Alma Mater Studiorum · University of Bologna

Department of Physics and Astronomy "Augusto Righi"
Bachelor Degree in Physics

A new algorithm for the online pixel vertex reconstruction at the CMS Experiment

Supervisor:
Prof. Francesco Giacomini

Co-Supervisors:
Dr. Felice Pantaleo, CERN
Dr. Simone Balducci, CERN

Submitted by:
Jonayed Ahmed

Academic Year 2024/2025

Abstract

Negli esperimenti di fisica delle alte energie al Large Hadron Collider al CERN, la ricostruzione delle traiettorie delle particelle cariche e la loro successiva associazione, vertexing, tramite le loro posizioni di massimo avvicinamento al fascio, sono fondamentali per la corretta interpretazione del processo fisico che ha avuto luogo nella collisione protone-protone. Il vertexing permette di discriminare i segnali provenienti dal vertice primario che potrebbero contenere della eccitante nuova fisica, dalle meno interessanti collisioni sovrapposte (pile-up).
Questa tesi presenta un nuovo algoritmo di vertexing per la ricostruzione online dell'esperimento CMS al CERN. L'algoritmo utilizza CLUEstering, una libreria di clustering pesato e density-based ad alte performance per clusterizzare le traiettorie di particelle cariche nel detector a Pixel e ricostruire così i *pixel-vertex*. CLUEstering è sviluppato utilizzando alpaka, una libreria che permette di scrivere codice in maniera portabile, i.e. eseguibile su una vasta gamma di processori, mantenendo prestazioni ottimali. Il nuovo algoritmo è stato completamente integrato nel software di simulazione e ricostruzione di CMS (CMSSW) ed è stato confrontato con l'algoritmo attualmente utilizzato in CMS su un dataset di simulazione di eventi contenenti coppie di quarks $t\bar{t}$ con 200 collisioni protone-protone sovrapposte.

In high energy physics experiments at the Large Hadron Collider at CERN, track reconstruction of charged particles and their subsequent association, known as vertexing, through their points of closest approach to the beam, is fundamental for the correct interpretation of the physical process that occurred in the proton-proton collision. Vertexing allows one to distinguish signals coming from the primary vertex, which may contain exciting new physics, from the less interesting overlapping collisions (pile-up).
This thesis presents a new vertexing algorithm for the online reconstruction in the CMS experiment at CERN. The algorithm uses CLUEstering, a high-performance, weighted, density-based clustering library, to cluster charged particle trajectories in the Pixel detector, thereby reconstructing *pixel vertices*. CLUEstering is developed using alpaka, a library that allows writing portable code, i.e. executable on a wide range of processors, while maintaining optimal performance. The new algorithm has been fully integrated into the CMS simulation and reconstruction software (CMSSW) and has been compared with the algorithm currently used in CMS on a simulated dataset of events containing quark pairs $t\bar{t}$ with 200 overlapping proton-proton collisions.

Contents

1 LHC and CMS

The LHC (Large Hadron Collider) is the largest and most powerful particle accelerator [1]. It accelerates two counter-rotating beams of protons, or heavy ions, which collide at four interaction points: CMS (Compact Muon Solenoid), ALICE [3] (A Large Ion Collider Experiment), ATLAS [2] (A Toroidal LHC Apparatus) and LHCb [4] (LHC beauty), as shown in Figure 1. When two beams collide, about 40 million times per second, they produce a spray of secondary particles that travel in many different directions. Most of these are well-known, relatively low-energy particles, but the collisions can also create much heavier and potentially new particles. These heavy states cannot be observed directly because they decay almost instantly into lighter particles, which may then decay further. To study such events, the four main LHC detectors are placed around the collision points. Their purpose is to detect, track, and measure the properties of all the outgoing particles so that the original event can be reconstructed and understood.

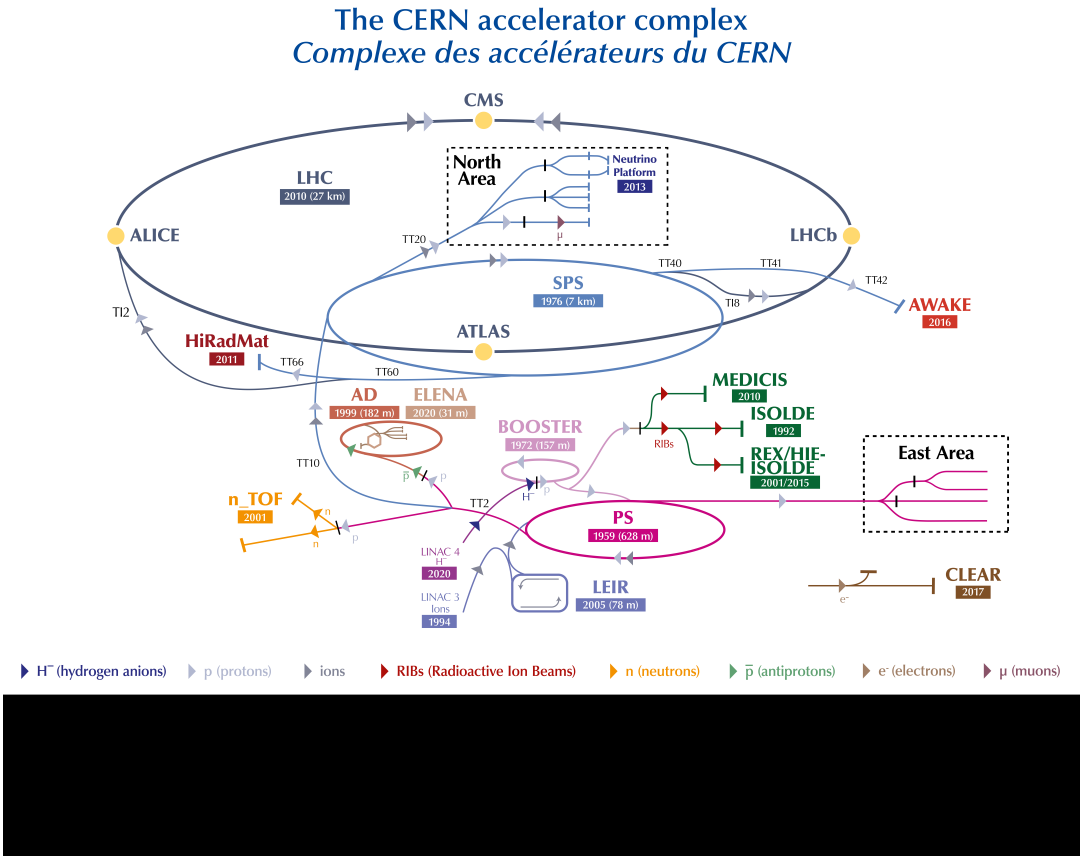

Figure 1: Accelerator Complex in 2022 [5]

1.1 The CMS Experiment

The Compact Muon Solenoid (CMS) detector sits at one of the four collision points of the LHC accelerator. It was designed to investigate the nature of the electroweak symmetry breaking, as well as studying the strong interaction, verifying the mathematical consistency of the Standard Model, extending the study of QCD matter, and exploring new symmetries, new forces, or constituents. It is made up of concentric detector layers (Fig. 2), wrapping around the beam direction, and employs the most powerful superconducting solenoid magnet ever made, with a magnetic field of 3.8 T. CMS can recreate a "picture" of the collisions by measuring the momenta and the energies of nearly all the stable particles produced. This is done by:

- Bending Particles: a strong magnet bends the charged particles as they move out from the collision point. This serves two purposes: identifying the charge of the particle, since opposite charged particles bend in opposite directions; and measuring the momentum, as high momentum particles bend less than low momentum ones.

- Identifying Tracks: in order to precisely identify the paths taken by the particles, a silicon tracker made of individual sensors arranged in concentric layers is employed. Charged particles interact electromagnetically with these sensors, producing hits, that are then joined together to reconstruct the track.

- Measuring Energy: understanding what happens at the collision point requires detailed information about the energies of the particles produced in each event. In the CMS detector, this energy data is gathered using two types of calorimeters. The Electromagnetic Calorimeter (ECAL), the inner layer, measures the energy of electrons and photons by fully absorbing them. Hadrons—particles composed of quarks and gluons—pass through the ECAL and are stopped by the outer layer, known as the Hadron Calorimeter (HCAL).

- Detecting Muons: the last type of particle that CMS can directly detect is the muon. Muons are part of the same particle family as electrons but are about 200 times more massive. Because they can pass through the calorimeters without being stopped, dedicated sub-detectors are required to track them as they travel through the CMS detector. These muon detectors are embedded within the return yoke of the solenoid. Thanks to CMS's powerful magnet, a muon's momentum can be measured both inside the superconducting coil using tracking systems and outside of it using the muon chambers.

