



Development of GPU-Accelerated Trigger Algorithms for the ATLAS Experiment at the LHC

Nuno dos Santos Fernandes

Thesis to obtain the Master of Science Degree in

Engineering Physics

Supervisor(s): Prof. Patricia Conde Muíño

November 2021

Written for, and dedicated to, every person who ever believed or trusted in me.

May Ariadne's red string always lead us to the safety of knowledge.

Epis sek tan, tan sek epis.

Acknowledgments

This work was supported by LIP - Laboratório de Investigação e Física Experimental de Partículas. The author must also acknowledge the computational resources provided by INCD - Infraestrutura Nacional de Computação Distribuída.

The work would also not be possible without all the support, pertinent questions and helpful suggestions from the author's supervisor Professor Patricia Conde Muño, from Cosmin-Gabriel Samoila and from the elements of the ATLAS Experiment group at LIP.

The author must acknowledge the efforts of Ademar Delgado, who developed the prototype implementation of the algorithm in a GPGPU context and a framework for validating its results which laid the foundations for the present work, and the contributions of Miguel Alves, in particular to the porting of that prototype version to the current Athena framework.

The author cannot thank Gonçalo (the T_EXan master!), Óscar, Pedro and Francisco enough for all the shared experiences, kind help and sage advice during the long and often arduous journey that is learning Physics, where, when and how them and the author did, from the simplest of derivatives to the most unsolvable of differential equations, from the most classical of mechanical problems to the most abstract quantum space, from the most basic bitwise operator to the most complex of computational tasks. And, especially, from the most rebellious of Stirling Engines to the less visible of Millikan oil drops.

It would be unfair not to mention, even if not in full detail, all the people who helped the author arrive at who and what he is (or is not) today, especially those that have appeared at both sides of the teaching process. It would be unfairer still not to mention specifically Professors Lígia and Maria José, to whom the author hopes the lack of contact in recent times might be partially repaid through this special mention.

The author is bound by the inescapable obligation of stating his infinite debt to the Eternal Muse of the author's aphaeretic soul for all she has granted the author, despite everything else that might have been taken along the way. If, by any twist of fate, her eyes ever skim this page, let this be yet another way of stating the immutable truth that she undoubtedly knows already. *Laem infite.*

Most of all, and above all else, the author must acknowledge the patience, and support, and understanding, and help, and dedication, and forgiveness, and attention, and - in short - the love of all those to whom the Universe granted the (mis)fortune of being related to the author, those who had to bear the many long nights, and strange hours, and overall out-of-sync existence that the author's idiosyncrasies, together with the fickleness of the quasi-artistical inspirations that drive the author's writing, made necessary for the completion of the present work. Though nothing can ever repay them for all that they have done, and will continue to do, for the author, it is the author's deepest hope that whatever small contributions the author may make, though this or whatever else the author may do, to the betterment of Science, and, with it, of humanity may allow all of their efforts to not have been (entirely) in vain.

And, as a final note of inevitable meta-textuality, the author cannot help but acknowledge the reader, for paying attention to this oft overlooked part of texts such as this is.

Resumo

A experiência ATLAS no LHC, no CERN, tem como propósito detectar diversos processos físicos que ocorrem quando as partículas colidem. Devido à elevada taxa de colisões, que aumentará mais ainda com o LHC de Alta Luminosidade (*High-Luminosity LHC Upgrade*), torna-se necessário empregar um sistema de *triggering* para seleccionar os acontecimentos que serão armazenados para análise posterior. O facto de este sistema funcionar em tempo real constrange significativamente o tempo de execução dos algoritmos que o compõem, pelo que importa aferir a exequibilidade de acelerar a execução destes últimos. Uma possibilidade é o paralelismo massivo providenciado pelas unidades de processamento gráfico (GPU), nomeadamente através do *framework* CUDA.

Entre outros algoritmos, o sistema de *triggering* executa a reconstrução dos chuviros electromagnéticos que ocorrem nos calorímetros do detector ATLAS, que é feita com base na energia depositada nas células que os compõem. Este trabalho foca-se no algoritmo conhecido por *Topological Clustering*, que agrupa as células de acordo com o rácio sinal-ruído da energia depositada para reconstruir os chuviros, e em particular numa sua variante mais adequada para a programação em GPU chamada *Topo-Automaton Clustering*.

São comparadas várias estratégias de implementação deste algoritmo, tendo em vista a optimização do seu tempo de execução, sendo investigadas as propriedades físicas reconstruídas dos grupos de células de modo a validar a implementação final.

Os resultados apontam para uma melhoria média do tempo de execução por um factor entre 3.5 e 5.5 (dependendo do tipo de evento), embora menos de 20% desse tempo corresponda ao algoritmo propriamente dito.

Palavras-chave: Física de Partículas; ATLAS; Programação em GPU; CUDA; Topological Clustering; Topo-Automaton Clustering.

Abstract

The ATLAS Experiment at the LHC, at CERN, is designed to detect several physical processes that happen when particles collide. Due to the high collision rate, which will increase even further with the High-Luminosity LHC Upgrade, a triggering system must be employed to select the events that will be stored for later analysis. The fact that the trigger must work in real-time significantly constrains the run time of the algorithms that comprise it, which means that the feasibility of accelerating their execution should be considered. One possibility is the massive parallelism provided by Graphical Processing Units (GPU), namely through the CUDA framework.

Among other algorithms, the triggering system performs the reconstruction of the electromagnetic showers that form within the ATLAS detector's calorimeters, which is based on the energy that is deposited in each of these calorimeters' cells. This work will focus on the algorithm known as Topological Clustering, which groups the cells according to the signal-to-noise ratio of the deposited energy to reconstruct the showers, and, in particular, on a variant which is better suited to GPU programming, the Topo-Automaton Clustering algorithm.

Several implementation strategies for the algorithm are compared in order to optimize its run time, and the reconstructed physical properties of the cell clusters are analysed to validate the final implementation.

The results suggest an average improvement of the run time by a factor between 3.5 and 5.5 (depending on the kind of the event), though less than 20% of that time corresponds to the algorithm itself.

