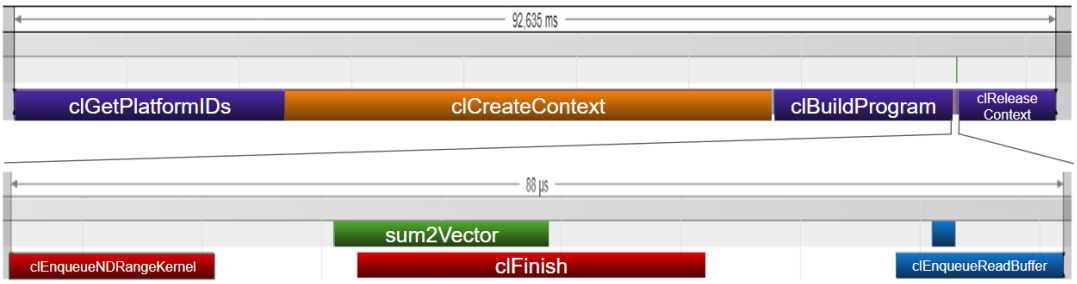

Figure 3.6: Execution flow of program sum2vector generated using Intercept Layer for OpenCL Applications.

Figure 3.7: Without `finish()` after sum2vector kernel launch, the reading operation starts before the end of kernel execution so the final result is not correct.

```
kernel.setArg(1,bInputV2);
kernel.setArg(2,bOutput2);
queue2.enqueueNDRangeKernel(kernel,
      cl::NullRange,
      cl::NDRange(N),
      cl::NDRange(1));


queue1.finish();
queue2.finish();
```

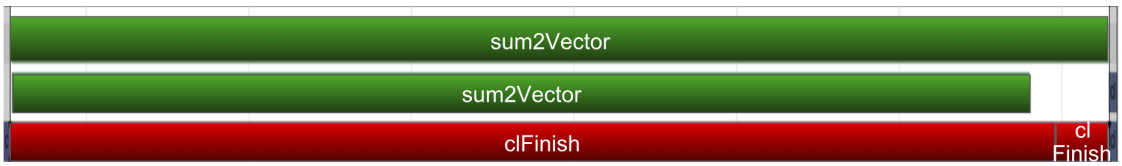

Figure 3.8: Example of multiple command queue running in parallel

The first 2 rows (in green) of figure 3.8 show the command queues execution, the last (the red one) show the functions called by the host; it is easy to recognize that using multiple queues we can improve the performance with an additional level of parallelism. A similar behaviour can be achieved by using a single command queue but with out-of-order mode (figure 3.9).

```
cl::CommandQueue
    queue(context,device,CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE);
kernel.setArg(0,bInputV1);
kernel.setArg(1,bInputV2);
kernel.setArg(2,bOutput1);
queue.enqueueNDRangeKernel(kernel,       /* kernel to execute */
      cl::NullRange,  /* global work offset */
      cl::NDRange(N), /* global work size */
      cl::NDRange(1)); /* local work size */

kernel.setArg(0,bInputV1);
kernel.setArg(1,bInputV2);
kernel.setArg(2,bOutput2);

queue.enqueueNDRangeKernel(kernel,       /* kernel to execute */
      cl::NullRange,  /* global work offset */
      cl::NDRange(N), /* global work size */
      cl::NDRange(1)); /* local work size */

queue.finish();
queue.enqueueReadBuffer(bOutput1,CL_TRUE,0,sizeof(int)*N,v3);
queue.enqueueReadBuffer(bOutput2,CL_TRUE,0,sizeof(int)*N,v4);
```

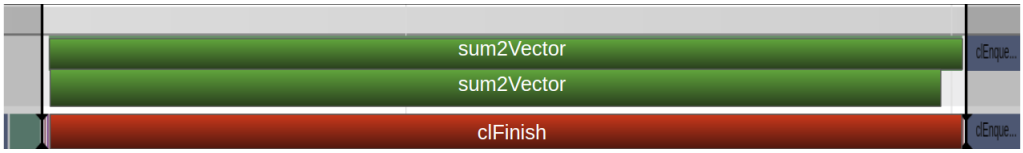

Figure 3.9: Execution of 2 kernels on an out-of-order command queue: the green rows correspond to the same command queue: they are executed in parallel and without respecting any particular order.

3.7 OpenCL vs CUDA

The main alternative to OpenCL is NVIDIA CUDA. CUDA identifies a set of hardware architecture and, as OpenCL, it is a general purpose platform for parallel computing over supported General-Purpose computing on Graphics Processing Units (GPGPU).

Contrary to OpenCL, CUDA is a proprietary technology and it works only on NVIDIA's GPUs; since CUDA is developed by the same company (NVIDIA) that

produces the hardware where the program will be executed, it offers more access to features and better performance [15]. In addition, CUDA offers a debugger and a profiler, OpenCL does not.

As OpenCL, CUDA is based on a host-device architecture paradigm and the execution starts from host which can execute computation kernels over device. Kernels are written in CUDA C (which widely supports C++) or PTX (an intermediate code that is compiled and translated by device driver to the actual device machine code) and they are compiled with *nvcc* compiler. The CUDA functionality can also be used with OpenCL interface provided by NVIDIA. As shown in fig.3.10 the flow execution of OpenCL and CUDA programs are very similar and often only the syntax or the name are different.

Device, execution and memory models of the two frameworks are very similar but the OpenCL models are more generic and abstract because it does not address a specific architecture as CUDA does (i.e. OpenCL substitutes processor with processing element). Table 3.1 shows a comparison between CUDA and OpenCL most important terms and keyword.

C Runtime for CUDA	CUDA Driver API	OpenCL API
Setup		
	Initialize driver Get device(s) (Choose device) Create context	Initialize platform Get devices Choose device Create context Create command queue
Device and host memory buffer setup		
Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result	Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result	Allocate host memory Allocate device memory for input Copy host memory to device memory Allocate device memory for result
Initialize kernel		
	Load kernel module (Build program) Get module function	Load kernel source Create program object Build program Create kernel object bound to kernel function
Execute the kernel		
	Setup kernel arguments	Setup kernel arguments
Setup execution configuration Invoke the kernel (directly with its parameters)	Setup execution configuration Invoke the kernel	Setup execution configuration Invoke the kernel
Copy results to host		
Copy results from device memory	Copy results from device memory	Copy results from device memory
Cleanup		
Cleanup all set up above	Cleanup all set up above	Cleanup all set up above

Figure 3.10: List of the features of CUDA and OpenCL API[26].

Table 3.1: Terminology adopted by using CUDA and OpenCL [12][14].

TERM	CUDA	OpenCL
Device	int deviceId	cl_device
Queue	cudaEvent_t	cl_command_queue
Memory	void *	cl_mem
Grid of threads	grid	NDRange
Subgroup of threads	block	work-group
Thread	thread	work-item
Thread-index	threadIdx.x	get_local_id(0)
Block-index	blockIdx.x	get_group_id(0)
Block-dim	blockDim.x	get_local_size(0)
Grid-dim	gridDim.x	get_global_size(0)
Device Kernel	__global__	__kernel
Kernel Launch	<<<>>>	clEnqueueNDRangeKernel
Global Memory	__global__	__global
Group Memory	__shared__	__local
Private Memory	(default)	__private
Constant	__constant__	__constant

Chapter 4

OpenCL implementation

The necessity to run a tracking algorithm on many types of hardware, different from CPU or NVIDIA GPU, has been the key factor to write an OpenCL porting version. In the following I will present three different versions of the OpenCL porting I have developed describing the analysis of their performance, advantages and disadvantages, differences with respect to the serial/CUDA version. In the next chapters the following implementation schema will be discussed:

- sort version: based on CUDA implementation schema;
- native OpenCL version (from now on indicated as native version): by using OpenCL 1.2 API and Boost Compute library for scan phase only;
- boost version: entirely developed with Boost Compute and VexCL libraries;

At first, I will present some characteristics that all the versions have in common. To compare the results with those obtained with serial/CUDA implementation (presented in section 2.3), I run the program with the same input file containing a collection of 100 Pb-Pb events with different level of pile-up. The system configuration used has the following features:

- CPU: Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz
- GPU: NVIDIA TITAN Xp 12 GB GDDRX5 1405 MHz
- RAM: RAM: 16 GB (2x8GB) 2666 MHz DDR4 ECC RDIMM
- OS: Linux Ubuntu 18.04 LTS
- C++ Compiler: gcc version 7.3.0, with -O3 optimization flag
- OpenCL 1.2

Due to the high number of supported devices it was decided to choose the version 1.2 of OpenCL, even if some useful functions (e.g. built in scan) are not available by adopting this version.

4.1 General implementation choices

All the results presented in this section have been obtained using the boost version on GPU device. In order to preserve the compatibility with CUDA and the serial version of the algorithm, the interfaces of the used C++ classes are kept the same except for minor modification managed by using preprocessor directives. Adopting this approach, the version that one wants to execute can be selected at compile time with a compiler flag.