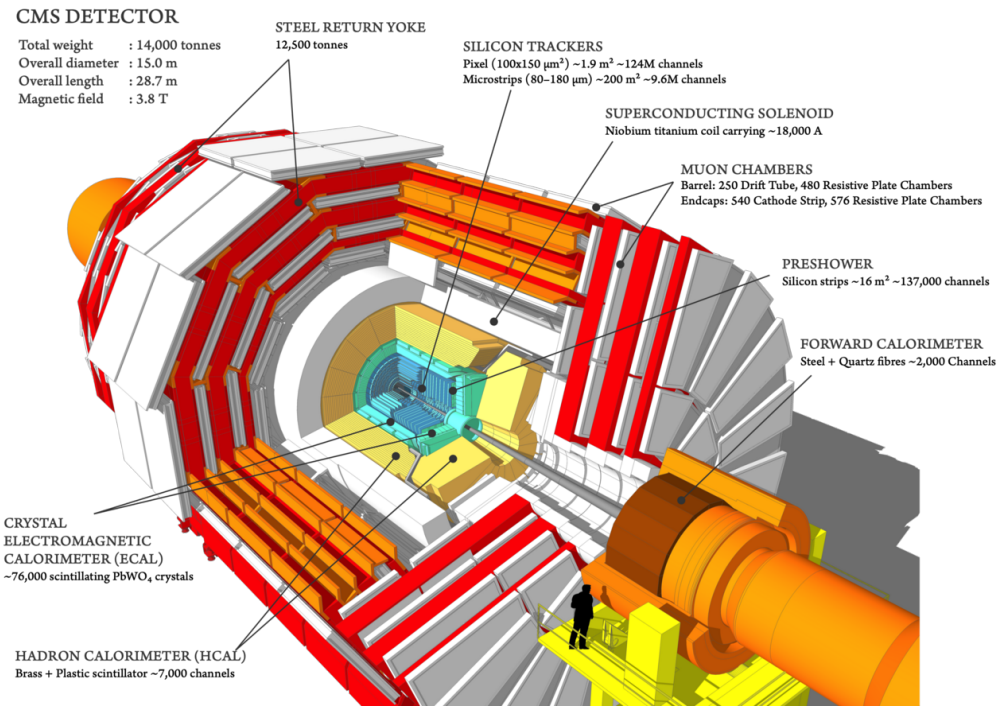

Figure 2: Diagram of the CMS detector

In Figure 3, the coordinate system used in CMS is shown. It is a **right-handed** system whose origin is set at the **nominal interaction point**. The x-axis points toward the *center of the LHC ring*, the y-axis points *upwards*, and the z-axis follows the *anticlockwise beam direction*.

The **azimuthal angle** ϕ is measured anticlockwise from the positive x-axis and ranges from $-\pi$ to $+\pi$. The **polar angle** θ is defined with respect to the positive z-axis.

Because the rest frame of the collision is not known, it is essential to define quantities that are *invariant under boosts along the beam axis*. For a particle with momentum $\vec{p}$, the measurable component in the transverse plane is the **transverse momentum**, $\vec{p}_T$.

Another important coordinate is the **pseudorapidity**, defined as

$$\eta = -\ln \tan \frac{\theta}{2}. \tag{1}$$

This variable ranges from $-\infty$ along the negative z-axis to $+\infty$ along the positive z-axis. It takes the value $\eta = 0$ at $\theta = 90°$, $\eta = 0.88$ at $\theta = 45°$, and $\eta = 2$ at $\theta = 15°$, illustrating its strong dependence on the polar angle.

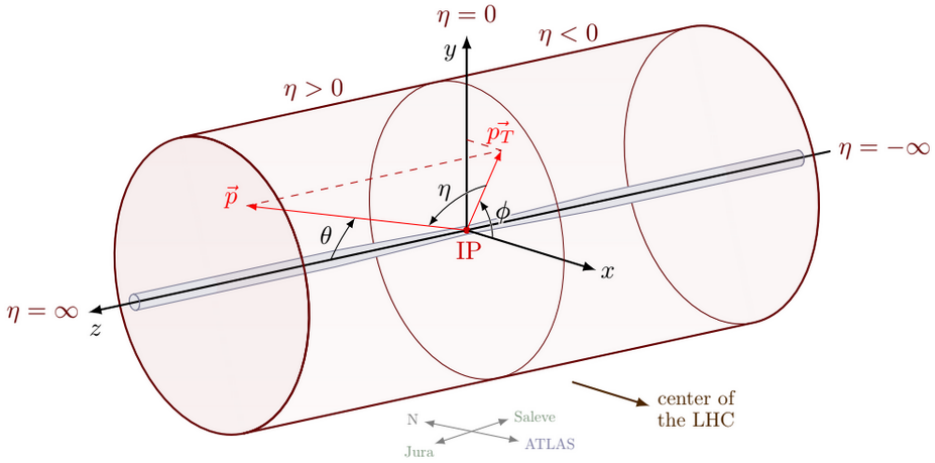

Figure 3: The CMS coordinate system is **right-handed**, with its **origin located at the nominal interaction point**. The x-axis points toward the *center of the LHC ring*, the y-axis points *upwards*, and the z-axis follows the *anticlockwise beam direction*. The **azimuthal angle** ϕ is measured *anticlockwise from the positive x-axis*, within the range $[-\pi, \pi]$, while the **polar angle** θ is measured *from the positive z-axis*. The **pseudorapidity** η extends from $-\infty$ along the *negative z-axis* to $+\infty$ along the *positive z-axis*.

1.2 Luminosity and Pileup

Luminosity is a central concept in collider physics, as it characterizes the performance of a particle accelerator in terms of its ability to produce collisions. It is formally defined as the proportionality factor between the number of collisions occurring per unit time dN/dt and the interaction cross section σ for a given process [6]:

$$\frac{dN}{dt} = \mathcal{L} \cdot \sigma \tag{2}$$

and its unit of measure is thus $cm^{-2}s^{-1}$. Another important operational parameter at the LHC is the so-called pileup (PU). Pileup refers to the number of simultaneous proton–proton interactions that take place in a single bunch crossing. Since the LHC operates with multiple proton bunches circulating at high frequency, each bunch crossing can give rise not to a single collision but to several, which are recorded on top of each other in the detector. The average number of such overlapping interactions is given by:

$$\langle PU \rangle = \frac{\mathcal{L}\sigma}{N_b f} \tag{3}$$

where N_b is the number of bunches, and f is the revolution frequency.
It is clear that these parameters quantify the number of events occurring every second,

and consequently the amount of data to analyze. The LHC entered Run-3 in July 2022, operating at its maximum collision energy of 13.6 TeV. During this period, the average pileup is 64 and the peak luminosity reaches approximately $2 \times 10^{34} cm^{-2} s^{-1}$ [7]. However, with the High Luminosity (HL) upgrade, it will reach $7 \times 10^{34} cm^{-2} s^{-1}$ with an average PU of 200 proton-proton collisions. To make the most of the higher luminosity, the CMS experiment will have to increase the readout rate from 100 kHz to 750 kHz, requiring larger processing power [8].

2 Vertexing in High Energy Physics

2.1 Introduction to Vertex Reconstruction

In high-energy physics experiments, a fundamental goal is to precisely determine the locations of particle collisions, and the decay points of unstable particles. Accurate vertex reconstruction is essential for several reasons. First, it allows for the identification and separation of multiple simultaneous proton-proton collisions which occur within the same bunch crossing. Second, the reconstructed vertex positions are critical inputs for tracking algorithms, enabling the association of particle trajectories with their corresponding interaction points. Third, vertex information is necessary for many physics analyses, such as the reconstruction of heavy-flavor hadrons, the measurement of lifetimes of short-lived particles, and the identification of jets originating from b-quarks (*b-tagging*). The challenge in vertex reconstruction arises from the large number of charged particle tracks produced in each collision, each measured with finite spatial resolution. Tracks originating from the same vertex are spatially correlated along the beam axis, but the presence of detector measurement uncertainties and overlapping events complicates the clustering process. Consequently, sophisticated algorithms are required to identify track groupings that correspond to real physical vertices while minimizing the inclusion of spurious combinations. In the context of the CMS Phase-1 pixel detector, which provides high-precision measurements of charged particle hits in three-dimensional space, vertex reconstruction typically begins with one-dimensional clustering along the beam axis (z-axis). The high granularity and near-hermetic coverage of the pixel detector allow for precise estimation of the z-coordinate of each track at the point of closest approach to the beam line. By grouping tracks based on their longitudinal proximity and iteratively refining these groups, vertexing algorithms can reconstruct both primary and secondary vertices with sub-millimeter resolution. The following sections describe the practical implementation of such an algorithm, focusing on the clustering of tracks along the z-axis and the handling of outliers to produce stable and accurate vertex candidates.

2.2 Pixel Detector and Track Reconstruction

The current CMS Phase-1 pixel detector consists of 1,856 silicon sensors, each comprising 66,560 individual pixels, which together correspond to a total active area of approximately 1.9 m^2. These sensors are arranged in a geometry designed to provide precise spatial resolution and tracking coverage: four concentric barrel layers (L1–L4 in Fig. 4) surround the beam line in the central region, while three disks (D1–D3 in Fig. 4) are placed on each end of the detector. The overlap between neighboring sensors in the azimuthal direction ensures nearly continuous coverage, often referred to as hermeticity, which is essential for reconstructing particle trajectories with high efficiency. Nevertheless, despite this careful design, there remain regions of incomplete coverage.

Specifically, small gaps appear along the beam direction, in the radial direction within the endcap regions, and in the transition zones between the barrel layers and the disks. These geometric limitations, while minimized as much as possible, can influence tracking performance and must be taken into account in both detector simulation and data analysis.