Keywords: Particle Physics; ATLAS; General-Purpose GPU Programming; CUDA; Topological Clustering; Topo-Automaton Clustering.

Table of Contents

Acknowledgments	iii
Resumo	v
Abstract	vii
Table of Contents	ix
List of Figures	xiii
List of Tables	xv
List of Abbreviations	xvii
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.3 Thesis Outline	2
2 The LHC and the ATLAS Experiment	3
2.1 CERN and the Accelerator Complex	3
2.2 The LHC	4
2.2.1 Luminosity	6
2.2.2 The High-Luminosity LHC	7
2.3 The ATLAS Experiment	8
2.3.1 Coordinate Systems in the ATLAS Detector	8
2.3.2 Structure and Geometry of the ATLAS Detector	10

2.3.2.1	Inner Detector	10
2.3.2.2	Calorimeters	11
2.3.2.3	Muon Spectrometer	13
2.3.3	The ATLAS Trigger and Trigger Upgrades	13
2.3.4	Athena	15
2.3.4.1	Data Processing Within Athena	16
2.3.4.2	Data Storage and Retrieval	17
2.3.4.3	Calorimeter Data	17
2.3.4.4	Calorimeter Cluster Information	17
2.4	Topological Clustering	18
3	Graphical Processing Units as Accelerators	21
3.1	GPU Programming Languages	22
3.2	CUDA Execution Model and Architecture	23
3.3	Topo-Automaton Clustering	25
3.4	The First GPU-Accelerated Trigger Prototype	27
4	Implementation and Optimization of the Topo-Automaton Clustering	29
4.1	Structure of the Implementation	29
4.2	Algorithm Implementation Strategies	30
4.2.1	Tag Attribution Strategies	30
4.2.1.1	First Tagging Strategy	30
4.2.1.2	Second Tagging Strategy	32
4.2.1.3	Special Tag Values for Unassigned Cells	33
4.2.2	Cluster Growing Strategies	34
4.2.2.1	An Improvement for Tag Propagation	34
4.2.2.2	Valid Cell Pair List Creation Strategies	35
4.2.2.2.1	Selective Cell Pairs	35
4.2.2.2.2	Fixed Cell Pairs	35
4.2.2.2.3	Implicit Cell Pairs	36
4.2.2.3	Protocluster Merging Strategies	36

4.2.2.3.1	Explicit Merging	36
4.2.2.3.2	Implicit Merging Through Tag Propagation	37
4.3	Data Structures	37
4.3.1	Object Wrappers	37
4.3.2	General Data Structures	38
4.3.2.1	Constant Data	38
4.3.2.2	Event Data	39
4.3.2.3	Data Whose Nature Depends on the Chosen Strategy	39
4.3.3	The TrigTopoAutomatonCluster Class	39
4.4	Implementation	40
4.4.1	Outline	40
4.4.2	Preparing the Data	40
4.4.2.1	Constant Data Preparation	41
4.4.2.2	Event Data Preparation	41
4.4.3	Signal-to-Noise Calculation	41
4.4.4	Pair Creation	42
4.4.5	Cluster Growing	42
4.4.5.1	Strategies With Selective or Fixed Pairs	44
4.4.5.2	Strategies With Implicit Pairs	44
4.4.6	Final Cluster Properties Calculation	45
4.4.7	Packaging the Results	46
5	Topo-Automaton Clustering Optimization, Validation and Performance	47
5.1	Experimental Conditions	47
5.1.1	Run System	47
5.1.2	Samples	47
5.2	Optimization Studies	48
5.2.1	Optimization Procedure	48
5.2.2	Optimization Results	49
5.3	Physical Validation of the Results	52

5.3.1	Validation Procedure	52
5.3.2	Cluster Matching	53
5.3.3	Validation of the Cluster Identification	55
5.3.4	Validation of the Physical Properties Reconstruction	59
5.3.5	Effect of Cell Differences	63
5.3.6	Properties of the Unmatched Clusters	65
5.4	Performance of the Topo-Automaton Clustering	68
5.4.1	Comparison with the CPU Algorithm	69
5.4.2	Decomposition of the Processing Times of the GPU Implementation	70
6	Conclusions	73
6.1	Optimization of the Algorithm	73
6.2	Validity of Physical Reconstruction	75
6.3	Time Performance of the Algorithm	75
6.4	Future Work	76
	References	79
A	Code for the Topo-Automaton Clustering Implementation	A.1
A.1	General Definitions	A.1
A.2	Data Structures	A.2
A.2.1	Object Wrappers	A.2
A.2.2	General Data Structures	A.3
A.3	Implementation	A.6
A.3.1	Preparing the Data	A.6
A.3.2	Signal-to-Noise Calculation	A.7
A.3.3	Pair Creation	A.8
A.3.4	Cluster Growing	A.9
A.3.5	Final Cluster Properties Calculation	A.16

List of Figures

2.1	CERN accelerator complex and associated experiments.	4
2.2	Layout of the LHC.	5
2.3	Render of the ATLAS detector.	8
2.4	Render of the ATLAS Inner Detector.	11
2.5	Render of the ATLAS Calorimeters.	11
2.6	Cumulative amount of material as a function of the pseudo-rapidity.	12
2.7	Render of the ATLAS Muon Spectrometer.	13
3.1	Layout of the execution model and architecture of a CUDA-compatible GPU.	24
3.2	Client-Server structure of the previous GPU accelerated prototype.	28
5.1	Distribution of the differences between number of clusters per event in the GPU and CPU implementations.	56
5.2	Distribution of the number of clusters per event.	56
5.3	Distribution of the number of unmatched clusters per event.	57
5.4	Distribution of the number of differently attributed cells per cluster.	58
5.5	Distribution of the number of differently attributed growing or seed cells per cluster.	58
5.6	Fraction of clusters with more differently attributed cells than the abscissa.	59
5.7	Fraction of clusters with more differently attributed seed or growing cells than the abscissa.	59
5.8	Cluster energy distribution.	60
5.9	Cluster transverse energy distribution.	60
5.10	Variations of the cluster energies.	61
5.11	Variations of the cluster transverse energies.	61
5.12	Variations of the cluster pseudo-rapidities.	61