The general schema of the program follows the same guidelines of the other versions. A `CATracker` object contains all the methods needed to execute the steps of the algorithm, in particular through `clustersToTracks` method. The events read from the input file are stored into `CAEvent` objects and are passed as argument of `clustersToTracks` method. Each call of this method processes a single event.

During the first iteration over the first `CAEvent`, before the initialization of data structures, all the global OpenCL objects are created. In particular, starting from the available `cl::Platform` and a specific `cl::Device` it is possible to obtain a `cl::Context`. The `cl::Context` can be used to create all the `cl::CommandQueue` (as many as the number of the upgraded ITS layers) necessary during the whole program execution, and also to create and compile the `cl::kernel`s. Especially the pre-creation of kernels allows increasing the global performance of the algorithm avoiding to recompile the kernels whenever they should be invoked. All of the OpenCL objects are stored in a global `Context` object, that is declared with the keyword `final` so during the next iterations the OpenCL objects are not recreated with an important boost in terms of algorithm's performance.

The next step is the initialization of all the data structures used during the execution; those structures are stored in a `PrimaryVertexContext` object that is a member of `CATracker`. In order to create vector of appropriate size, in particular for host tracklets and cells, I use the same size used by CUDA version as explained in [10]. Depending on the program version, this phase is done in different modes, especially for the features derived from the use of Boost Compute library. The details of this phase are explained in the specific section of each version.

At this point the tracking algorithm steps can start by using the `CATracker`'s methods listed in the following:

- `computeTracklets()`
- `computeCells()`

- `findCellsNeighbours()`
- `findTracks()`
- `computeMontecarloLabels()`

The OpenCL porting concerns the first two methods, `computeTracklets()` and `computeCells()`, while the others are kept equal to the CUDA/serial version and executed in a serial way.

In order to perform all the steps of the tracking algorithm, it is necessary to set threshold values as explained in Cap.2. These values are kept the same calculated for the CUDA version described in [10]. The thresholds and the other constants used during the program execution (e.g. ITS's features as the number of layers) can be found in the *Constants.h* file. This file must be available not only for the host program, but also for the kernels. In any case, an adjustment of the file by adopting preprocessor directives, is necessary to allow the kernel to correctly include this file.

As explained in Cap.3, when a kernel is launched by the host, it is mandatory to specify NDRange for setting the number of dimensions (at most three), the total number of work items and the work group size (number of work items executed in parallel). The number of dimensions and the number of work items is strictly linked to the used data structure, the work group size is, instead, related to the features of the hardware: in this case, the parallelism depends only on the number of clusters (for the tracklet finding phase) or on the number of tracklets (for the cell finding phase), which are unidimensional variables: this means that the choice is to set an NDRange of dimension 1. The total number of work items deployed in each kernel execution is the smallest number greater than the size of processed buffers (containing clusters or tracklets) and as multiple of 32 or 64 (depending on the device hardware). For instance one would try to set the number of work items equal to the size of processed buffers. This is lesser performing than the previous setup. In figure 4.1 it is depicted the percentage difference between elapsed times in these two choices.

Using the smallest multiple of 32 or 64 as global size, a mechanism to prevent a wrong memory access must by implemented. For example one can operate as follows: during tracklet finding phases the parallelism depends on the number of clusters for each level. In particular, each kernel, using its global ID, accesses to the element of the clusters vector in order to read all the information about the cluster. Assuming a cluster vector of size N and to launch $N+k$ work-items ($N+k$ is the smallest multiple of 32, or 64, and higher than N), the *N+1-th* work-item will access an out of bound element (neither allocated nor correctly initialized). To avoid this behavior, the work-item must know the exact size of the clusters vector.

Regarding the work group dimension, two choices are possible: specify the work group size or let OpenCL implementation to decide a suitable dimension; in any

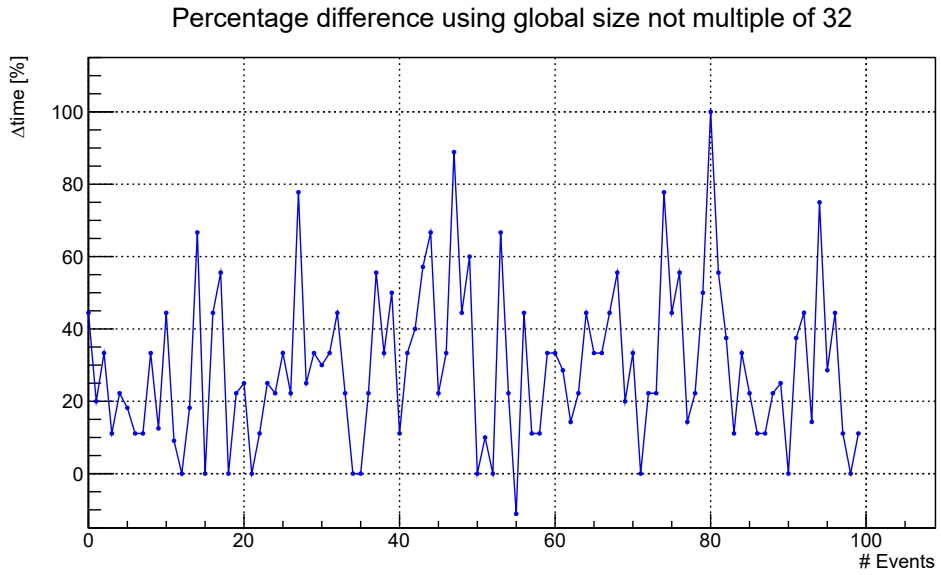

Figure 4.1: Computational time increment using a global size not multiple of 32. The system configuration used is the one reported in section 4.5.1.

case the size must be a sub–multiple of global work size and less than *CL_DEVICE_MAX_WORK_GROUP_SIZE* (maximum number of work-items inside a work-group and it depends on the device hardware); as seen before, the size is strictly linked to the features of the hardware device, so it could be difficult to find the better value for each device. For this reason I decided to follow the second approach and to let OpenCL free to choose the work group size.
Before the submission of kernel to a command queue, it is necessary to set the proper arguments through `clSetKernelArg()` function. The argument setting by using CUDA is done by means of a single *struct* which contains all the data structures necessary for the kernel execution. Differently from CUDA, OpenCL does not allow specifying variable length arrays and structures with flexible (or unsized) arrays as kernel arguments[24]. For each kernel the arguments object specified are as many as the vectors/arrays necessary for the execution.
Differently from CUDA version, all the versions developed with OpenCL use an *inclusive sum* instead of *exclusive sum*. As shown in 4.2, *inclusive sum* includes the input x_i to compute the output y_i.

The output of the program is composed by three text file reporting the list of correct roads, fake roads and duplicate roads. To produce these report files is necessary an additional input file containing the data of the Monte Carlo simulation.

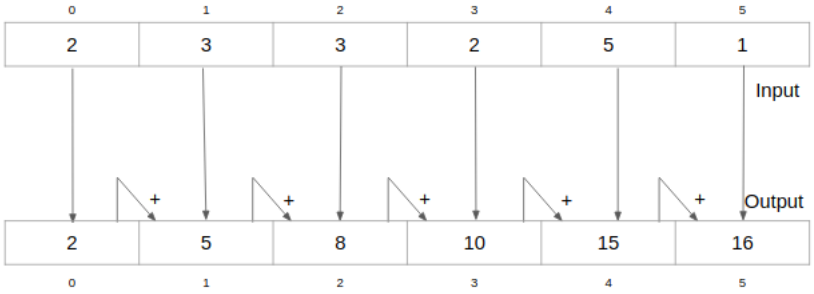

Figure 4.2: Inclusive sum: each output element y_i is filled summing the input element x_i plus the previous output element y_{i-1}.

4.1.1 Boost Compute library

All the code versions presented in this thesis are implemented using Boost Compute library.[27]
Boost Compute is a GPU parallel-computing library for C++ based on OpenCL which allows accessing to all the OpenCL features with a more user friendly C++ interface compared to OpenCL API, and provides some additional functions and parallel-computing algorithms, such as exclusive/inclusive scan, sort, accumulate and many others.
One of the main advantage derived from the use of this library is its C++ vector compliant API that simplifies and speeds up the copy operations of the `std::vector` used to store all the main structures used by the program (i.e. clusters, tracklets, cells) through the function `compute::copy()`.
Boost Compute is a header only library (no linking is required) and it is sufficient to include its header file during the compilation phase. This step is performed with CMake tool.

4.1.2 Compilation

The compilation phase is managed with CMake tool [28], an open source tool useful to automatize the compilation and to check if all requirements are met (i.e. compiler version, OpenCL version, Boost library presence). In addition, CMake is in charge of preparing the link to the OpenCL library, including header only libraries (i.e. Boost Compute) and the source files of the program. All the compilation flags can be specified using this tool; for this program `-std=c++14` and `-O3` are used.

4.2 Sort Version

The Sort version is very similar to the CUDA implementation and it is based on the same three steps performed by using the CUDA algorithm:

- compute tracklets/cells
- scan tracklet/cell lookup table
- sort tracklets/cells

Figure 4.3 shows the distribution of time execution using this version. The tracklet and cell finding phases take most of the time while only about 28.2% of total time is spent during the others phases .