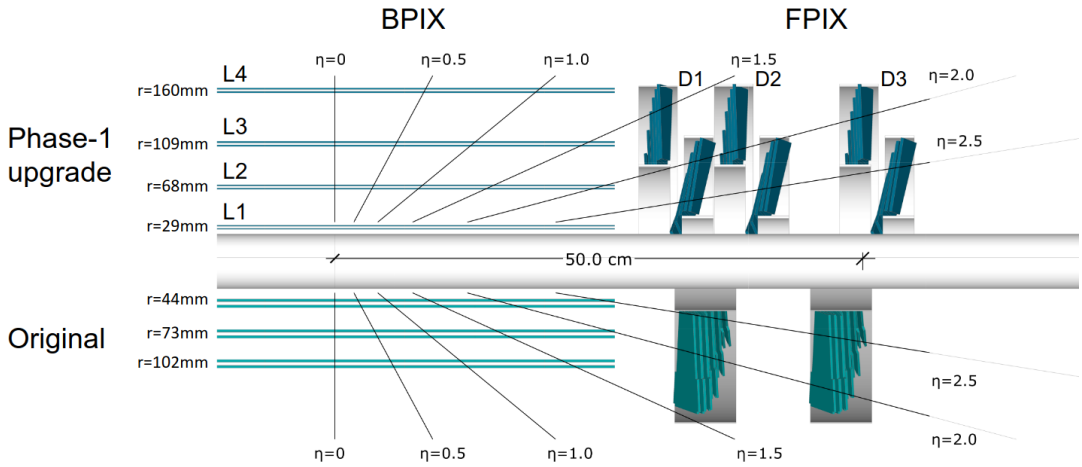

Figure 4: Layout of the CMS Phase-1 pixel detector compared to the original detector layout, in longitudinal view [11]

The reconstruction of tracks begins with a local reconstruction of the electrical signals generated as charged particles traverse the pixel detector. These raw signals are then interpreted and digitized into so-called *digis*. Each digi corresponds to a single pixel and contains essential information such as the measured charge and the position coordinates of the hit. To handle the large number of pixels efficiently, this process is highly parallelized: each detector module is assigned to an independent block of threads, and within each block, every digi is given a unique index. This index ensures that each digi can be processed by a separate thread without conflict, thereby exploiting the computational power of parallel architectures.
Once the digis have been generated, the clusterization step begins. At this stage, the digis are arranged on a two-dimensional grid that reflects the geometry of the pixel matrix. Whenever digis are found to be adjacent to one another, they are grouped together in order to represent the charge deposited by a single particle. The digi with the smallest index in a connected group is designated as the *seed*, serving as the representative for the cluster. Subsequently, a dedicated thread is assigned to each seed, which facilitates the organization of the clustering process across multiple computing threads. To ensure that each cluster receives a distinct identifier, a global atomic counter is incremented by all active threads. As a result, every seed acquires a unique index that can be consistently

tracked throughout the subsequent stages of the reconstruction chain.

2.2.1 Creation of n-tuplets

The next stage in the track reconstruction workflow is the formation of n-tuplets of clusters, which serve as the building blocks for the later fitting procedure used to extract the final track parameters. This process begins with the identification of doublets, defined as pairs of clusters whose hits originate from adjacent detector layers. In order for two clusters to be linked into a doublet, they must satisfy specific geometrical and physical constraints. For instance, the relative positions of the hits must be consistent with the expected trajectories of charged particles propagating outward from the interaction point, and the cluster sizes must fall within a range compatible with the passage of a single particle. Once a sufficient number of valid doublets have been constructed, the algorithm proceeds to the creation of triplets. This is accomplished by testing, in parallel, all doublets that share a common hit and attempting to extend them with an additional cluster from the next detector layer. A critical compatibility check at this stage involves ensuring that the three hits align within the R–Z plane (the plane defined by the radial distance from the beam axis and the longitudinal coordinate along the beam line). However, this is not enough, since multiple n-tuplets could correspond to the same particle. To resolve the ambiguity, among all the aligned doublets that share the outermost hit, only the shortest one is kept and among all the tracks that share a hit-doublet, only the ones with the largest number of hits are kept. The fits are then performed in parallel using one thread per n-tuplet. The fitting procedure accounts for realistic effects such as energy losses and includes the Broken Line Fit [12], which accounts for multiple scattering due to the interaction of the particles with the detector. Finally, the fitted tracks are used to make pixel vertices.

2.3 Divisive Cluterizer (Existing Algorithm)

Divisive Clusterizer is a 1-dimensional clustering algorithm used in vertex finding — specifically to identify primary vertices along the beam (z) axis from a set of tracks. It groups tracks based on their z positions (typically the track's distance along the beamline at the point of closest approach). The goal is to find clusters of tracks that come from the same collision vertex. The "divisive" part refers to the algorithm's strategy: it recursively divides clusters of tracks until each cluster corresponds to a well-defined vertex. The core idea is the following:

1. Tracks are sorted by z coordinate
2. Adjacent tracks that are close enough (within a `zSep` threshold) are grouped
3. Each group is tested to see if it contains enough tracks (`nTkMin`)

4. Within each group: outlier tracks far from the cluster's center (more than `zOffset`) are iteratively cleaned out. Cleaned clusters are merged to form stable vertex candidates. Tracks discarded from one cluster are examined to see if they can form a new one.

5. This continues until all tracks are assigned to clusters or discarded

2.3.1 Track Clustering Along the z-axis

After the tracks are sorted according to their z-coordinate, the algorithm proceeds by evaluating the longitudinal distance between consecutive tracks. Specifically, for each track i, the distance

$$\Delta z_i = z_i - z_{i-1} \tag{4}$$

to the previous track is computed. As long as this distance remains below a predefined threshold, `zSep`, the track is added to a temporary collection intended for clustering. This step effectively groups together tracks that are close in the longitudinal dimension, providing an initial approximation of potential vertex candidates.

When a track is encountered for which $\Delta z_i >$ `zSep`, the current collection is passed to the method `makeCluster1Ds`. This method serves a dual purpose: it first identifies and removes outliers, and then it refines the position of the resulting cluster. Outliers are defined as tracks whose z-coordinate lies further from the weighted mean position of the cluster than a configurable tolerance, `theZOffset`. The weighted mean z-position of cluster k is calculated as:

$$\bar{z}_k = \frac{\displaystyle\sum_{i \in I_k} \frac{z_i}{\sigma_i^2}}{\displaystyle\sum_{i \in I_k} \frac{1}{\sigma_i^2}} \tag{5}$$

where z_i is the measured z-coordinate of track i, σ_i is the corresponding uncertainty, and I_k denotes the set of tracks assigned to cluster k. The use of the inverse-variance weighting ensures that tracks with smaller measurement uncertainties contribute more strongly to the estimated cluster position, which improves the robustness of the vertex reconstruction.

For each track in the cluster, the algorithm computes a normalized distance from the cluster's mean position:

$$d_i = \begin{cases} \dfrac{|z_i - \bar{z}_k|}{\sigma_i}, & \text{if useError} = \text{true} \\ |z_i - \bar{z}_k|, & \text{otherwise} \end{cases} \tag{6}$$

Here, the choice of using the uncertainty σ_i allows the algorithm to account for the differing precision of each track measurement. Tracks for which $d_i >$ `theZOffSet` are considered outliers and temporarily removed from the cluster. These outliers are not discarded; instead, they are collected together and subsequently merged to form a new cluster. This merging process ensures that tracks initially separated by large z-distances are re-evaluated in the context of forming additional vertex candidates.

This iterative procedure continues until the number of tracks in the outlier collection falls below a minimum threshold, `nTkMin`. By doing so, the algorithm guarantees that only statistically significant clusters are kept, preventing spurious clusters formed by isolated tracks. The result is a refined set of one-dimensional clusters, each representing a localized grouping of tracks along the z-axis, which can then be further processed to reconstruct primary and secondary vertices with improved accuracy.

3 Overview of clustering techniques

3.1 Heterogeneous Computing

Modern high-energy physics (HEP) experiments need huge amounts of computing power. The Compact Muon Solenoid (CMS) experiment at the Large Hadron Collider (LHC) measures proton–proton collisions at rates of up to 40 million per second. This produces tens of petabytes of data every year [1]. The CMS Software (CMSSW) framework is used to reconstruct, simulate, and analyze these events, and it must do so under tight time and resource limits.

In the past, this work was done using only general-purpose Central Processing Units (CPUs). But as the amount of data has grown, and reconstruction algorithms have become more complex, CPUs alone are no longer enough. This has led to the use of heterogeneous computing — systems that combine CPUs with special accelerators like Graphics Processing Units (GPUs). In such systems, each processor type does the work it is best at: CPUs handle complex, step-by-step tasks, while GPUs can process large amounts of data in parallel.

In CMSSW, heterogeneous computing makes it possible to run heavy, parallel workloads — such as the clustering algorithm in this thesis — on the GPU, while the CPU manages the workflow, input/output operations, and tasks that are not suitable for the GPU. This chapter explains the differences between CPU and GPU architectures and how these differences affect parallel computing in large scientific applications.

3.1.1 CPU vs GPU Architectures

A typical CPU is made up of several powerful cores, usually 4-64 in modern servers, each containing an Arithmetic Logic Unit (ALU), a control unit, registers, and small but fast memory units called caches, all connected to a larger dynamic random access memory (DRAM). Caches are arranged in levels:

1. L1 cache: Very small (tens of KB) but extremely fast, located directly inside the core.

2. L2 cache: Larger (hundreds of KB), a bit slower, often private to each core.

3. L3 cache: Shared between all cores, several MB in size, slower than L1/L2 but still much faster than DRAM.

On the other hand, GPUs are throughput-optimized devices, designed to run tens of thousands of lightweight threads in parallel. They sacrifice the complex control logic of CPUs in favor of more arithmetic units, allowing huge numbers of operations to be executed at once. The GPU is divided into several Streaming Multiprocessors (SMs),

each containing arithmetic units, as well as registers and a small shared memory area. The SM schedules and executes groups of threads called warps, each executing the same instruction at the same time (SIMT - Single Instruction, Multiple Threads). Each core has an L1 cache, while all the SMs share an L2 cache. There is also an external DRAM that stores global memory, and a small amount of shared memory that allows threads in a block to communicate.