5.13 Variations of the cluster azimuthal angles.	62
5.14 Cluster distance based on reconstructed coordinates.	62
5.15 Cluster distance based on reconstructed coordinates (zoomed in).	63
5.16 Fraction of pairs of clusters with greater distance than the abscissa.	63
5.17 Relative reconstructed energy differences for clusters with the same cells.	64
5.18 Relative reconstructed transverse energy differences for clusters with the same cells.	64
5.19 Reconstructed pseudo-rapidity differences for clusters with the same cells.	65
5.20 Reconstructed azimuthal angle differences for clusters with the same cells.	65
5.21 Unmatched cluster size distribution.	66
5.22 Section of the overall distribution of cluster sizes.	66
5.23 Unmatched cluster energies.	67
5.24 Unmatched cluster transverse energies.	67
5.25 Unmatched cluster azimuthal angles.	68
5.26 Unmatched cluster pseudo-rapidities.	68
5.27 Event processing times for both implementations.	69
5.28 Ratio between GPU and CPU event processing times.	69
5.29 GPU speed-up of the algorithm (ratio between CPU and GPU event processing times).	70
5.30 Event processing times for the GPU implementation.	70

List of Tables

2.1	Expected output rates for the stages of the ATLAS trigger.	15
4.1	Maximum signal-to-noise ratio for guaranteed tag unicity.	32
5.1	Optimal parameters for signal-to-noise ratio calculation and tag initialization.	49
5.2	Optimal parameters for cell pair creation.	50
5.3	Optimal parameters for cluster growing.	51
5.4	Optimal parameters for final cluster properties calculation.	52
5.5	Decomposition of the average GPU event processing time.	71
5.6	Comparison between expected and measured execution times.	72

List of Abbreviations

ATLAS	A Toroidal LHC A pparatus
CERN	<i>Conseil Européen pour la Recherche Nucléaire</i> (now the European Organization for Nuclear Research)
CPU	Central P rocessing U nit
CUDA	Compute U nified D evice A rchitecture
GPGPU	General- P urpose Computing on G raphical P rocessing U nits
GPU	G raphical P rocessing U nit
HL-LHC	High L uminosity L arge H adron C ollider
LHC	Large H adron C ollider

Chapter 1

Introduction

1.1 Motivation

The Large Hadron Collider (LHC), the world's largest particle accelerator, is an invaluable tool for the study of particle physics. However, scientific advancement requires the study of increasingly less probable events in search of yet unexplained or not as well understood phenomena. In the case of particle accelerators, this translates to a need to increase the number of collisions that happen per unit of time, a parameter which is related to the *luminosity* of the detector. The LHC is slated to undergo, until 2027, the High-Luminosity Upgrade, which will represent an increase in the luminosity by a factor of about 4, leading to many more collisions happening at each instant. Even in its current state, the rate of collisions at the LHC is so high that no system can feasibly store all the information that could presumably be generated if every collision was being recorded. Instead, *trigger systems* are employed, to decide in real time which events are relevant and which can be discarded.

One of the experiments at the LHC is ATLAS, which is designed to detect a wide variety of physical processes. Thus, the ATLAS triggering systems are particularly important, using a variety of signatures (such as particles with high momentum in the transverse plane or the presence of tauons, for example), and, being real-time systems, their speed (and, thus, that of the algorithms that are used within them to make the decisions regarding the events) is paramount. Given the upgrades to the LHC, the ATLAS trigger must also be upgraded to be able to keep up with the increased luminosity.

There are two stages in the ATLAS trigger. The first is hardware-based, using custom-designed electronic circuits that also interface directly with the detector, which will be redesigned as part of the upgrade. The second, on the other hand, is software-based, running on commercially available components. This means that the necessary improvements in the speed of this second stage can be achieved by a combination of increasing the computational power available and improving and optimizing the algorithms that are used in it.

The present work will assess the possibility of taking advantage of the massive parallelism made possible

by Graphical Processing Units (GPUs) to accelerate one of the algorithms in this second stage of the ATLAS trigger, the one which is used to reconstruct the electromagnetic showers that take place within the calorimeters of the ATLAS detector. The algorithm in question, in its CPU implementation, is known as Topological Clustering, though the focus will be on a variant of it that is more amenable to GPU operations, the Topo-Automaton Clustering algorithm.

1.2 Objectives

The most fundamental aspect of the present work is producing a working implementation of the Topo-Automaton Clustering algorithm on GPUs using `CUDA`. Once that has been achieved, the primary objective is three-fold:

1. To study different alternatives for the optimization of that implementation in order to have the best performance possible.
2. To validate the results of the algorithm from the physical point of view, by comparison with those from the current, CPU-based implementation of the Topological Clustering algorithm.
3. To compare the execution time of the optimized GPU implementation with the existing CPU algorithm.

1.3 Thesis Outline

The present work begins with a brief introduction on the Large Hadron Collider and the ATLAS Experiment, together with the software that is used within the latter to control the trigger system and on which the implementation that will result from the present work will need to be integrated. The basic form of the Topological Clustering algorithm will also be presented. This corresponds to Chapter 2.

Next, the basics of GPU programming are outlined in Chapter 3, together with the full description of the Topo-Automaton Clustering algorithm. A previous attempt at parallelizing this same algorithm in a GPGPU context is summarized.

Chapter 4 presents the several implementation strategies that were considered and whose performance will be compared in order to optimize the algorithm. Some details of the classes and data structures used in the implementation are provided.

These different implementation strategies are compared in Chapter 5 and the physical validity of their results is investigated, by comparison with the reference CPU implementation. Finally, time measurements of the optimized version of the GPU algorithm are compared with the CPU reference and the overall speed-up made possible by GPU acceleration is calculated.