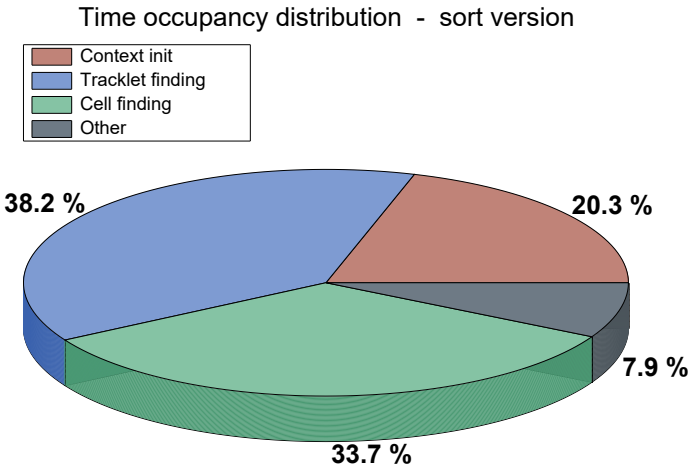

Figure 4.3: Time distribution over 100 Pb-Pb events using the sort version OpenCL porting. Others indicates all the other phases that follow the step in which the cells are computed.

4.2.1 Implementation details

During the initialization phase, all data structures are allocated and initialized (as seen in section 2.2). Boost Compute is helpful to simplify and speed up this phase. Indeed it allows creating `compute::vector`, that can be used inside `compute::kernel`, using directly `std::vector` (this is not possible with native OpenCL 1.2). In addition, Boost Compute allows the resize of previous allocated C++ `compute::vector`

transparently. The relative lookup tables are reallocated only if the required size is larger than their current one. The operation of creation and resize performed by Boost Compute uses a grow factor of 1.5 and this translates in the fact that the real allocated size is larger than expected. This is crucial to limit the number of reallocation but can create problem if the system has low memory. Figure 4.4 shows the initialization time of 100 Pb-Pb events.

During the execution of the program, some auxiliary buffers are necessary: one can be used to store the index of layer, one for the storage of the number of tracklets or cells found for each layer All these objects are created during the first execution and then, if necessary it is possible to perform a reset to zero for each event with the `compute::fill()` function. The only data available at this moment is the vector of clusters so the relative `compute::vector` can be filled with the `compute::copy_async()` function which is a non-blocking function that allows speeding-up the process; after the initialization phase and before the usage of initialized data, a `compute::finish()` must be executed on each command queue.

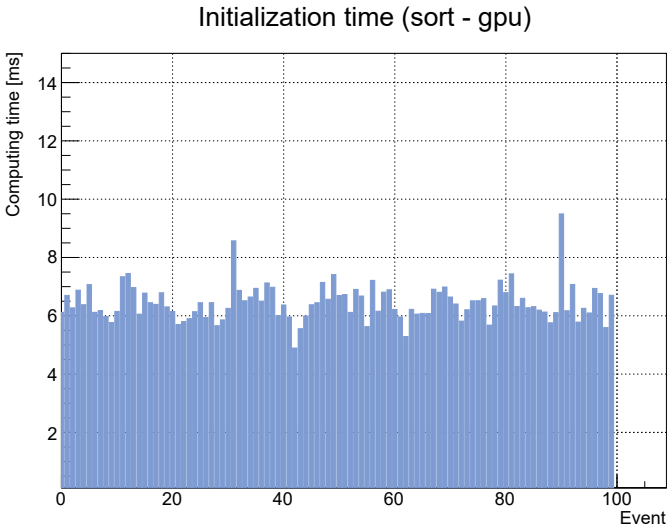

Figure 4.4: Initialization time analyzing 100 Pb-Pb events by using the sort version of the OpenCL porting.

As seen for sort version, three operations are performed during the tracklet finding phase. The first one, which corresponds to the computation of all the tracklets, can start only after the end of all the operations started during the initialization phase. For each layer, a kernel (4.1) is launched over the command queue relative to the layer: any instance of kernel tries to find all the tracklets that have the first cluster index equal to the *global ID*. All the valid tracklets are stored in an unsorted way. Each work-item is executed independently from the other ones: this means

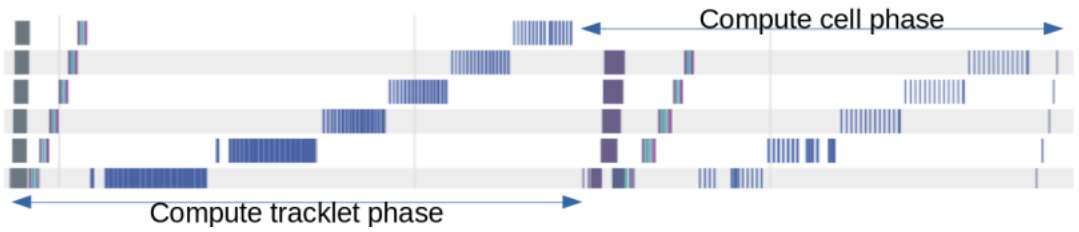

Figure 4.5: Distribution of tracklet and cell finding phases over different command queues using sort version of the OpenCL porting. The image has been obtained using Intel Intercept Layer application.

there is no guarantee that each instance of the kernel saves the tracklets found in an empty slot of the vector; to avoid tracklets overlap, an atomic function is necessary, in particular `atom_inc(var)` increments a variable `var` (shared among all the work-items of same layer) and it returns the vector index where the tracklet must be saved. During the kernel execution, the lookup table for tracklets is filled with the number of the tracklets found for each cluster.

The second step is the inclusive scan of the tracklet lookup tables: when the compute tracklet step of *i-layer* is finished, the `compute::inclusive_scan()` function performs the scan of tracklet lookup table (that is necessary for cell compute phase).

Listing 4.1: Pseudocode of kernel used in the sort version for the computation of the tracklets

```
void computeTrackletKernel(int sharedIndex){
    int clusterIndex = get_global_id(0);

    Cluster c1 = clusterVector(clusterIndex);
    Vector<Cluster> compatibleClusters = getCompatibleClusters(c1);

    for(Cluster c2 : compatibleClusters){
        if(validTracklet(c1,c2)==true){
            int validIndex = atomic_increment(sharedIndex);
            trackletVector[validIndex]= newTracklet(c1,c2);
        }
    }
}
```

At the end, `compute::stable_sort()` function perform a sorting algorithm over the vectors of clusters filled during the first step of this phase. By using the Intel Intercept Layer (fig.4.6) it is easy to notice how the sort function provided by the Boost Compute library is a blocking function and that no parallelism is possible at this point with a sizeable decrease of performance.

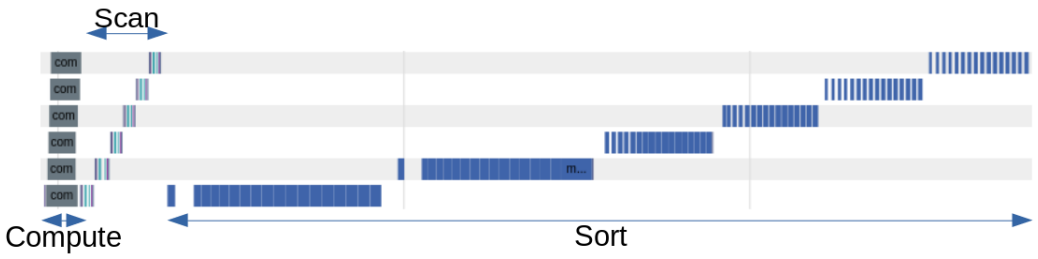

Figure 4.6: Distribution over different command queues of tracklet finding phase of a single event executed using sort version.

The compute cell phase is very similar to the previous one. A layer by layer execution is performed to guarantee a high level of parallelism. So, when the sorting algorithm has finished, the three steps (compute, inclusive scan and sorting) for cell finding can start. After the end of the sorting phase, a copy from `compute::vector` (which host sorted cells) to `std::vector` is necessary in order to make available the result for the following phases (in particular for the cell neighbourhood finding) that are executed without OpenCL. Figure 4.7 shows the distribution of the cell finding phase over the command queues; also in this case the serialization of the sort step, derived from the Boost Compute implementation, causes a degradation of the performance.

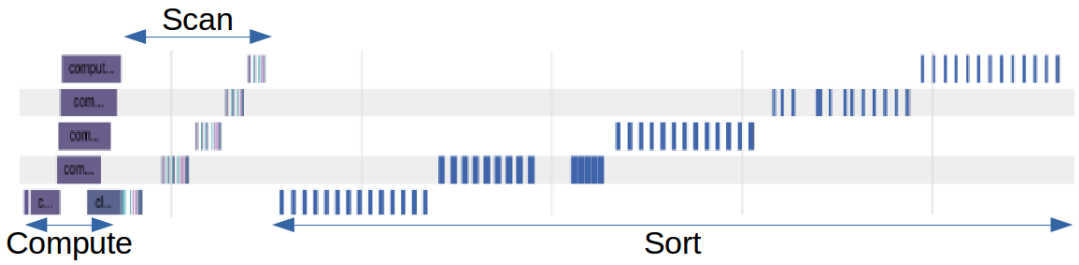

Figure 4.7: Distribution over different command queues of the cell finding phase for a single event executed using the sort version.