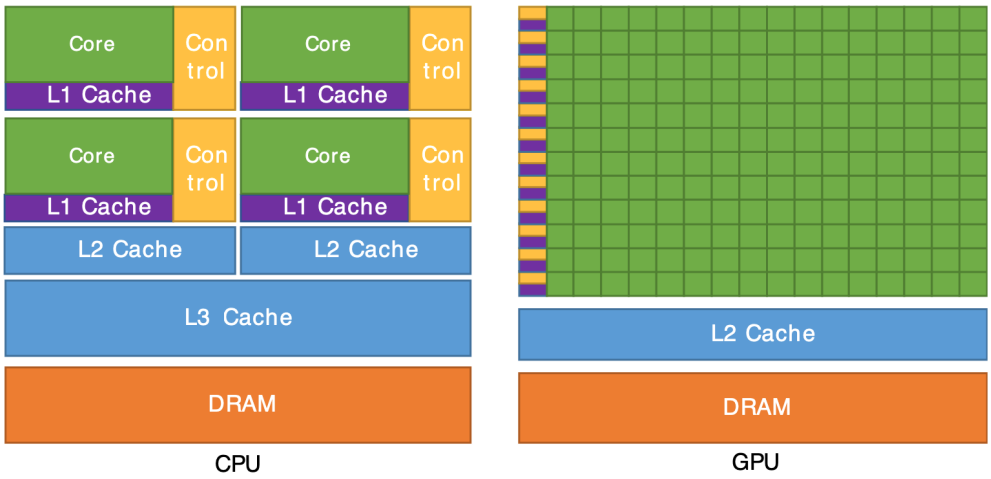

Figure 5: Architectures of a typical CPU and GPU

3.1.2 How CPUs and GPUs work together

It is clear now that CPUs and GPUs are designed for different purposes and can complement each other: while a CPU's aim is to execute a set of instructions, called a thread, as fast as possible, the GPU's goal is executing thousands of them in parallel, making up for the slower speed per thread. In CMSSW this balance is particularly useful: the CPU is responsible for reading and preparing the event data, after which the GPU takes over to perform the heavy, parallel tasks such as clustering or track fitting. Once the GPU finishes, the CPU processes the results and continues with the next steps of the reconstruction.

3.2 The alpaka Library

GPUs are now widely used in many fields of science and technology, including machine learning, image and signal processing, computational chemistry, and high-energy physics. In addition to GPUs, there are other types of accelerators such as FPGAs, TPUs, and custom ASICs. Each of these devices often comes with its own programming model and toolchain — for example, CUDA is specific to NVIDIA GPUs, while HIP/ROCm

is used for AMD devices. This variety creates a significant challenge: software written for one platform cannot easily run on another without major changes. Adapting a large codebase each time the hardware changes is time-consuming and error-prone. To address this, developers need programming models that allow them to write algorithms in a single, unified way, while still being able to run them efficiently on different kinds of accelerators. This is the aim of alpaka [13], a header-only C++20 library that provides performance portability across different back-ends using CUDA, HIP, SYCL, OpenMP 2.0+, std::thread, and also serial execution. The alpaka abstraction of parallelization is a multidimensional grid of threads, made up of four main hierarchies called *grid, block, thread, element* (Fig.6). The algorithms to be parallelized are called *kernels*. They consist of a common set of instructions executed by all threads in a grid, and they are defined as function objects with specific requirements, while the *accelerator* type and *work division* handle the parallelization and the mapping of threads and blocks.

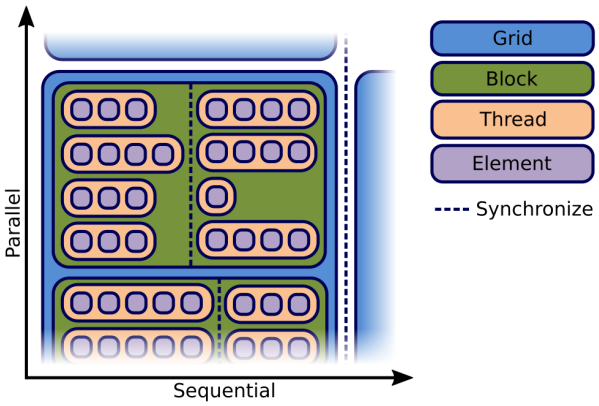

Figure 6: The alpaka parallelization hierarchy consists of a grid of blocks, and each block consists of threads, that in turn process multiple elements. [13]

Each parallelization level corresponds to a specific memory level:

- **Grid**: Set of blocks with a large global memory, accessible by all threads in all blocks. Grids are independent from one another, and can thus be executed sequentially or in parallel.
- **Block**: Set of threads with small shared memory, accessible all threads in the same block. Blocks on a grid are independent of each other and can thus be executed sequentially or in parallel.
- **Thread**: Threads have access to their private registers, the shared memory of the block and the global memory. Threads in a block are independent from each other and can thus be executed sequentially or in parallel.

- **Element**: This level is necessary since the optimal workload on threads depends on the backend. For example on CPUs, each thread often benefits from SIMD (Single Instruction, Multiple Data) execution, while on GPUs it is more beneficial to scale the number of threads employed instead. Furthermore, on some architectures processing multiple elements per thread may improve caching.

3.2.1 Integration within CMSSW

The alpaka module integration in CMSSW is a separate layer on top of the core framework, and certain modules get compiled once for each enabled alpaka backend.
In order to do that without having to reuse any code, they are defined in the `namespace ALPAKA_ACCELERATOR_NAMESPACE`, a macro substituted with a concrete, backend-specific namespace name to guarantee different symbol names for all backends, which can take the following values:

- CPU serial: `alpaka_serial_sync`
- CUDA (NVIDIA): `alpaka_cuda_async`
- ROCm (AMD): `alpaka_rocm_async`

The namespace also defines several classes, to implement fully asynchronous producers (`global::EDProducer<...>`), to give access to host-side and device-side data products (`device::Event`, `device::EventSetup`), and to consume/produce device-side data products (`device::EDGetToken<T> / device::EDPutToken<T>`).

4 Pixel-vertexing in CMSSW with CLUEstering

4.1 Vertexing Performance: Definitions

To assess the performance of a vertexing algorithm (validation), several quantitative parameters are calculated. These parameters are used to compare the reconstructed vertices produced by the algorithm with the corresponding simulated, or "true" vertices. By performing this comparison, it becomes possible to evaluate the accuracy and efficiency of the algorithm in identifying the correct number and position of vertices, as well as its ability to distinguish closely spaced interaction points. This analysis provides a clear measure of how well the implementation performs relative to the expected physical truth and helps identify potential areas for optimization or improvement. New criteria have been developed to achieve a more robust validation of pixel vertices. In the previous validation approach, a reconstructed vertex was considered **matched** to a simulated one if it satisfied the following conditions:

- $|\Delta z| < 1$ mm
- $|\Delta z|/\sigma_z < 3$

This validation is purely geometrical, which means that a reconstructed vertex could be matched to a simulated vertex if they are close in z, but their content could still be very different. In this case the association would match two very different vertices. For this reason the validation has been improved to add additional criteria on the content of the vertices. In particular, to have a match between a reco and a sim we can impose a cut on one of the three following quantities:

- Plain fraction of **shared tracks**, between the simulated and reconstructed vertex
- Fraction of shared tracks weighted by their transverse momentum squared, p_T^2
- Fraction of shared tracks weighted by $1/\sigma_z^2$, i.e. the resolution of the z of each track

Ideally each reconstructed vertex should be associated to one, and only one, simulated vertex. In reality this is never the case, in particular wrongly associated vertices as:

- **fake**: a reconstructed vertices that isn't associated to any simulated vertex
- **merged, or multi-matched**: a reconstructed vertex that is associated to more than one simulated vertex
- **duplicate**: a reconstructed vertices that is associated to a simulated vertex, which in turn is associated to more than one reconstructed vertex

With these definitions we can now define the metrics used to validate the quality of a vertex reconstruction algorithm:

- **Efficiency**: Fraction of simulated vertices associated with at least one reconstructed offline vertex.
- **Fake rate**: Fraction of reconstructed vertices that do not match to any simulated vertex.
- **Split rate**: Fraction of reconstructed vertices identified as duplicates.
- **Merge rate**: Fraction of reconstructed vertices which are associated to more than one simulated vertex, i.e. multi-matched.

4.2 CLUstering of Energy

CLUstering of Energy (CLUE) is a density-based clustering algorithm that offers easy parallelization and linear scalability [14]. In CLUE, each sensor pixel is treated as a two-dimensional point, weighted on its energy.

Like other density-based algorithms, CLUE constructs clusters by identifying the regions of the data space where the density of points is higher. CLUE divides points into three classes: *seeds*, which are the centers of the different clusters and this the points around which the density is highest, *followers*, which are the points linked iteratively to a seed and that make up the clusters and finally the *outliers*, which are the points that are not linked to any cluster and are thus discarded as noise. The thing that separates CLUE from other density-based algorithms is that it's also *weighted*, meaning that to each point can be associated a weight, and this weight gives a measure of the respective importance of the points, information that is used when computing the local density of the points.