Chapter 2

The LHC and the ATLAS Experiment

2.1 CERN and the Accelerator Complex

The European Organization for Nuclear Research, which has kept the acronym of the organization that preceded it (the European Council for Nuclear Research, *Conseil Européen pour la Recherche Nucléaire*, hence CERN), was established in 1954 to operate one of the world's foremost laboratories for particle physics, located near the Swiss municipality of Meyrin, north-west of Geneva and close to the Franco-Swiss border. Since its foundation, it has helped push back the boundaries of human knowledge by studying increasingly more fundamental aspects of the functioning of the Universe, at the same time it has contributed to technological advancement, both directly due to the requirements of constructing the several particle accelerators and detectors it harbours, and indirectly through the general needs of large-scale scientific research and collaboration.

A complex chain of particle accelerators, capable of achieving increasingly higher energies, was gradually established as the need for exploring more energetic processes arose and newer experiments were designed. In its current iteration [1], shown in Figure 2.1, the particles begin by being accelerated in a **linear accelerator** (LINAC4 for protons, LINAC3 for heavy ions), then pass to the first circular accelerator (the **Proton Synchrotron Booster**, BOOSTER or PSB, for protons, and the **Low Energy Ion Ring**, LEIR, for heavy ions), which brings them up to a sufficient velocity (and hence energy) for the next stage, the **Proton Synchrotron** (PS). From there, the particles are sent to the **Super Proton Synchrotron** (SPS) and finally to the world's largest particle accelerator, the **Large Hadron Collider**, more commonly known as the LHC.

A diverse set of experiments take advantage of the particle beams created by the accelerator complex at its various stages, focusing on a broad range of physical phenomena, such as unstable nuclei (the **Isotope Separator On Line Device**, or ISOLDE [2]), antimatter (the **Antiproton Detector**, or AD [3]) or plasma wakefield acceleration (the **Advanced Wakefield Experiment**, or AWAKE [4]), besides the more generic explorations of the Standard Model of Particle Physics (and whatever might lie beyond it) made possible by high-energy particle collisions.

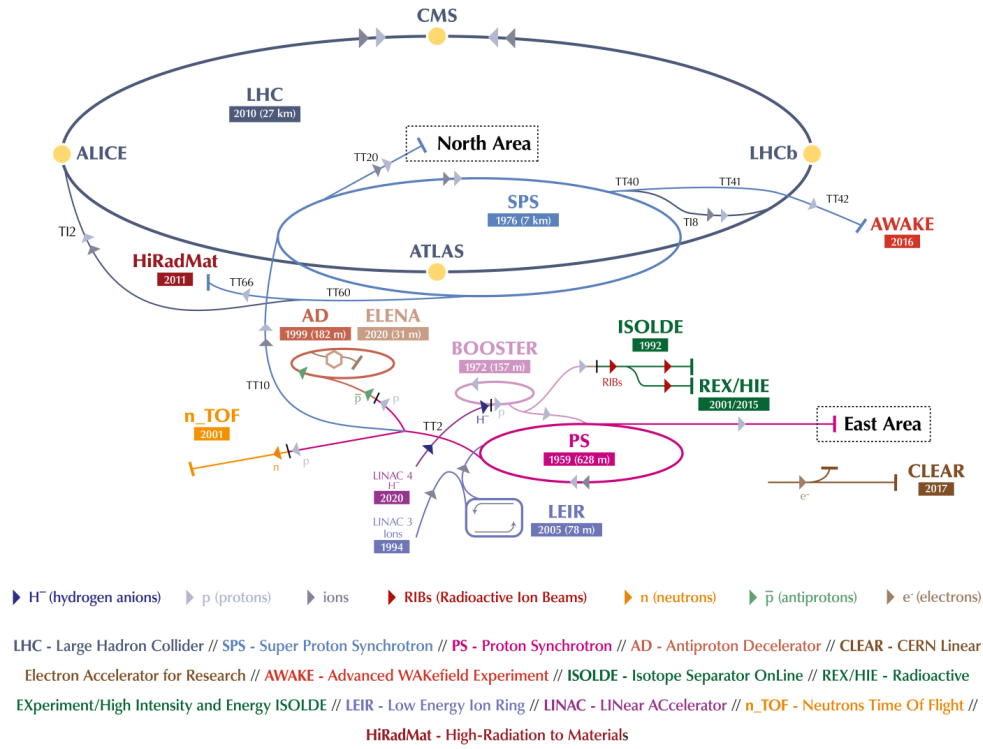


Figure 2.1: Diagram of the CERN accelerator complex and several of the experiments associated with it. Dates for the beginning of the experiments are provided. The coloured arrows indicate the flow of the different particle species within the accelerators. Source: Esma Mobs, CERN [5].

2.2 The LHC

The LHC, as stated before, is currently the largest particle accelerator in the world. With a circumference of about 27 km, it can accelerate protons to energies of up to 7 TeV (thus achieving proton-proton collisions with center-of-mass energies of 14 TeV) and heavy ions to energies of up to 2.76 TeV per nucleon (which, for the lead-208 ions that are commonly used, yields a center-of-mass energy for the ion-ion collisions of 1.15 PeV) [6].

These particles circle around the LHC in groups termed *bunches* of about 10^{11} particles each, as a consequence of the acceleration process, such that a point in the LHC is crossed by a bunch typically every 25 ns, which is equivalent to a frequency of 40 MHz. There are two separate rings within the LHC so that bunches can travel in opposite directions, creating the collisions that give the LHC its name when they are brought to meet at a slight angle. However, given the nature of sub-atomic particles, only some of the particles that comprise the bunches will actually interact in a meaningful way. The number of such interactions is the *number of collisions per bunch crossing*, μ , often conflated by abuse of language with the *pile-up*, which is the number of events simultaneously registered in a detector and that the electronics cannot properly separate (which means that, depending on the detector and on the physical processes that take place during the interaction, one can have a pile-up greater than the number of collisions per bunch crossing, as the events that took place during the previous crossing might still produce an effect on the electronics, for example).

As the LHC was built on tunnels excavated for a previous accelerator, the **Large Electron-Positron Collider (LEP)**, it inherited a structure in which there are eight arcs (with a length of about 2500 m) with eight short straight sections (with a length of about 530 m) between them, which was important to minimize synchrotron radiation losses in LEP. Despite all of the straight sections being dubbed *interaction points*, it is only on four of these points that the collisions actually take place. Figure 2.2 illustrates the overall layout of the LHC and the distribution of experiments along these interaction points.