The main difference with respect to the CUDA implementation is the type of scan used: in particular, for CUDA implementation, an exclusive sum has been chosen, instead I decided to use an inclusive sum.
The performance of `compute::inclusive_scan()` and `compute::exclusive_scan()` are very similar but the inclusive scan stores the total number of tracklets/cells found on the last element of the vector and this number is useful in the next step of the algorithm.

Differently from native version and boost version, the sort version of the OpenCL porting implementation, processes the data only once and this is the real advantage

with respect to the other versions. In order to allow this single iteration, an atomic variable (inside the compute tracklets/cells kernel) and a sort step are necessary. Those steps represent a sort of bottleneck for the algorithm. This has a large impact on performance: the atomic variable forces the kernel to a serial-like execution; the `compute::stable_sort()` is slower than a second data processing (as in native and boost versions).

Listing 4.2: Pseudocode of kernel used by sort version for the compute of cells

```
void computeCellKernel(int sharedIndex){
    int trackletIndex = get_global_id(0);

    Tracklet t1 = trackletVector[clusterIndex];
    Vector<Tracklet> compatibleTracklets = getCompatibleTracklets(t1);

    for(Tracklet t2 : compatibleTracklets){
        if(validCell(t1,t2)==true){
            int validIndex = atomic_increment(sharedIndex);
            cellVector[validIndex]= newCell(c1,c2);
        }
    }
}
```

4.3 Native Version

The native version of the OpenCL porting tries to bypass the limits of the sort version with a different resolution schema. Also in this case three steps are executed over the events used as input:

- counting of the tracklets/cells
- scan of tracklet/cell lookup table
- computation of the tracklets/cells

The main difference with respect to the sort version is the second iteration over the same data in order to avoid the use of atomic operation during the first step and to avoid the use of the sort step.

Figure 4.8 shows the whole execution flow distributed over the command queues.

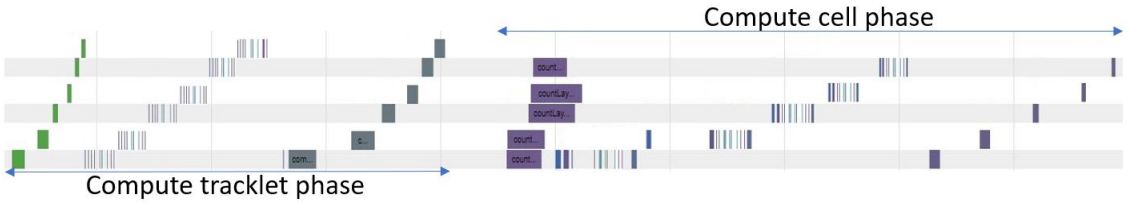

Figure 4.8: Complete execution flow distributed over the command queues using the native version of OpenCL porting.

4.3.1 Implementation details

The initialization phase, differently from what is done in the sort version, is implemented using only OpenCL API and all the `compute::objects` are replaced with `cl::objects`. The use of native OpenCL object has an important impact on the initialization phase performance. Similarly to the sort version, all the auxiliary buffers are created at this moment. Figure 4.9 shows the initialization time for 100 Pb-Pb events using this version.

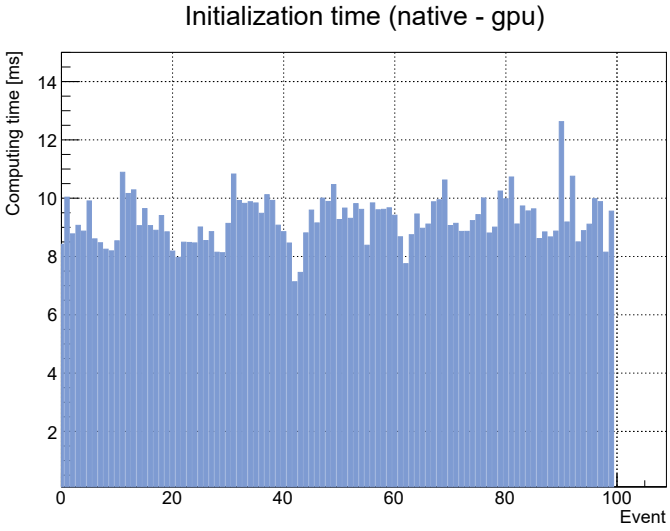

Figure 4.9: Initialization time for 100 Pb-Pb events using native version.

The first step for booth phases is radically different with respect to the sort version. Instead of calculating and storing the tracklets/cells on the wrong position of the relative vector, and then apply a sort operation, in this phase only the filling of the lookup table is performed. In this way no atomic operations are needed because *i-element* of the lookup table is filled with the number of tracklets/cell found for *i-cluster/tracklet*. The kernel pseudocode of the counting step is reported in 4.3

and 4.5 and the result show that, differently form what happen on sort version, no elements are calculated and stored.

After the end of the counting step, an inclusive scan operation is performed for each lookup table and the last element of the table corresponds to the effective total number of tracklets/cells found for the corresponding layer.

The last step (pseudo code of kernels can be found in 4.4 and 4.6), computes tracklets/cells and then it uses the lookup table calculated in the previous step to store the computed elements in the correct vectors position. The *i-element* of the lookup table specifies the starting offset inside the vector of tracklets/cells where to put the calculated element, so, also in this case, no atomic variable or particular synchronization mechanism are necessary. The process to calculate tracklets/cells is very similar to the first step (elements counting) but, at the end of the kernel execution, the new object is calculated and stored.

Listing 4.3: Pseudocode of kernel used by boost version and native version for the count of tracklets

```
void countTrackletKernel(){
    int clusterIndex = get_global_id(0);
    int trackletsFound = 0;

    Cluster c1 = clusterVector[clusterIndex];
    Vector<Cluster> compatibleClusters = getCompatibleClusters(c1);

    for(Cluster c2 : compatibleClusters){
        if(validTracklet(c1,c2)==true)
            trackletsFound++;
    }

    trackletLookupTable[clusterIndex]=trackletsFound;
}
```

Listing 4.4: Pseudocode of kernel used by boost version and native version for the compute of tracklets

```
void computeTrackletKernel(){
    int clusterIndex = get_global_id(0);
    int firstIndex =trackletLookupTable[currentClusterIndex-1];

    Cluster c1 = clusterVector(clusterIndex);
    Vector<Cluster> compatibleClusters = getCompatibleClusters(c1);

    for(Cluster c2 : compatibleClusters){
```

```
        if(validTracklet(c1,c2)==true){
            trackletVector[firstIndex++]= newTracklet(c1,c2);
        }
    }
}
```

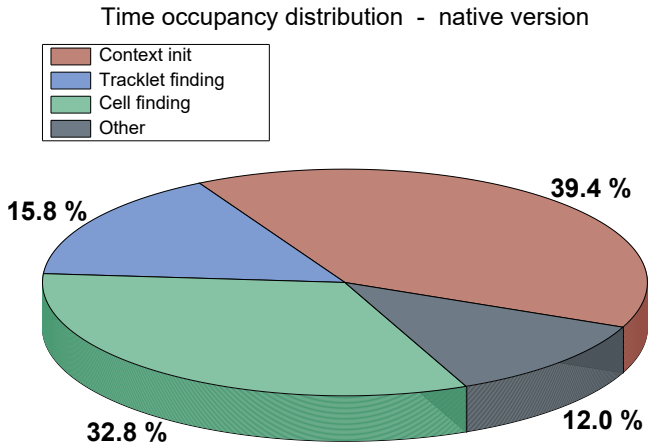

Figure 4.10: Time execution distribution of different phases using native version.

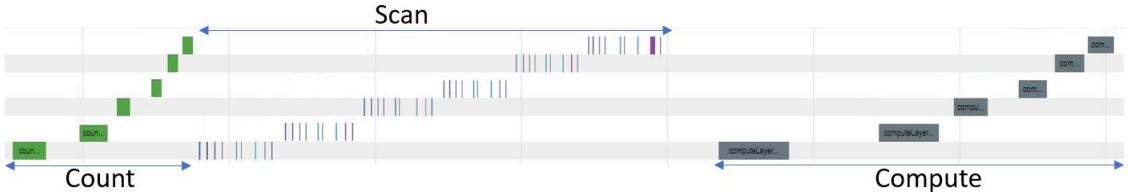

Figure 4.11: Distribution over different command queues of the tracklet finding phase for a single event executed using the native version.

At the end of the cell finding phase a final copy from device memory to host memory (in particular to a C++ `std::vector<Cell>`) is necessary to make available the computed cells for the following phases of the algorithm.

The algorithm used is radically different from the CUDA implementation: two iterations are performed on the data in order to increase the performance avoiding the use of atomic operations and sorting processes.