4.2.1 Spatial Indexing

As shown in Fig. 7, data points can be accessed with high efficiency by partitioning the space into fixed rectangular bins and employing a spatial index to query local neighborhoods. This strategy avoids looping over the entire dataset for each point, which would be very demanding computationally. For each layer of the detector, a fixed-grid index is created by assigning the indexes of the two-dimensional points to the appropriate square bins in the grid. Formally, for each point i, two sets are defined: $\Omega_d(i)$ is the collection of points that lie in the bins intersecting the square window $[x_i \pm d, y_i \pm d]$:

$$\Omega_d(i) = \{j : j \in \text{tiles touched by the square window } [x_i \pm d, y_i \pm d]\} \tag{7}$$

while the d-neighborhood of i, $N_d(i)$, is the set of points within a distance d from i:

$$N_d(i) = \{j : d_{ij} < d, j \in \Omega_d(i)\} \tag{8}$$

where it's clear that $N_d(i) \subseteq \Omega_d(i)$. Thus, to query $N_d(i)$, the algorithm only needs to loop over the points in $\Omega_d(i)$, and since that d is small, and the maximum granularity of the points is constant, the complexity of this operation is $O(1)$.

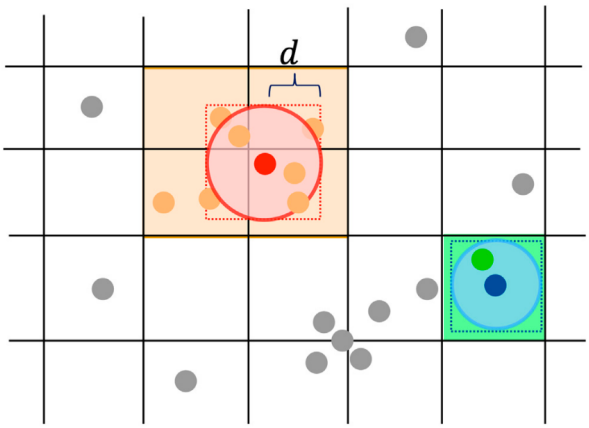

Figure 7: Illustration of the fixed grid highlighting N_d and the search window [14]

4.2.2 Clustering

The clustering procedure, shown in Fig. 8, requires the following tunable parameters: d_c, the cut-off distance in the computation of the local density; ρ_c, the minimum density to promote a point to a seed and the maximum density to demote a point to an outlier; and lastly δ_c and δ_o are the minimum separation required for seeds and outliers. Firstly, ρ and δ are calculated for each point, the ones with density $\rho > \rho_c$ and separation $\delta > \delta_c$ are promoted as seeds. On the other hand, the ones with $\rho < \rho_c$ and $\delta > \delta_c$ are demoted as outliers. The points that are neither seeds nor outliers are registered to the follower list of their nearest-higher, and finally, cluster indexes are passed down from seeds through their follower list.

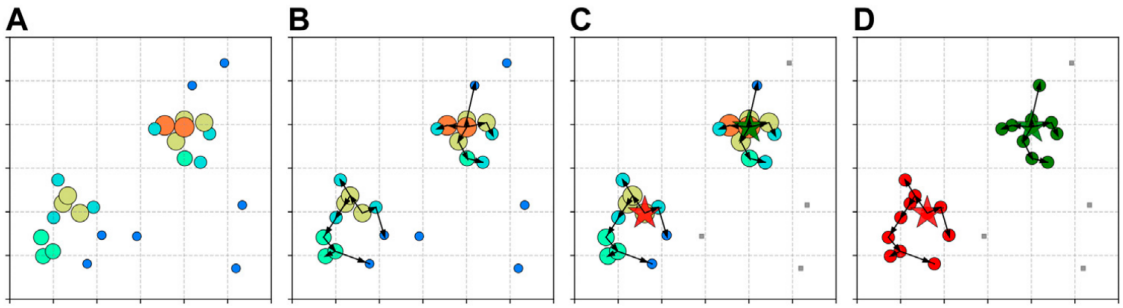

Figure 8: Clustering procedure. **(A)**Local densities are computed and represented by the color and size of each point. **(B)**Nearest-highers are computed, the arrows point from the nearest-higher of a point, to the point itself. **(C)** Points are promoted as seeds or demoted as outliers. **(D)**Cluster indexes are passed down from seeds to followers [14]

.

4.2.3 Parallelization

The parallelization of CLUE on GPU is achieved by assigning a thread to each point, in order to construct spatial indexes, compute ρ and δ, promote points to seeds or demote them to outliers, and register the rest to the lists of followers of their nearest-highers. Block size for each kernel does not affect performance significantly [14], thus it is set to 1024. Afterwards, a thread is assigned to each seed to pass down the cluster index. Since the results of each step are required for the following, synchronization between threads at the end of each step is necessary and is obtained by implementing each step as a separate kernel. GPU memory access is optimized with coalescing, by storing all the points in a single structure of array (SoA). Thread conflicts to access the same data in the GPU global memory may occur in the following cases:

1. While building the fixed grid spatial index, when multiple points need to register to the same bin

2. While promoting seeds, when multiple points need to register to the list of seeds

3. While expanding follower lists, when multiple points need to register as followers of the same seed

Thus, atomic operations are needed in order to avoid these conflicts. This leads to some microscopic serialization, since during atomic operations, a thread is given exclusive access to read or write to memory. However, the serialization in cases (1) and (3) is negligible, as the number of bins and followers of a point is small. On the other hand, the serialization in case (2) can be costly, given that the number of seeds k is large. Data transfer between the CPU (host) and the GPU (device) is both computationally expensive and time-consuming. Therefore, minimizing these memory copy operations is

crucial for achieving optimal performance. The overall workflow of the CLUE algorithm is illustrated in Fig. 9.

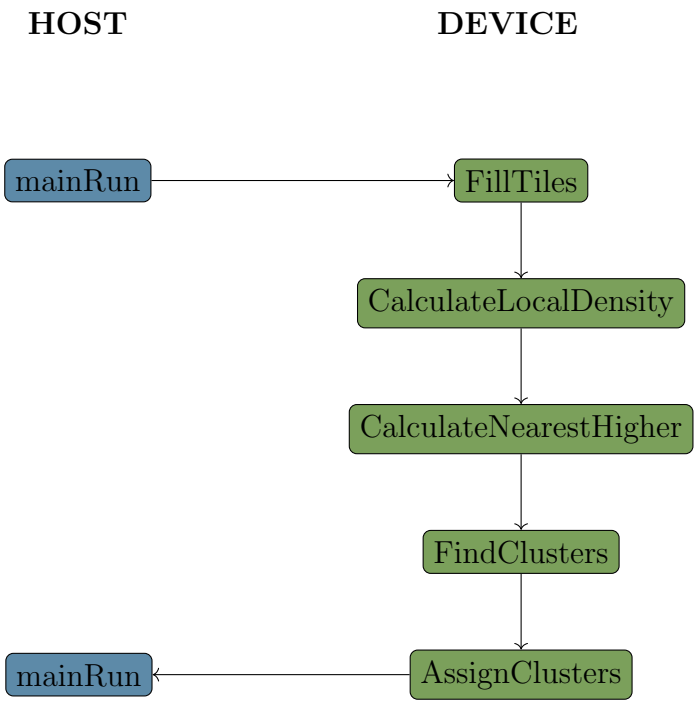

Figure 9: Illustration of the workflow of the CLUE algorithm. Notice how the entire algorithm is executed on the device, so that the only memory copies with the host are at the beginning and at the end of the execution, which results in much better performance [16].

4.2.4 The CLUEstering library

The CLUEstering library [16][17] has been developed as a versatile and extensible software package designed for use across a wide range of applications, both within and beyond High Energy Physics. While it was initially conceived for vertex reconstruction and clustering tasks in particle physics experiments, its generic design enables its application in diverse domains — such as the identification of forested regions through tree clustering in satellite imagery, the detection of stellar clusters in astronomical CCD images, and the implementation of nearest-neighbor searches in latent spaces for machine learning models. The computational backend of the library is implemented in C++, a language well known for its high execution speed and close-to-hardware control. This choice ensures optimal performance and facilitates compatibility with emerging performance-portable frameworks, many of which are themselves based on C++. However, limiting the library to a C++ interface would considerably restrict its accessibility, as modern data analysis

and machine learning workflows are predominantly implemented in Python. Python has become the de facto standard language in scientific computing due to its ease of use, extensive ecosystem, and integration with major frameworks such as PyTorch, Keras, and Scikit-learn. To bridge this gap, a Python interface was developed for the CLUEstering library. This interface provides users with a high-level, intuitive means of interacting with the underlying C++ core, while preserving computational efficiency. It also simplifies the installation process and broadens the potential user base, making the library more appealing and accessible to researchers and practitioners across the scientific community.

4.3 Pixel-Vertexing in CMSSW

4.3.1 Structure of Array (SoA) Generation

The code written for this thesis makes extensive use of the SoA classes [15] mentioned in section 2.1. These classes are generated by preprocessor macros, and in turn, they generate multiple, aligned columns from a memory buffer, which is allocated separately by the user and can be located in a space different from the local one. These are the preferred structures to hold data on the GPU, since the contiguous storage allows coalesced access on cache line aligned data, as shown in Fig. 10.