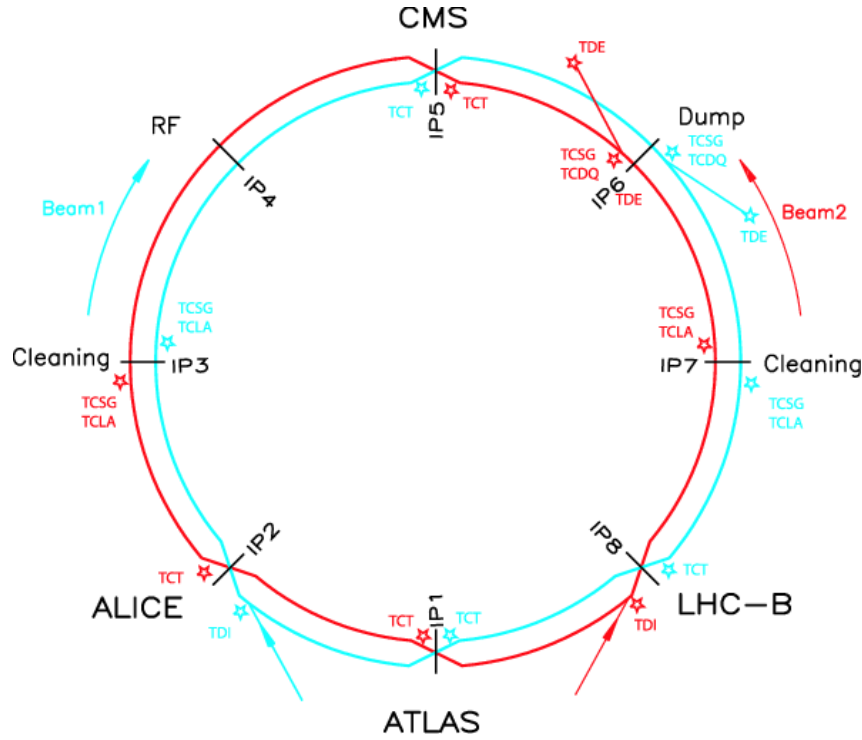


Figure 2.2: Schematic diagram of the layout of the LHC, showing the direction of the beams, the interaction points and the four main experiments. Stars mark the collimators and the targets for beam dumping. Source: Redaelli et al [7].

The currently existing experiments at the LHC and their respective locations are:

- ALICE (**A Large Ion Collider Experiment**) [8]: Interaction Point 2
- ATLAS (**A Toroidal LHC Apparatus**) [9, 10]: Interaction Point 1
- CMS (**Compact Muon Solenoid**) [11, 12]: Interaction Point 5
- LHCb (**LHC beauty**) [13]: Interaction Point 8
- LHCf (**LHC forward**) [14]: Interaction Point 1
- MoEDAL (**Monopole and Exotics Detector At the LHC**) [15]: Interaction Point 8
- TOTEM (**Total Elastic and Diffractive Cross Section Measurement**) [16]: Interaction Point 5

With an additional experiment, FASER (**Forward Search Experiment**) [17], being scheduled to begin taking measurements around 2022 at Interaction Point 1.

Given their larger scope and the larger space occupied by the detectors in the interaction points, ALICE, ATLAS, CMS and LHCb can be considered the “main experiments”, so they are the only ones shown in Figure 2.1 and Figure 2.2 (and in similar representations of the LHC). Among these main experiments, ATLAS and CMS are of particular significance as general-purpose detectors, equipped to detect a broad

range of particles, within and possibly beyond the Standard Model, and to study a variety of physical processes.

Especially for these two general-purpose experiments, but of general relevance to all of them, one should note that the 40 MHz rate at which the collisions happen (and, thus, the rate at which the events take place in the detectors) is much higher than any feasible transfer rate to a storage medium (not to mention the issues with finite capacity: even assuming just 1 KiB per event, which is unreasonably low given the diversity of information one would want to store, more than 38 GiB of data would be generated each second), which implies that there must systems in place to select the events that will be stored. Typically, there are multi-tiered *triggering systems*, both hardware and software-based, which will gradually reduce the data rate by applying increasingly more sophisticated selection criteria to the events, so that those that hold relatively low scientific interest can be discarded in favour of keeping only those that might contain physical processes that have been less extensively studied so far and/or that are the focus of current research.

2.2.1 Luminosity

The most useful metric to express the number of collisions that happen when the beams meet is the *luminosity*, $\mathcal{L}$, which can be taken as an intrinsic property of the accelerator. Under a set of simple assumptions that provide a reasonable approximation to the processes taking place in the LHC (namely that the two beams are identical, that the spatial distribution of particles within the beam is Gaussian and that the width of the bunches along the transverse direction is much greater than the width along the direction of propagation), that luminosity can be calculated as: [18]

$$\mathcal{L} \simeq \frac{N^2 N_b f}{4\pi w_t^2} \frac{1}{\sqrt{1 + \left(\frac{\theta}{2} \frac{w_l}{w_t}\right)^2}} \quad (2.1)$$

Where N is the number of particles per bunch, N_b the number of bunches that constitute each beam, f the revolution frequency of the beams, w_t the width of the bunches in the transverse direction at the interaction point, w_l the width in the longitudinal direction and θ the angle at which the beams meet.