Figure 4.10 shows how total execution time is distributed among the implemented phases. The main limit of this implementation is the slow initialization phase derived

Figure 4.12: Distribution over different command queues of the cell finding phase for a single event executed using the native version.

from the use of native OpenCL API instead of Boost Compute: as seen before, Boost Compute is especially useful to handle vectors with variable size, and this is the case. Figs. 4.11 and 4.12 show the distribution of the tracklet finding phase and the cell finding phase over the command queues.

Listing 4.5: Pseudocode of kernel used by boost version and native version for the count of cells

```
void countCellKernel(){
    int trackletIndex = get_global_id(0);
    int cellsFound = 0;
    Tracklet t1 = trackletVector[clusterIndex];
    Vector<Tracklet> compatibleTracklets = getCompatibleTracklets(t1);

    for(Tracklet t2 : compatibleTracklets){
        if(validCell(t1,t2)==true)
            cellsFound++;
    }
    cellLookupTable[trackletIndex]=cellsFound;
}
```

Listing 4.6: Pseudocode of kernel used by boost version and native version for the compute of cells

```
void computeCellKernel(){
    int trackletIndex = get_global_id(0);
    int cellsFound = 0;
    Tracklet t1 = trackletVector[clusterIndex];
    Vector<Tracklet> compatibleTracklets = getCompatibleTracklets(t1);

    for(Tracklet t2 : compatibleTracklets){
        if(validCell(t1,t2)==true){
            cellVector[firstIndex++]= newCell(t1,t2);
        }
```

```
    }
}
```

4.4 Boost Version

The Boost version of the porting tries to combine the advantages of the sort and the native versions. The limit of the native version are bypassed with an extensive use of Boost Compute library (in particular for the implementation phase) which makes more clear all the operations of copy between the host memory and the device memory. The bottleneck of the sort version are removed using two iterations over the same event instead of the sort step.

The general schema is the same of native version and it is composed by:

- counting of tracklets/cells
- scan of tracklet/cell lookup Table
- computing of tracklets/cells

The time distribution of each phase is shown in fig. 4.13. It is clear from the results that the initialization is the more expensive step (51.1% of the total time). This result has been achieved thanks to the improvement of the tracklet and the cell finding phases which take only about 30% of the total execution time.

Looking at fig. 4.14 it is easy to identify each phase and the distribution of the execution over the different command queues used for this version. In particular it is clear how the scan phase is the real limit of this version.

4.4.1 Implementation details

The inizialization phase is substantially the same of the one used in the sort version except for small differences needed for the implementation change of the following phases. Thanks to Boost Compute library all the variable size vectors are created only once and than resized when necessary. To speedup and obtain the highest level of parallelism all the operations are executed on the command queue relative to the proper layer. Figure 4.15 represents the initialization time for all the processed events: the peaks correspond to a resize operation.

The count (4.3) and the compute (4.4) of tracklets are identical to native version except for the use of `compute::objects` instead of `cl::object`.

The scan phase is slightly different. As shown in figs. 4.16 and 4.17 the scan phase has a very important impact on the performance also if the real execution time necessary for scan operation is very limited, so it is important to speedup this step.

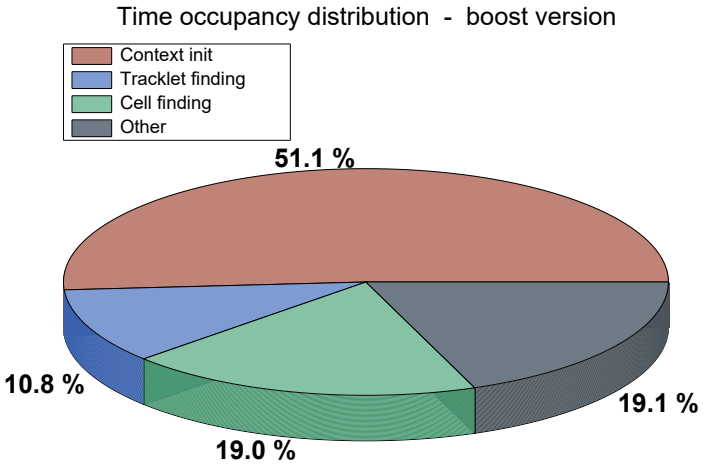

Figure 4.13: Time execution distribution of different phases using boost version.

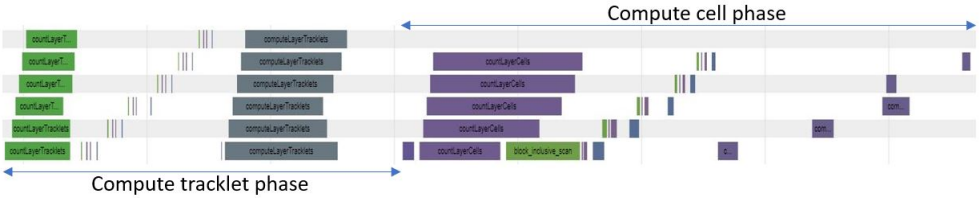

Figure 4.14: Time execution distribution of different phases using boost version.

Boost Compute implementation of this function is not the best choice: every time a `compute::inclusive_scan` is called the relative kernels are created and compiled. A better alternative is the use of a different library, VexCL[29].

VexCL is a C++ vector expression template library for OpenCL and CUDA; it provides some built in function for parallel operation over vectors, in particular, the `vex::inclusive_scan()` is helpful in our case. VexCL is compatible with native OpenCL API and with Boost Compute library and small adjustments must be applied to use VexCL scan with `boost::vector`. The figure 4.18 shows the difference of performance between `vex::inclusive_scan()` and `compute::inclusive_scan()`.

4.5 Performance comparison

In this chapter I try to summarize the main differences between the three OpenCL porting versions I have developed and the final performances obtained with each one are presented.

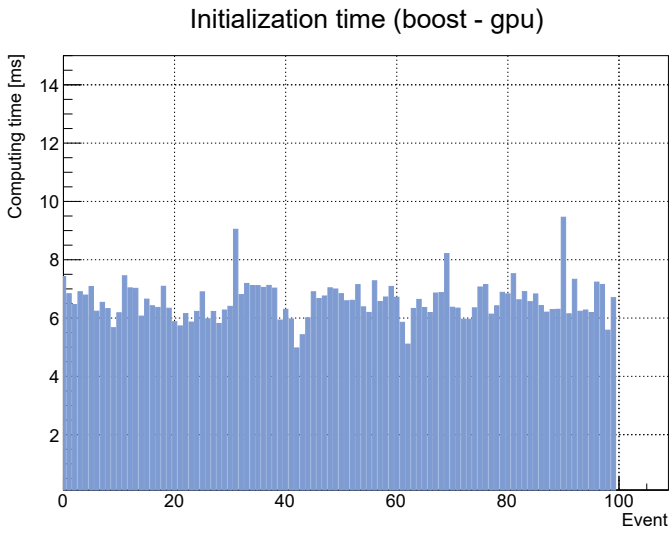

Figure 4.15: Initialization time using boost version over 100 Pb-Pb events.

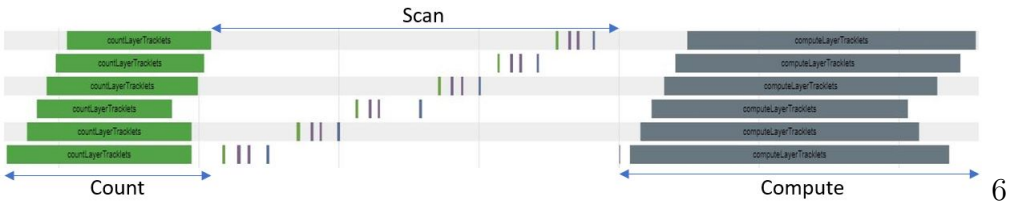

Figure 4.16: Distribution over command queues of compute tracklet phase of a single Pb-Pb event using boost version.

From the data initialization point of view, the main difference is due to the usage of the Boost Compute library. Figure 4.19 shows the important impact that the usage of the boost library has on the algorithm performance; the native version, which does not use boost, has a higher initialization time with respect to other versions. In the analysis of 100 Pb-Pb events, the boost and the sort versions have comparable initialization time (mean value of 6.6 ms and 6.4 ms, respectively) which are about 30% faster than the native one (mean time of 9.26 ms)

As shown in figure 4.20 the results in terms of performance are different for the tracklet finding phase. The use of atomic variable and especially the sort operations made the sort version the slowest among the three with a mean time of 12.13 ms. The native version, with 3.70 ms of mean time is about ∼70% faster than the sort version. The quickest solution is the boost version with a mean time of 1.38 ms (∼88% faster than the sort version and ∼62% faster than the native one).

The cell finding phase (fig.4.21) has a similar behavior, but with reduced differences between the three versions: native version, with mean time of 7.69 ms, is

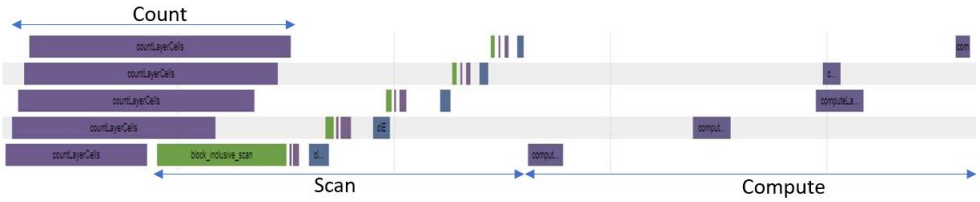

Figure 4.17: Distribution over command queues of compute cell phase of a single Pb-Pb event using boost version.