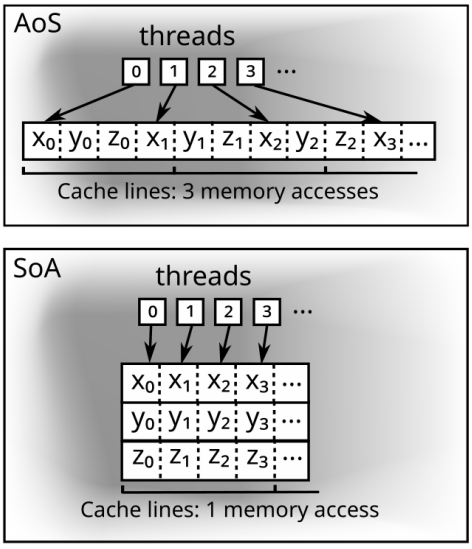

Figure 10: AoS and SoA access patterns [15]

A SoA supports three types of elements: numeric columns, targeted to store numeric classes; scalars, which hold only a single element per SoA; and Eigen columns that hold vectors and matrices.`Layouts` divide a memory buffer into columns whose size is computed at runtime, whereas `Views` provide the interface to the data via a pointer for

each column, making them light structures that can easily be passed to kernels. The `View` can be generated as constant or non-constant, with the same interface where scalar elements are accessed with an `operator()`: `soa.scalar()` while columns and Eigens are accessed via an array of structure (AoS)-like syntax: `soa[index].x()`.

4.3.2 Implementation

This section describes how CLUEstering was integrated into the CMSSW workflow for pixel vertex reconstruction. The process begins by retrieving tracks from the `device::Event` by consuming a `device::EDGetToken<T>`, which provides access to Event data products of type `T`. In this context, `T` refers to a SoA class that contains several columns, such as the track χ^2, the transverse momentum p_t, and a scalar representing the number of tracks (`nTracks`). The data is then loaded onto a `WorkSpaceSoA` in the device that serves as an intermediate container to execute the CLUEstering algorithm with. This process is easily parallelized since the SoA layouts are very similar and is thus executed with a kernel that performs cuts by only choosing high-purity tracks, with a p_t higher than a configurable `ptMin` (1.0 GeV), and clamping the rest to a configurable `ptMax` (75.0 GeV). Next, CLUEstering is executed in one dimension where the z-coordinate of each track is used as the clustering variable, the squared transverse momentum p_T^2 serves as the weight, and the parameters of the algorithm are set as follows: $d_c = d_m = 0.04cm$, $\rho_c = 0.01GeV$. In the post-processing of the vertices we also required each vertex to contain at least 2 tracks. The algorithm produces two output buffers: the first assigns a cluster index to each track, effectively labeling which cluster it belongs to, while the second provides a binary flag indicating whether a track is identified as a seed (1) or not (0). Finally, the tracks are fitted to refine the position, weight, and χ^2 of each vertex. An iterative fitting and refinement procedure is applied in order to obtain precise vertex positions and to improve the quality of the reconstruction. The procedure consists of four successive steps, executed through dedicated GPU kernels:

1. **Initial Vertex Fit**: Each proto-vertex obtained from clustering is fitted using a weighted approach. The position along the beam axis (z) is determined as the weighted average of the associated track positions, with weights given by the inverse of the track uncertainties. For every track, a χ^2 contribution with respect to the fitted vertex is evaluated. Tracks that exceed a configurable χ^2 threshold are removed as outliers, and the vertex weight is scaled according to the number of retained tracks and the overall fit quality.

2. **Vertex Splitting**: Vertices exhibiting a poor fit quality are further analyzed. The associated tracks are reclustered into two groups along the z coordinate through an iterative, k-means–like algorithm. If the two groups are significantly separated compared to their uncertainties, a new vertex is created and the corresponding

tracks are reassigned. This step is essential to disentangle cases where two nearby interaction vertices were initially merged into a single cluster.

3. **Refined Vertex Fit**: After splitting, the fitting procedure is repeated on the updated vertex collection. This ensures that the positions, uncertainties, and χ^2 values are consistently recalculated with the new track-to-vertex assignments. Outlier rejection is re-applied, which further stabilizes the reconstruction.

4. **Vertex Ordering**: Finally, the vertices are ranked according to their physical significance. For each vertex, the sum of squared transverse momenta (p_T^2) of its tracks is computed. The vertices are then sorted by this quantity, which prioritizes the most energetic interactions and facilitates their selection in subsequent analyses. The first vertex of the sorted list is taken to be the **primary vertex**, which is used as an indicator of the performance of the algorithm. Comparing the primary reconstructed vertex with the primary simulated one reveals whether the algorithm merged distinct vertices, when the former has a significantly higher p_T^2, or conversely, split a single vertex.

4.4 Results

To evaluate the performance of the CLUEstering clustering algorithm relative to the legacy vertexing method, a sample of 1000 simulated $t\bar{t}$ events with an average pileup of 200 interactions was processed. The reconstructed vertices obtained from both algorithms were then compared to the corresponding simulated vertices. An image of the amount and location of the simulated vertices can be drawn from the plots in Fig. 11 and Fig. 12, respectively. As shown in Fig.13 and Fig. 14, CLUEstering demonstrates a higher vertex efficiency, successfully matching a larger fraction of simulated vertices. This improved efficiency is accompanied by a modest increase in the vertex fake rate (Fig. 16), indicating a higher number of reconstructed vertices not associated with any simulated counterpart. However, the number of vertices reconstructed with CLUE is lower than the number of vertices reconstructed with the legacy algorithm, as shown in Fig. 17 and Fig. 18. Thus, the higher fake rate may partly result from a smaller denominator in the definition (Fake Rate = #Fakes/#All Reconstructed Vertices). Nevertheless, CLUE exhibits reduced merge (Fig. 15), implying fewer instances of multiple simulated vertices being reconstructed as one. The standard deviation in the z coordinate of the two procedures is comparable, with the one associated with CLUE being slightly higher (Fig. 21). Overall, these results indicate that CLUEstering provides a net improvement over the legacy algorithm, offering enhanced reconstruction efficiency with better vertex separation performance under high-pileup conditions.

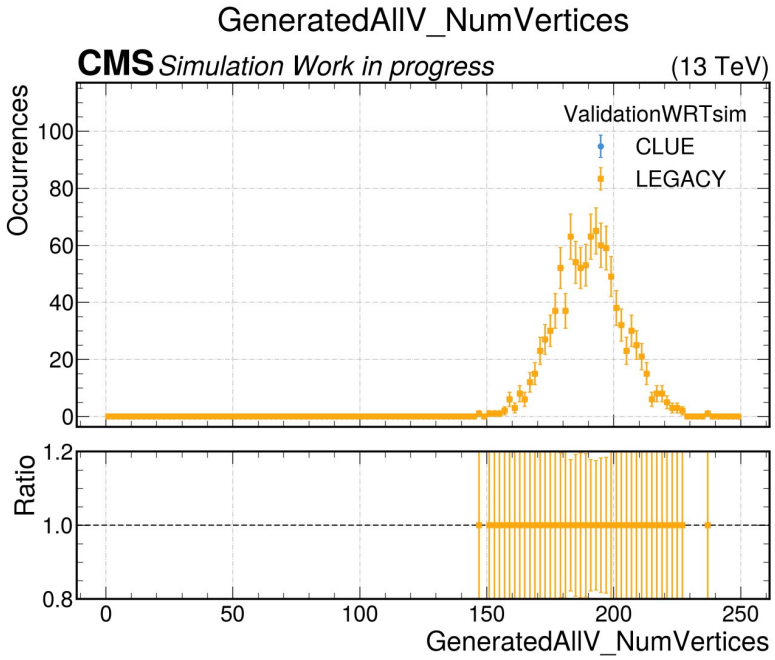

Figure 11: Distribution of the number of simulated vertices per event.

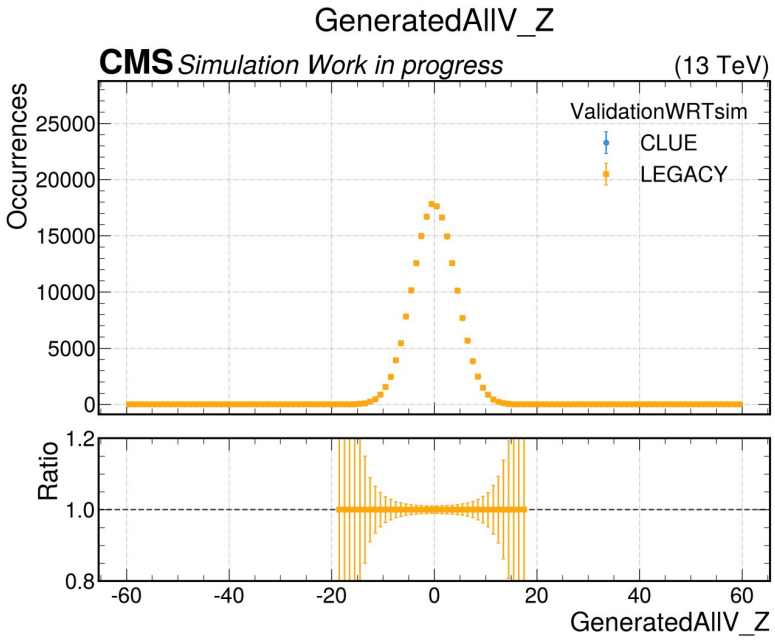

Figure 12: Distribution of the position of simulated vertices per event (in cm).