The width of the bunches within the accelerator is not constant, due to the effect of the electromagnetic fields that are applied along their trajectory (and that make it possible for the trajectory to be curved). The usual parametrization is through the *Twiss parameters*, or *Courant-Snyder functions* [19], namely $\beta(s)$, which is related to the width of the bunch in the transverse direction in a position s along the trajectory through another property of the beams, their *emittance*, ε , that expresses the average dispersion of the particles within each bunch in phase space. Specifically, we have:

$$w_t(s) = \sqrt{\beta(s)\varepsilon} \quad (2.2)$$

We can introduce the concept of a normalized emittance, $\varepsilon_n = \varepsilon \gamma \frac{v}{c}$, where γ is the relativistic Lorentz factor and $v \simeq c$ is the velocity of the particles in the longitudinal direction. If we designate the value of

the β function at the point of collision by β^* and substitute in equation 2.1, we get an expression that presents the factors that affect the luminosity in a more expressive way: [18, 20]

$$\mathcal{L} \simeq \frac{N^2 N_b f \gamma}{4\pi \varepsilon_n \beta^*} \frac{1}{\sqrt{1 + \left(\frac{\theta}{2} w_1\right)^2 \frac{\gamma}{\varepsilon_n \beta^*}}} \quad (2.3)$$

All of these factors depend exclusively on the design and operating conditions of the accelerator, hence the previous remark that the luminosity can be taken as one of its intrinsic properties. This holds true also for some less simplified versions of this expression that try to take into account other processes that may affect the width of the beams at the point of collision [18].

The relation between the luminosity and the *cross-section*, which expresses the probability of an interaction between two particles taking place, means that, for an interaction with a given cross-section σ_I , the number of events that can be expected to be observed per unit of time will be:

$$N_{\text{events}} = \mathcal{L} \sigma_I \quad (2.4)$$

From this, we can conclude that, for events with low probability of occurring (and, thus, low cross-section) to be observed, high luminosities are required, a fact that has driven the evolution of both the CERN accelerator complex and of particle accelerators worldwide, as the more probable events (for a given center-of-mass energy) became better studied and understood, making the less probable ones more relevant for scientific advancement.

2.2.2 The High-Luminosity LHC

The operation of the LHC has been organized in a series of data acquisition periods (*Runs*) interspersed with interruptions (*Long Shutdowns*) that allow for maintenance and upgrades. Since 2018, the LHC is undergoing Long Shutdown 2, during which several improvements have been being done to allow for an increase in the luminosity and, hence, the capability of studying less probable physical processes, following the High-Luminosity LHC project (HL-LHC) [20]. Though the project is only scheduled to be completed in 2026, for the beginning of Run 4, some of the upgrades taking place during the present shutdown, especially in what regards the detectors of the main experiments, already anticipate at least partially what will be the operating conditions under the HL-LHC.

To quantify these improvements, the peak luminosity achieved during Run 2 was $\mathcal{L} \simeq 2 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$, with the interactions per bunch crossing averaging at $\langle \mu \rangle \simeq 60$. For Run 3, an increase by a factor of ~ 1.5 in the luminosity is expected, so that $\mathcal{L} \simeq 3 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$, with $\langle \mu \rangle$ also increasing to about 80.

The High-Luminosity Upgrade is expected to improve further on these parameters, bringing the luminosity up to $\mathcal{L} \simeq 8 \times 10^{34} \text{ cm}^{-2} \text{ s}^{-1}$ (a factor of 4 in relation to the conditions of Run 2) and the number of collisions per bunch crossing to $\langle \mu \rangle \simeq 200$ (a factor of about 3 in relation to Run 2), along with a possible increase in the maximum beam energy from 7 TeV to 7.5 TeV (which would allow collisions with center-of-mass energies up to 15 TeV). All of this places additional requirements on the performance of

the detectors, and in particular of their triggering systems, which must cope with the increased pile-up and the typically increased computational cost of applying the selection criteria to the *denser* events that come from it.

These increases in both luminosity and number of collisions per bunch crossing are made possible by improving the several factors present in equation 2.3, within the margin allowed by technological and design constraints. In particular, the number of particles in each bunch will be increased while β^* and ε_n will be decreased, which requires some adjustments to the accelerator complex and to the several beam collimation systems on the LHC. A more detailed discussion of these changes, their requirements and potential drawbacks or caveats can be found in [20].

2.3 The ATLAS Experiment

The ATLAS experiment (short for **A Toroidal LHC Apparatus**, as mentioned in the previous section) was proposed in 1994 to take advantage of the possibilities offered by the then recently approved LHC. From its inception, it was designed to be sensitive to as many different physical processes as possible, making it one of the two general-purpose detectors at the LHC. Figure 2.3 shows a computer-generated render of the ATLAS detector, along with its dimensions.

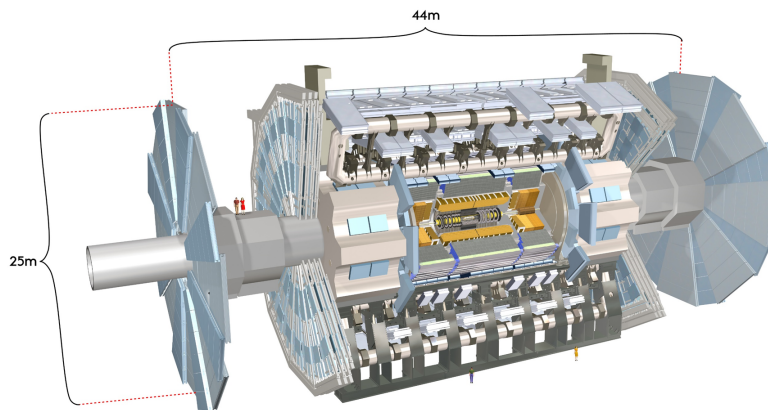


Figure 2.3: Render of the ATLAS detector, showing its structure and dimensions and also human figures, for scale. Source: João Pequeno, CERN [21] (adapted).

2.3.1 Coordinate Systems in the ATLAS Detector

To describe positions within the ATLAS detector, besides the common Cartesian (x, y, z) or spherical (ρ, θ, ϕ) coordinate systems, it is usual and useful to adopt a set of coordinates that provide some guarantees in terms of Lorentz invariance, since the particles involved in the collisions are highly relativistic and it is often necessary to employ different frames of reference, especially in terms of the velocity along the direction of the beam. Even in the case of Cartesian and spherical coordinates, it is important to establish a consistent definition, as some of the choices of axes are essentially arbitrary.