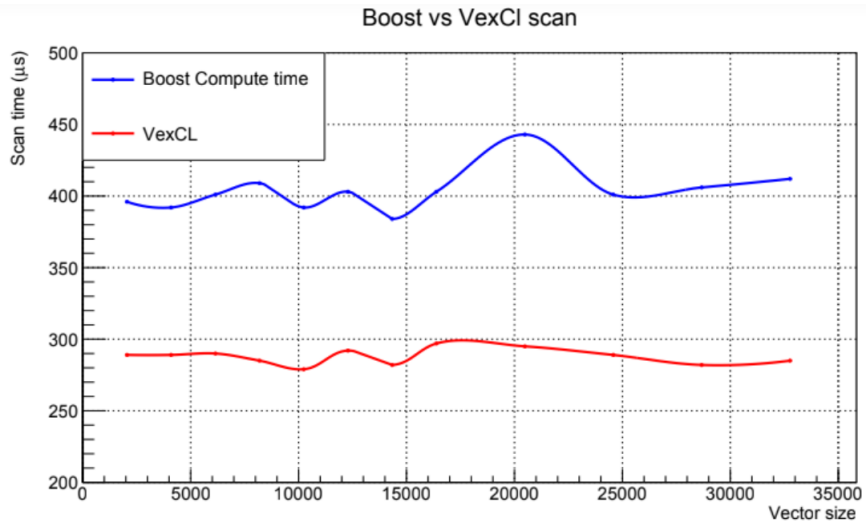

Figure 4.18: Execution time of `vex::inclusive_scan()` `compute::inclusive_scan()` over variable size vectors of integer. The execution time is calculated over a second iteration of the scan in order to better simulate the tracking algorithm case. The system configuration used is reported in 4.5.1

∼28% faster than the sort version (mean time of 10.70 ms); boost version, with 2.49 ms, is ∼76% faster than the sort version and ∼67% than the native one. The difference of performance of the sort version is strictly linked to the Boost Compute implementation of the sort operation which represents the main limit of all the versions. The difference between native and boost versions depends on the usage of the boost library for memory management and copy operations and especially on the usage of `vex::inclusive_scan()` instead of `{compute::inclusive_scan()`.

Finally, figure 4.22 shows the total execution time of the three versions over 100 Pb-Pb events. The results show that the better solution is the boost version with a mean time of 12.88 ms (∼69% faster than the sort version and ∼60% faster than the native one) while the native version has a mean time of 23.46 ms and the sort version has a mean time of 31.78 ms. Summing up all the results about the execution time

of initilization, tracklet finding and cell finding, it is clear that the next step of the algorithm takes a very small time to be executed, generally ∼1.1 ms.

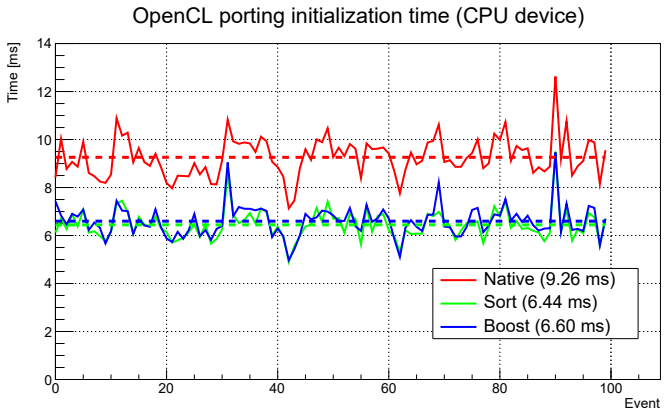

Figure 4.19: Initialization time of the three different version of OpenCL porting. Dashed line represents the mean value.

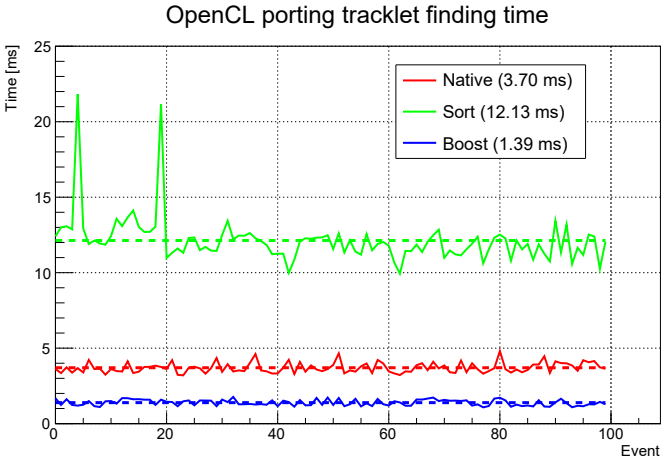

Figure 4.20: Tracklet finding phase execution time time of the three different version of OpenCL porting. Dashed line represents the mean value.

4.5.1 CPU performance

As explained in Cap.3, one of the main advantage derived by the usage of OpenCL is the ability to run the program not only on GPUs but also on CPUs. In order to evaluate the performance on CPU device I use the same hardware used for the tests performed on GPU:

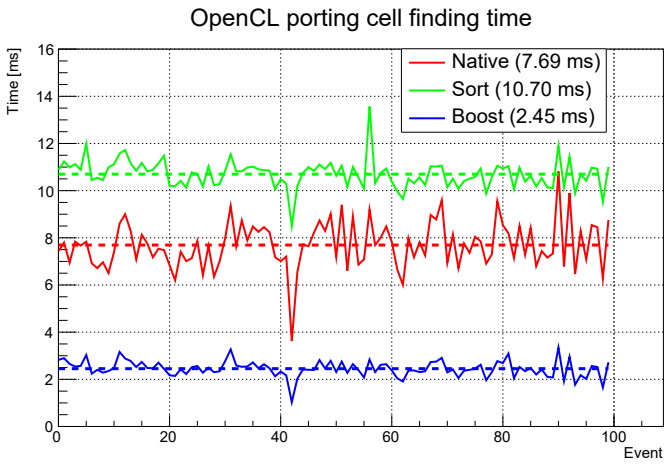

Figure 4.21: Cell finding phase execution time time of the three different version of OpenCL porting. Dashed line represents the mean value.

- CPU: i7-8700 CPU @ 3.20GHz
- RAM: Corsair DDR4 16GB (2x8) 3000MHz
- OS: Linux Ubuntu 16.04 LTS
- C++ Compiler: gcc version 6.4.0, with -O3 optimization flag
- OpenCL 1.2

Figure 4.23 shows how the native version initialization time is significantly longer (slower step) on CPU with respect to the execution on GPU. Instead, native and boost versions have comparable values when they are executed on CPU and GPU.

The total execution time (fig.4.24), describes how the performance on CPU are worse than those achieved on GPU for all the developed versions. The overall performance decrease depends mainly on the algorithm schema (designed to be specific for GPU) which does not perfectly fit with CPU devices. Also the implementation of boost/vexcl functions (scan and sort, in this case) slow down the execution; indeed booth libraries adapt the implementation of functions to the device where it will be executed. In particular, the boost version is 125% slower on CPU with respect to the execution on GPU and the native and the sort version are 38% and 35%, respectively, slower on CPU.

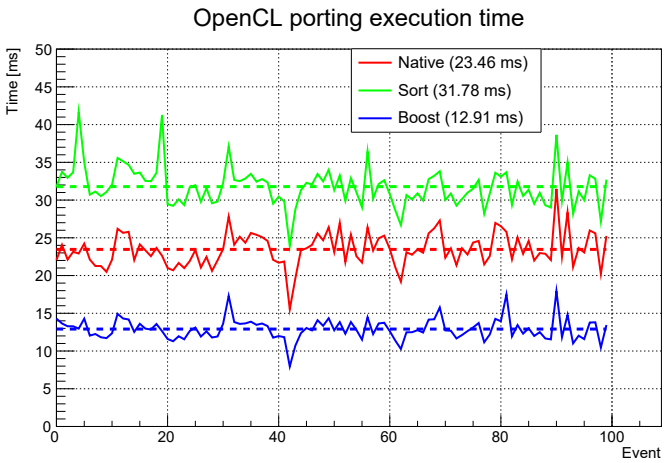

Figure 4.22: Total execution time time of the three different version of OpenCL porting. Dashed line represents the mean value.

Table 4.1: OpenCL implementation executed on NVIDIA TITAN Xp: min, mean and max time (in ms) for execution of 100 Pb-Pb events.