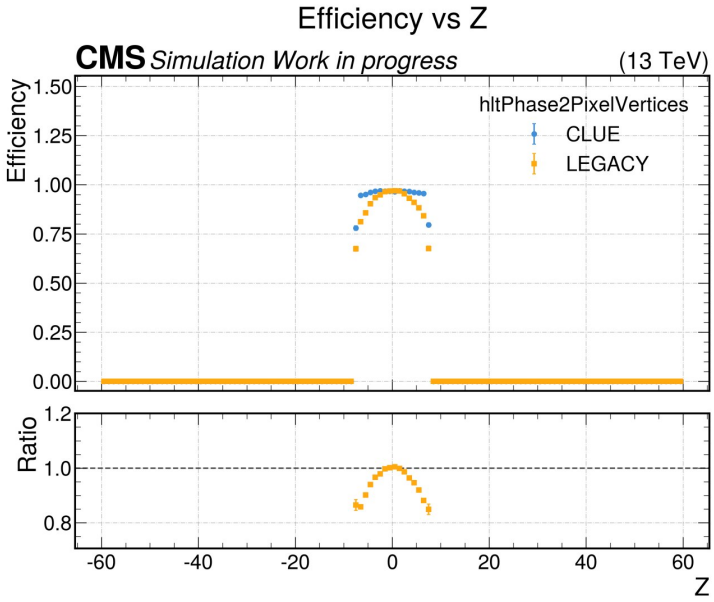

Figure 13: Efficiency as a function of the z coordinate (in cm). The efficiencies of the two procedures are comparable, but overall the vertices reconstructed with CLUE perform better.

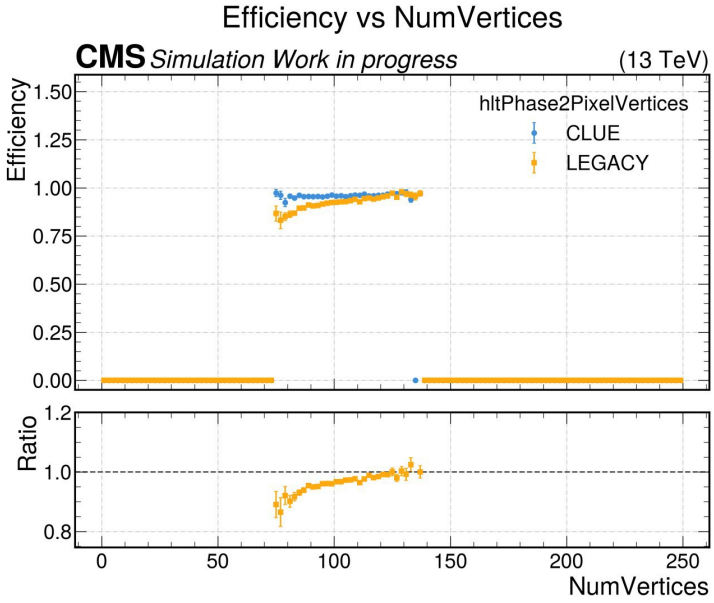

Figure 14: Efficiency as a function of the number of vertices. The efficiencies of the two procedures are comparable, but overall the vertices reconstructed with CLUE perform better.

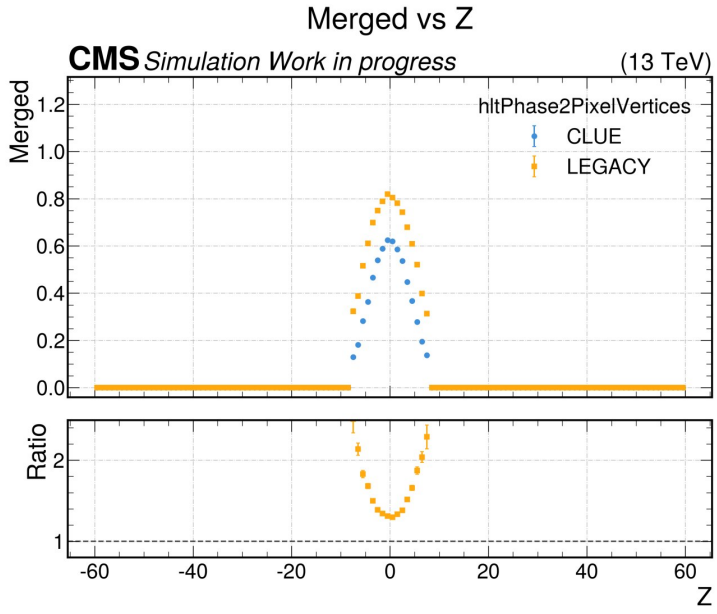

Figure 15: Merge rate as a function of the z coordinate (in cm). CLUE exhibits reduced merge, implying fewer instances of multiple simulated vertices being reconstructed as on

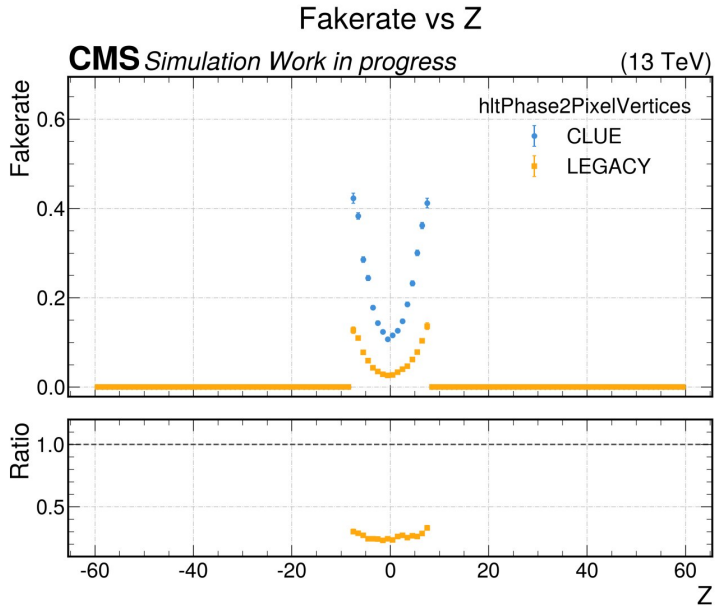

Figure 16: Fakerate as a function of the z coordinate (in cm). Reconstructed vertices with CLUE exhibit a higher fake rate, possibly resulting from a smaller denominator in the definition (Fake Rate = #Fakes/#All Reconstructed Vertices).

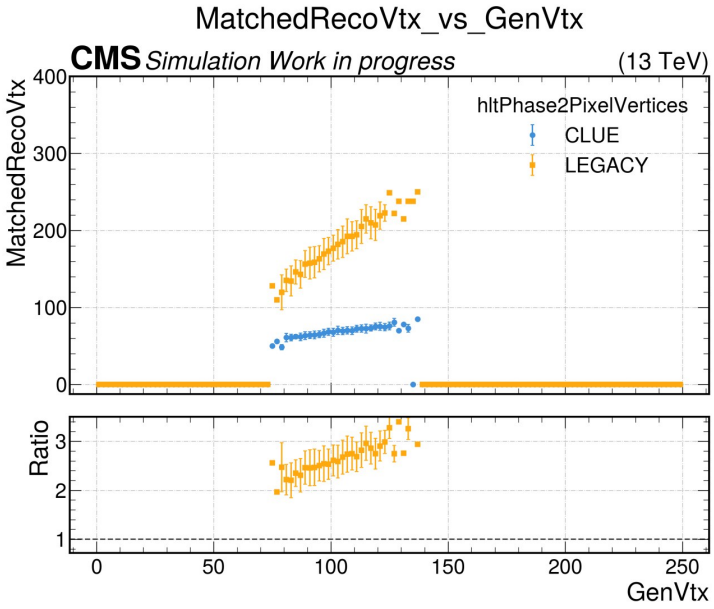

Figure 17: Number of matched vertices as a function of number of reconstructible simulated ones, the number of matched vertices reconstructed with CLUE is lower than the number of matched vertices reconstructed with the legacy algorithm

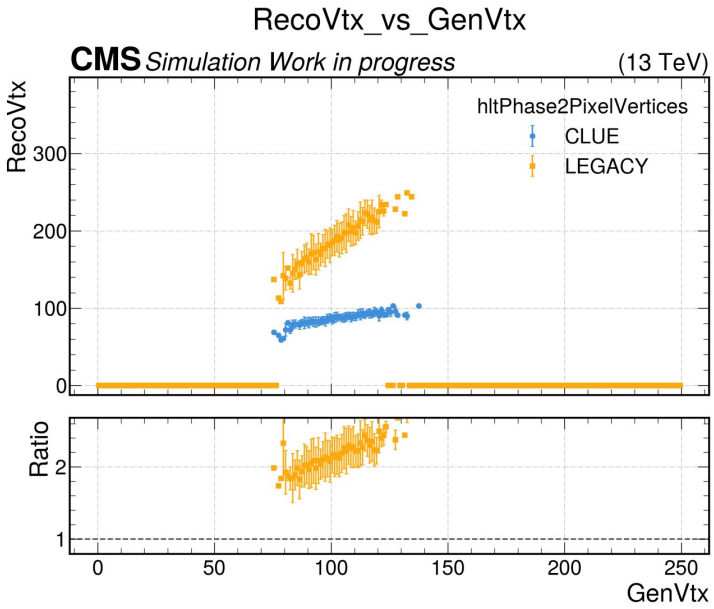

Figure 18: Number of vertices as a function of number of reconstructible simulated ones, the number of vertices reconstructed with CLUE is lower than the number of vertices reconstructed with the legacy algorithm

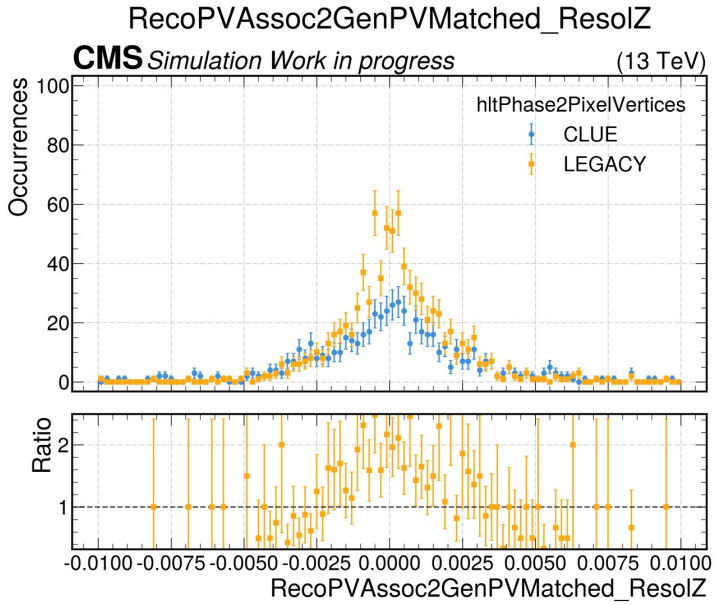

Figure 19: Number of matched primary vertices as a function of the resolution in the z coordinate (in cm). The values between the two procedures are comparable, but CLUE stands slightly lower.