By convention [10], the interaction point is the origin of the coordinate system and the z axis is aligned with the beam direction (with one side of the detector - side A - being considered the positive direction while the other - side C - is the negative). The x axis is defined as pointing towards the center of the approximate circumference that is the LHC, while the y axis points vertically up (away from the center of the Earth). If adopting spherical coordinates, the azimuthal angle ϕ is measured around the beam direction in relation to the x axis while the polar angle θ will be measured relative to the former direction (all of which is consistent with the previous choice of z axis).

As for the coordinates with some Lorentz invariants, one must begin by realising that, under Lorentz transformations that change the velocity along the z axis, the momentum in that same direction (usually called *longitudinal momentum*, p_ℓ) is obviously not conserved, while the momentum in the remaining directions (the *transverse momentum*, p_t) is. This means that, while the azimuthal angle ϕ is left unchanged, θ , which is the angle between the beam direction and the trajectory of the particle, is not Lorentz invariant, which can be inconvenient. However, one can define a quantity $\mathcal{Y}$, called *rapidity*, that can be transformed additively when considering reference frame changes where only the velocity in the z direction varies: the rapidity of a particle in the new frame is simply the sum of rapidity of the particle in the old frame with the rapidity of the new frame in relation to the old frame, as one would do with velocities in a Galilean transformation. The rapidity may be calculated by: [10, 22]

$$\mathcal{Y} = \frac{1}{2} \ln \left(\frac{E + p_\ell c}{E - p_\ell c} \right) \quad (2.5)$$

However, this quantity has a significant drawback: to be calculated, both the value of the longitudinal momentum and that of the energy of the particle must be known, which may not always be possible or practical. One can use an alternative quantity, the *pseudo-rapidity*, η , that only depends on the values of the total and the longitudinal momenta, or, equivalently, on the angle between the trajectory and the beam direction: [10, 22]

$$\eta = \frac{1}{2} \ln \left(\frac{p + p_\ell}{p - p_\ell} \right) = \frac{1}{2} \ln \left(\tan \left(\frac{\theta}{2} \right) \right) \quad (2.6)$$

Given the usual expression for relativistic energies, $E^2 = p^2 c^2 + m^2 c^4$, it is easy to see that, in the limit where particle masses are negligible in comparison to their momentum (which tends to be the case in the highly relativistic collisions that happen at the LHC), that is, when $E \simeq pc$, expressions 2.5 and 2.6 are the same, which means the pseudo-rapidity can be taken to transform additively.

Thus, it is commonplace to describe the trajectories of particles in terms of their pseudo-rapidity, η , and azimuthal angle, ϕ , and one can further define a distance between two directions expressed in these quantities as $\Delta R = \sqrt{(\Delta\eta)^2 + (\Delta\phi)^2}$.

It is useful to keep in mind that, by its definition, a pseudo-rapidity of 0 corresponds to a direction that is perpendicular to the z axis, with the latter corresponding to the direction given by $\eta = \pm\infty$.

2.3.2 Structure and Geometry of the ATLAS Detector

As seen in Figure 2.3, the ATLAS detector has cylindrical geometry, being mostly symmetrical along ϕ and also under reflections by the transverse plane, that is, under changes of z to $-z$. Its structure can be divided into three concentric layers, from the nearest to the most distant to the interaction point: the *Inner Detector*, the *Calorimeters* and the *Muon Spectrometer*.

There are also two regions that can be considered within each layer of the detector: the *barrel* region, which generally corresponds to the central region of the detector, and the *end-cap* region, which, as the name implies, closes off the detector at higher values of $|z|$ to provide measurements for particles with greater (pseudo-)rapidities.

Now follows a brief overview of each of these layers, with some focus on the Calorimeters since the purpose of the present work is to develop trigger algorithms that operate on data from that layer. A much more detailed description of the ATLAS Detector is given in [10].

2.3.2.1 Inner Detector

The ATLAS Inner Detector, represented in Figure 2.4, is designed to detect charged particles with $|\eta| < 2.5$ and measure their position and momentum with good resolution. As this region of the detector is exposed to the magnetic field generated by a superconducting solenoid, of up to $B = 2\text{ T}$, an accurate reconstruction of the particles' trajectories, the *tracks*, is a crucial step in the identification of their nature, and, with it, of the physical processes that took place during the collision. This follows from the usual expression for the Larmor Radius, r_L , of a particle with charge q that is exposed to a magnetic field of intensity B :

$$r_L = \frac{p_T}{qB} \quad (2.7)$$

Where p_T is the component of the particle's momentum that is perpendicular to the applied magnetic field and we consider the radius to be positive when the particle curves in the direction given by $\vec{p} \times \vec{B}$ and negative otherwise.

The innermost part of the Inner Detector is comprised of Pixel Detectors, which are individual, silicon-based sensors that provide an accuracy in the track positions of about $10\text{ }\mu\text{m}$ in the transverse plane and $115\text{ }\mu\text{m}$ in the longitudinal direction. The Pixel Detectors are surrounded by the **Semiconductor Tracker (SCT)**, which is formed by a series of long silicon strips, providing accuracies of about $17\text{ }\mu\text{m}$ in the transverse plane and $580\text{ }\mu\text{m}$ in the longitudinal direction. Finally, we have the **Transition Radiation Tracker (TRT)**, which, as the name implies, takes advantage of the electromagnetic radiation emitted when a charged particle crosses a boundary between two different media, being based on polyimide drift tubes ("straws") containing a gas mixture to which sufficient voltage is applied to enable them to function as a proportional counter, with the time taken for the electrons resulting from the ionization of the gas in the tube to reach the anode (the *drift time*) being used to determine the position of the incoming particle. The TRT only measures positions on the transverse plane, with an accuracy of $130\text{ }\mu\text{m}$, though the greater distance from the interaction point enables this part of the detector to contribute to a more

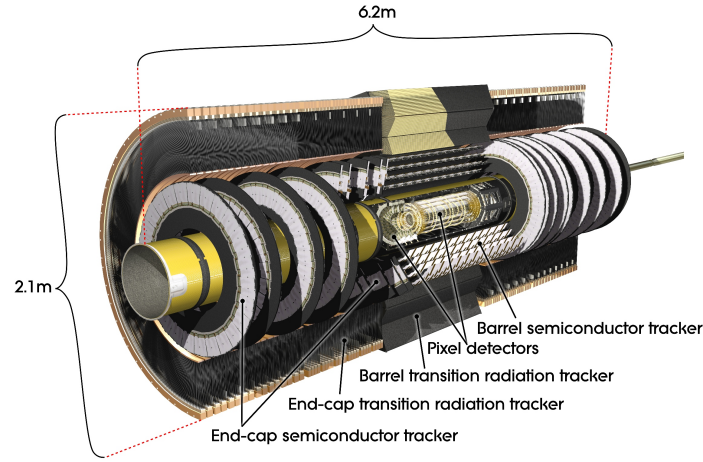


Figure 2.4: Render of the ATLAS Inner Detector, showing some of its constituent elements. Source: João Pequeno, CERN [23].

precise measurement of the momenta of the particles.