Version		Native	Sort	Boost
Context init	min	7.1	4.9	5.0
	mean	9.3	6.4	6.6
	max	12.6	9.5	9.4
Tracklet finding	min	3.1	9.9	1.0
	mean	3.7	12.1	1.4
	max	4.8	21.8	2.0
Cell finding	min	3.6	8.5	1.0
	mean	7.7	10.7	2.5
	max	10.8	13.6	3.8
Total	min	15.5	23.9	7.9
	mean	23.5	31.8	12.9
	max	31.5	41.6	17.9

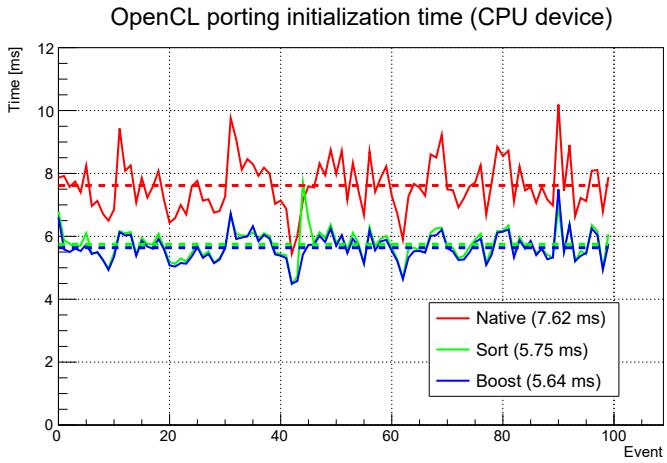

Figure 4.23: Initialization time of 100 Pb-Pb event with different OpenCL porting versions. Dashed line represents the mean value, dotted line represents the mean value of the relative GPU version.

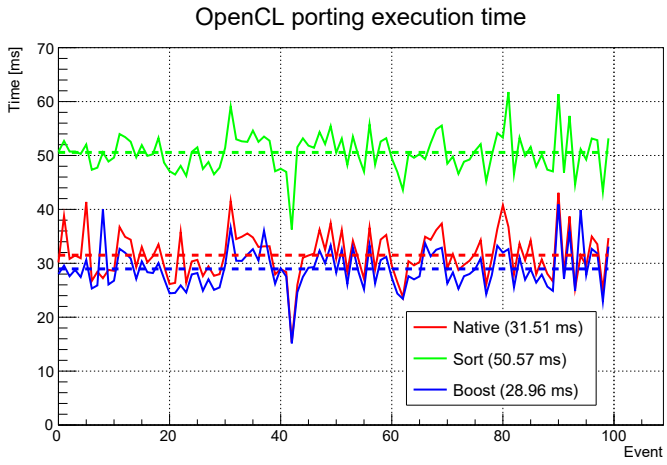

Figure 4.24: Total execution time of 100 Pb-Pb event with different OpenCL porting versions. Dashed line represents the mean value, dotted line represents the mean value of the relative GPU version.

Table 4.2: OpenCL implementation executed on CPU device: min, mean and max time (in ms) for execution of 100 Pb-Pb events.

Version		Native	Sort	Boost
Context init	min	5.4	4.5	4.5
	mean	7.6	5.7	5.6
	max	10.2	7.7	7.5
Tracklet finding	min	2.5	20.4	2.0
	mean	4.2	27.3	3.7
	max	5.5	36.6	8.9
Cell finding	min	7.0	10.9	8.2
	mean	17.9	15.8	17.8
	max	26.7	19.4	28.8
Total	min	15.4	36.3	15.1
	mean	31.5	50.7	29.0
	max	43.0	61.8	40.2

Conclusion

From January 2019, ALICE, one of the four experiments installed at the Large Hadron Collider (LHC) of the CERN laboratories, will be upgraded. The currently installed Inner Tracking System detector, dedicated to the tracking of the particles emitted in the collisions of protons and lead ions provided by the LHC, will be replaced with a new detector. The upgraded ITS will be composed by 7 layers of silicon pixels detectors, closer to the point where the particles circulating in the LHC collide. This upgrade allows the detector to operate with a read out rate of 50 kHz for Pb-Pb collisions and 2 MHz for p-p collisions, which are the collision rates expected during the Run 3 that will start in 2021. This increase of frequency causes an increment of data that must be managed up to 1 TB/s for Pb-Pb collisions. In this experimental conditions it is mandatory to minimize the execution time of all the algorithms involved in the data processing. In particular this thesis is dedicated to the optimization of the algorithm used to reconstruct the trajectory of the particles emitted in the LHC collisions and traversing the ITS (tracking code), using a parallel computing approach. The main goal of the present work was to obtain the best performance in terms of execution time using as starting point a tracking algorithm developed in [6].

From this algorithm, a CUDA parallel implementation has been also developed in [10]. This last version is characterized by an important boost of performance but also by a limited portability derived by CUDA constraints: only NVIDIA GPUs support the CUDA framework.

In my project, starting from the algorithm versions mentioned before, a third version of the code, based on OpenCL framework, has been developed. The usage of OpenCL allows writing a parallel program that can be executed on a large number of devices including CPUs, GPUs, FPGAs and others, without any constraint based on device's vendor.

In order to guarantee the compatibility with existing implementations, all the class interfaces are kept almost unchanged, managing the small necessary changes with a preprocessor directive schema. In this way, at compile time, it is possible to choose the desired version (serial, CUDA or OpenCL). Also the compatibility with the specific framework (the online-offiline O^2 of the ALICE experiment) where the

application will be executed has been maintained.

To this goal I have developed three different versions of OpenCL porting, all based on OpenCL 1.2 (because this version guarantees the maximum number of supported devices) and with Boost Compute library which brings advantages regarding performance and easiness of usage. The three versions are the following ones:

- sort: based on CUDA porting schema;
- native: based on two successive iterations over the same data, in order to avoid the use of atomic operations and sorting operations;
- boost: based on the native version schema but with a more extensive use of boost library and using VexCL library for scan operations.

One of the main limitation of heterogeneous programs is the loading of memory objects from and to device memory. In order to limit the impact of data creation and successive loading on device memory, a system of preallocation has been adopted using the size calculated for CUDA version[10]. As shown in Table 4.3, the boost

Table 4.3: Execution time for all the main phases of the three versions of OpenCL porting. The serial and CUDA porting execution times are reported in the last column. The times obtained using CPU device are reported between brackets.

Version		Native	Sort	Boost	Serial	CUDA
Context init	min	7.1 (5.4)	4.9 (4.9)	5.0 (4.5)	4.5	5.7
	mean	9.3 (7.6)	6.4 (5.7)	6.6 (5.6)	5.7	7.2
	max	12.6 (10.2)	9.4 (7.7)	9.4 (7.5)	7.6	10.4
Tracklet find.	min	3.1 (2.5)	9.9 (20.4)	1.0 (2.0)	23.5	1.7
	mean	3.7 (4.2)	12.13 (27.3)	1.4 (3.7)	51.6	2.3
	max	4.8 (5.5)	21.8 (36.6)	1.7 (8.9)	74.1	3.1
Cell find.	min	3.6 (7.0)	8.5 (10.9)	1.0 (8.3)	11.5	2.3
	mean	7.7 (17.9)	10.7 (15.8)	2.5 (17.8)	31.8	3.7
	max	10.8 (26.7)	13.6 (19.3)	3.3 (28.9)	49.5	5.8
Total	min	15.5 (15.4)	23.9 (36.3)	7.9 (15.1)	40.0	9.3
	mean	23.5 (31.5)	31.8 (50.6)	12.9 (29.0)	92.0	16.2
	max	31.5 (43.1)	41.6 (61.8)	17.9 (40.9)	134.7	23.9

version guarantees the best performance. In particular, using a GPU device, the boost version is 1.25 times faster than the CUDA porting, and 7.13 times faster than the serial version. The same version, using a CPU device is 2.3 times slower than using GPU device, and 3.1 times faster than the serial version. From this

results one can conclude that the algorithm used fits better on GPU device. Figure 4.25 shows that the most expensive phase is the initialization step where the data are loaded into the device memory; this phase occupies more than half of total execution time, instead the other two parallelized phases occupy 30% of total time. Figure 4.26 shows that the OpenCL porting (the boost version in particular) has an efficiency very similar to serial implementation.

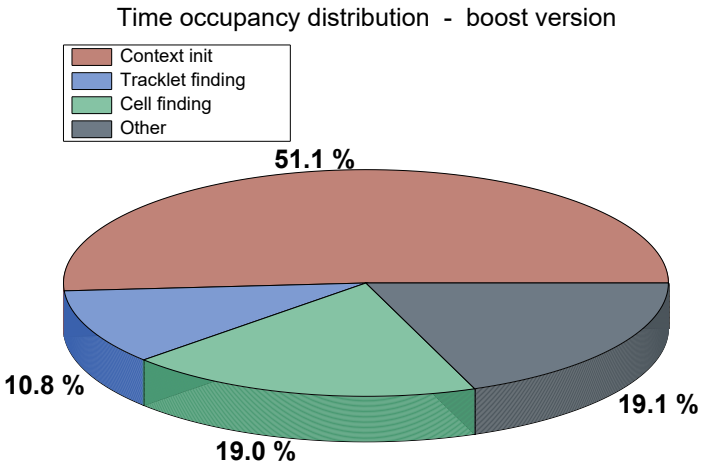

Figure 4.25: Distribution of execution time over the main phases of the boost version of OpenCL porting.