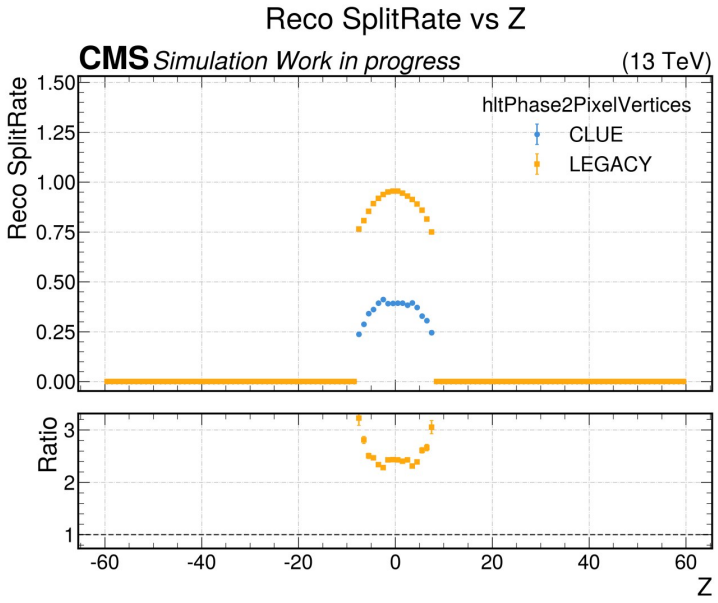

Figure 20: Split Rate as a function of the z coordinate (in cm). The reconstruction with CLUE exhibits a better performance by having a lower fraction of duplicates.

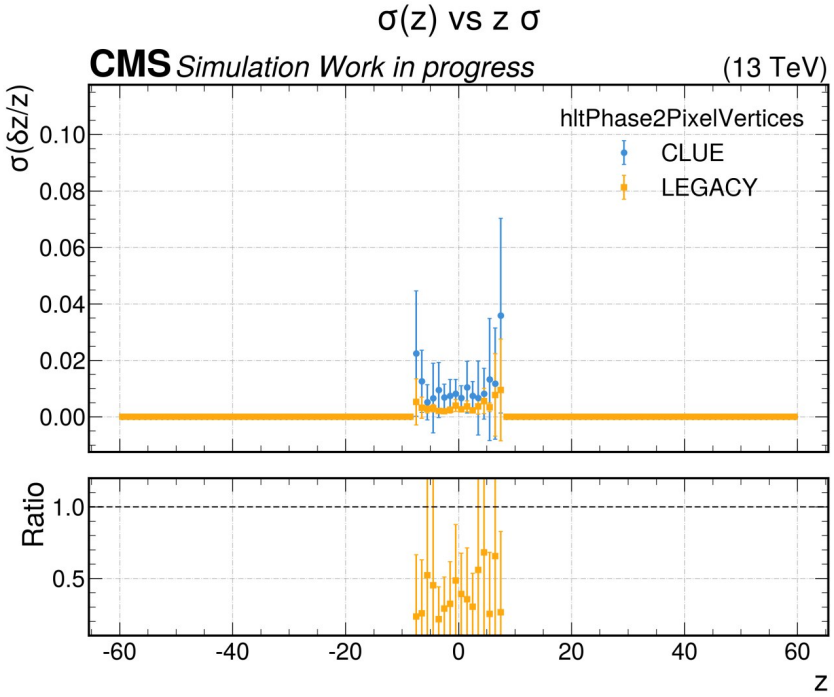

Figure 21: Standard deviation of the error in the z position of the reconstructed primary vertices as a function of the z coordinate (in cm). The standard deviation in the z coordinate of the two procedures is comparable, with the one associated with CLUE being slightly higher.

Conclusions and Future Work

This thesis presented the implementation and validation of a vertex reconstruction workflow using the CLUEstering library to cluster pixel-tracks into pixel-vertices for the High-Level Trigger (HLT) in Phase 2 of CMS.

The reconstruction was validated on a sample of $t\bar{t}$ events with an average of 200 overlapping proton-proton collisions, so to test it against the realistic conditions for the Phase 2 of the CMS experiment starting from 2030.

This new vertex reconstruction showed promising results with an efficiency comparable to or higher than the legacy algorithm, as well as lower split and merge rates. The higher fake-rate and inefficient primary vertex reconstruction will have to be investigated, and indicate that a parameters tuning is needed to improve its physics performance.

Furthermore, since the CLUEstering library used in the workflow is implemented using alpaka, the performance portability library already widely used in the CMS software, the workflow is ready to be included in the alpaka reconstruction, which has the advantage of being able to leverage both NVIDIA and AMD GPUs, which provide much better scalability with the problem's size. The legacy algorithm used currently in production, on the other hand, is implemented to only run serially on CPUs, which will be too inefficient as the amount of data produced in each event will increase. The increase in performance provided by the heterogeneous reconstruction with alpaka is a necessity for preparing the High-Level Trigger for HL-LHC.

References

[1] CMS Collaboration, *The CMS Experiment at the CERN LHC*, JINST 3 S08004, 2008

[2] Aad, G.; Anduaga, Xabier Sebastian; Antonelli, S.; Bendel, M.; Breiler, B.; et al.; *The ATLAS Experiment at the CERN Large Hadron Collider*; Journal of Instrumentation; 2008

[3] The ALICE Collaboration, *The ALICE experiment at the CERN LHC*, Journal of Instrumentation, 2008

[4] The LHCb Collaboration, *The LHCb Detector at the LHC*, Journal of Instrumentation, 2008

[5] E. Lopienska, *The CERN accelerator complex, layout in 2022*, 2022

[6] W. Herr, B. Muratori, *Concept of luminosity*, CERN Accelerator School: Intermediate Accelerator Physics, 2006

[7] Morovic, *CMS detector: Run 3 status and plans for Phase-2*, arXiv:2309.02256 [hep-ex], 2023

[8] CMS Collaboration, *The Phase-2 Upgrade of the CMS Level-1 Trigger*, Tech. Rep., 2020

[9] A. Bocci, M. Kortelainen, V. Innocente, F. Pantaleo, M. Rovere, *Heterogeneous reconstruction of tracks and primary vertices with the CMS pixel tracker*, Frontiers in Big Data, 2020

[10] C.D. Jones, M. Paterno, J. Kowalkowski, L. Sexton-Kennedy and W. Tanenbaum, *The new CMS event data model and framework*, *Proceedings of International Conference on Computing in High Energy and Nuclear Physics (CHEP06)*, 2006

[11] The Tracker Group of the CMS Collaboration, *The CMS Phase-1 Pixel Detector Upgrade*, arXiv:2012.14304 [physics.ins-det], 2020

[12] V. Blobel, *A new fast track-fit algorithm based on broken lines*, Nucl. Instrum. Meth., 2006

[13] E. Zenker et al., *alpaka - An Abstraction Library for Parallel Kernel Acceleration*, IEEE Computer Society, 2016

[14] Rovere M., Chen Z., Di Pilato A., Pantaleo F., Seez C., *CLUE: A Fast Parallel Clustering Algorithm for High Granularity Calorimeters in High-Energy Physics*, Frontiers in Big Data, 2020

[15] Cano E., *Implementation of generic SoA data structure in the software*, CMS-CR-2023-040, 2023

[16] Balducci S., *CLUEstering: a high-performance density-based clustering library for scientific computing*, `https://amslaurea.unibo.it/id/eprint/32544/`, 2024

[17] Balducci S., Pantaleo F., Perego A., Redjeb W., Rovere M., *CLUE: A Scalable Clustering Algorithm for the Data Challenges of Tomorrow*, Newsletter of the EP Department, 2025

Acknowledgements

I would like to thank Professor Giacomini for this opportunity, working on such a big project has been a unique experience. I am also grateful to Simone and Felice, for guiding me on this journey and being present every step of the way. They gave me the chance to take a glimpse into what it's like to work in a large research facility. I am thankful to the Patatrack team at CERN, for helping out and being supportive through every minor and major obstacle encountered along the way. Furthermore, none of this would have been possible without the support and prayers of my Parents and my Brother, and their unconditional love. This is just the first step in making them proud, and paying them back for all the sacrifices they made. Finally, my thoughts go to all my friends, whether new or old, whether close or far away, thank you for everything.