In the future, some improvement can be achieved with additional work. In particular, the code fits well with a multithreading implementation of the tracker: each of the seven ITS layers can be handled by a single thread independently from the other threads for both tracklet and cell finding phases. Further improvements can be reached using a more recent version of OpenCL. In particular, version 2.0 and 2.1 implement a built-in inclusive scan function that can lead to a better performance with respect to the Boost/VexCL implementation. Finally, analyzing the results obtained with different version of the OpenCL porting, a CUDA porting based on the schema used for the OpenCL boost version (sop without atomic operation and sorting phase), may give the opportunity to improve the general performance.

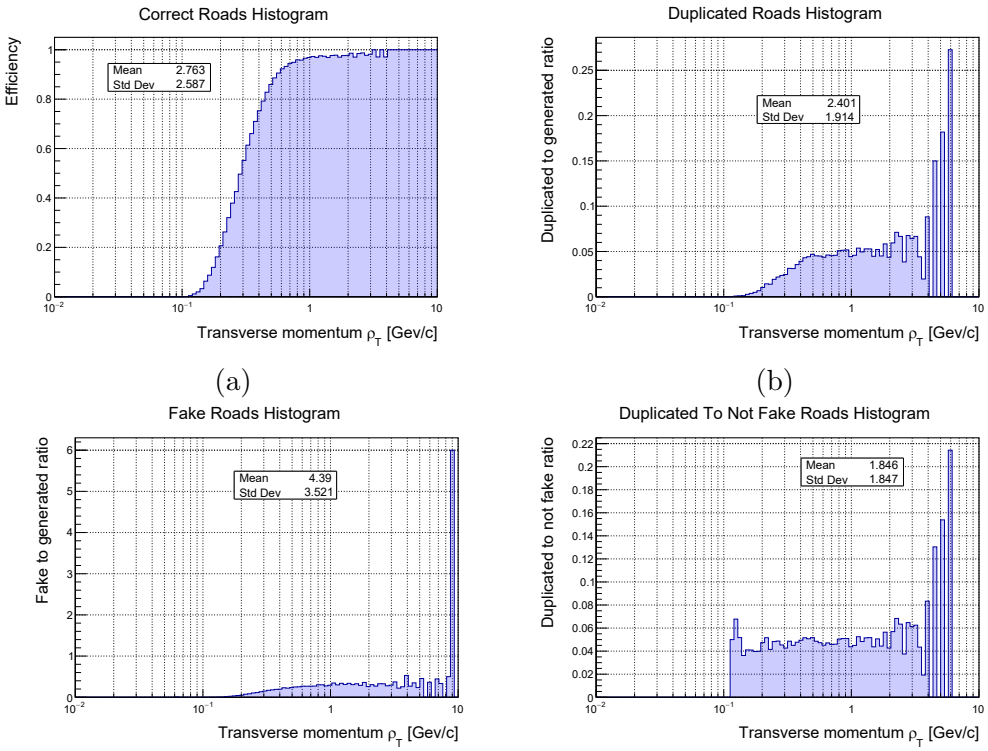

Figure 4.26: Efficiency evaluation of boost version of OpenCL porting over transverse momentum for a sample of 100 central Pb–Pb events without pile-up obtained dividing the number of reconstructed roads over the total number of generated roads.

Acronyms

ALICE A Large Ion Collider Experiment. 1

API Application Programming Interface. 32

ATLAS A Toroidal LHC Apparatus. 1

CA Cellular Automata. 12

CERN Conseil Européen pour la Recherche Nucléaire. 1

CMS Compact Muon Solenoid. 1

CPU Central Processing Unit. 27

CTF Compressed Time Frame. 11

CUDA Compute Unified Device Architecture. 18

DCA Distance of Closest Approach. 16

DSP digital signal processor. 27

EMCa Electromagnetic Calorimeter. 4

EPN Event Processing Nodes. 12

FLP First Level Processor. 11

FPGA Field Programmable Gate Array. 11, 27

GPGPU General-Purpose computing on Graphics Processing Units. 40

GPU Graphic Processing Unit. 11, 27

GUI Graphic User Interface. 10

HIJING Heavy-Ion Jet Interaction Generator. 10

HMPID High Momentum Particle Identification. 4

IEEE Institute of Electrical and Electronic Engineers. 28

IP Interaction Point. 2

JSON JavaScript Object Notation. 37

LEIR Low Energy Ion Ring. 1

LHC Large Hadron Collider. 1

LHCb Large Hadron Collider beauty. 1

LINAC Linear Accelerator. 1

LS2 Long Shutdown 2. 5

MC Monte Carlo. 7

OpenCL Open Computing Language. 27

PEs Processing Elements. 29

PHOS Photon Spectrometer. 4

PS Proton Synchrotron. 1

PSB Proton Synchrotron Booster. 1

QGP QuarkGluon Plasm. 1

SDD Silicon Drift Detector. 4

SFM String Fusion Mode. 10

SPD Silicon Pixel Detector. 4

SPS Super Proton Synchrotron. 1

SSD Silicon Strip Detector. 4, 5

STF Sub-Time Frame. 11

TF Time Frame. 11

TOF Time of Flight. 4

TPC Time Projection Chamber. 4

TRD Transition Radiation Detector. 4

WLCG Worldwide LHC Computing Grid. 12

Bibliography

[1] Esma Mobs. "The CERN accelerator complex. Complexe des accélérateurs du CERN". In: (July 2016). General Photo. URL: https://cds.cern.ch/record/2197559.

[2] K. Aamodt et al. "The ALICE experiment at the CERN LHC". In: *JINST* 3 (2008), S08002. DOI: 10.1088/1748-0221/3/08/S08002.

[3] L. Betev and P. Chochula. *Definition of the ALICE Coordinate System and Basic Rules for Sub-detector Components Numbering.* [accessed 2018-05-24]. URL: https://edms.cern.ch/document/406391.

[4] B Abelev et al. "Technical Design Report for the Upgrade of the ALICE Inner Tracking System". In: *J. Phys.* G41 (2014), p. 087002. DOI: 10.1088/0954-3899/41/8/087002.

[5] Giuseppe E Bruno. *Prospettive di fisica con l'upgrade dell'apparato ALICE ad LHC.* Accessed: 2018-05-30. URL: https://www.sif.it/static/SIF/resources/public/files/congr14/ip/Bruno.pdf.

[6] Maximiliano Puccio. "Study of the production of nuclei and anti-nuclei at the LHC with the ALICE experiment". PhD thesis. Turin U., 2017. URL: http://www.infn.it/thesis/thesis_dettaglio.php?tid=11773.

[7] Y. Belikov et al. "TPC tracking and particle identification in high density environment". In: *eConf* C0303241 (2003), TULT011. arXiv: physics/0306108 [physics].

[8] https://root.cern.ch/. Accessed: 2018-05-30.

[9] http://alice-offline.web.cern.ch/AliRoot/Manual.html. Accessed: 2018-05-30.

[10] Iacopo Colonnelli. "Design of an high-performance tracking algorithm optimised for the Inner Tracking System of the ALICE experiment". PhD thesis. Politecnico di Torino, 2017. URL: http://www.infn.it/thesis/thesis_dettaglio.php?tid=12030.

[11] P. Buncic, M. Krzewicki, and P. Vande Vyvre. "Technical Design Report for the Upgrade of the Online-Offline Computing System". In: (2015).

[12] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*. Version 3.2. 2010.

[13] https://github.com/AliceO2Group/CodingGuidelines. Accessed: 2018-05-30.

[14] https://www.khronos.org/opencl/. Accessed: 2018-05-30.

[15] Kamran Karimi, Neil G. Dickson, and Firas Hamze. "A Performance Comparison of CUDA and OpenCL". In: arXiv:1005.2581 (May 2010).

[16] https://www.amd.com. Accessed: 2018-05-30.

[17] http://www.nvidia.it/page/home.html. Accessed: 2018-05-30.

[18] https://www.qualcomm.com/. Accessed: 2018-05-30.

[19] https://www.ibm.com. Accessed: 2018-05-30.

[20] https://www.intel.it/content/www/it/it/homepage.html. Accessed: 2018-05-30.

[21] https://www.khronos.org/. Accessed: 2018-05-30.

[22] https://www.khronos.org/news/press/khronos-releases-opencl-2.2-with-spir-v-1.2. Accessed: 2018-05-30.

[23] Aaftab Munshi et al. *OpenCL Programming Guide*. 1st. Addison-Wesley Professional, 2011. ISBN: 0321749642.

[24] https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf. Accessed: 2018-05-30.

[25] https://github.com/intel/opencl-intercept-layer. Accessed: 2018-05-30.

[26] https://wiki.aalto.fi/download/attachments/40025977/Cuda+and+OpenCL+API+comparison_presented.pdf. Accessed: 2018-05-30.

[27] https://github.com/boostorg/compute. Accessed: 2018-05-30.

[28] https://cmake.org. Accessed: 2018-05-30.

[29] https://github.com/ddemidov/vexcl. Accessed: 2018-05-30.