For the High-Luminosity LHC upgrade, the Inner Detector is planned to be replaced with the Inner Tracker [24], which will remove the TRT in favour of an entirely semiconductor-based constitution, with several pixel detector layers surrounded by strip detectors.

2.3.2.2 Calorimeters

The calorimeters, as could be expected, are designed to measure the energy of incoming particles. They are comprised of many different cells (a total of $N_{\text{cells}} = 187652$) that cover a small range in η and ϕ , with the entirety of the calorimeters being able to detect particles with pseudo-rapidities up to $|\eta| = 4.9$, though with greater granularity for $|\eta| < 2.5$ (the same η range of the Inner Detector). Figure 2.5 shows the structure of this layer.

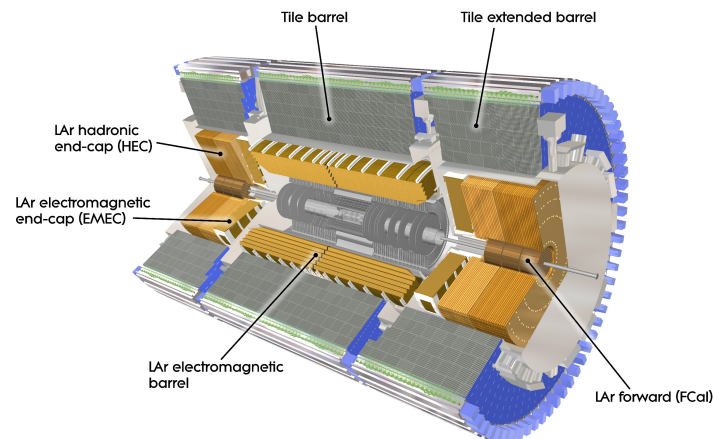


Figure 2.5: Render of the ATLAS Calorimeters. Source: João Pequeno, CERN [25].

This calorimeter layer can be divided in an inner, liquid-argon based Electromagnetic Calorimeter (LAr Calorimeter or EM Calorimeter) and an outer Hadronic Calorimeter. Before the Electromagnetic Calorime-

ter, there is an additional Pre-Sampler to correct for energy losses in the Inner Detector, and both the calorimeter and pre-sampler can be separated between the barrel (EM Barrel and Pre-Sampler B), covering the pseudo-rapidity range $|\eta| < 1.475$, and end-cap regions (EM End-Cap, or EMEC, and Pre-Sampler E), for $1.375 < |\eta| < 3.2$. As for the Hadronic Calorimeter, the region with $|\eta| < 1.7$ is covered by the Tile Calorimeter, which is comprised of a barrel ($|\eta| < 1.0$) and an extended barrel in each z direction ($0.8 < |\eta| < 1.7$), and the higher pseudo-rapidities are covered by two other liquid-argon based calorimeters, the **Hadronic End-Caps** (HEC), for $1.5 < |\eta| < 3.2$, and the **Forward Calorimeters** (FCal), for $3.0 < |\eta| < 4.9$. With the exception of the Pre-Sampler, all the calorimeters are made up of several layers, which, together with the overlap in η (as Figure 2.6 shows), ensures that most of the energy of the incoming particles is deposited in this part of the detector.

All of these calorimeters function on the basis of detecting the electromagnetic showers generated by the passage of a particle through an active medium, such as the liquid argon present in most of them or, in the case of the Tile Calorimeter, organic scintillators. To maximize the chances of an incoming particle interacting with the calorimeter and generating these showers, absorbers made of appropriately dense materials are included (lead for the EM Calorimeter, steel for the Tile Calorimeter, copper for the HEC and some of the layers of the FCAL and tungsten for the remaining layers of the FCal). The likelihood of a particle interacting with the material is best expressed by the interaction length, which can be seen as the mean free path, λ , between two inelastic interactions of the incoming particle with the material; as such, for a length L , the average fraction of incoming particles that can pass through without interaction is given by $e^{-L/\lambda}$. Figure 2.6 shows the number of interaction lengths within the calorimeter as a function of the pseudo-rapidity.

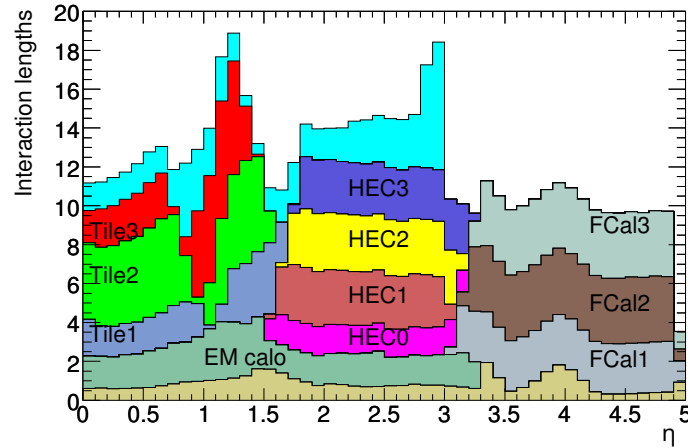


Figure 2.6: Cumulative amount of material, in units of interaction length, as a function of the pseudo-rapidity of the incoming particle. The different parts of the calorimeter are identified in different colours, with the amount of material closer to the interaction point than the calorimeters at the bottom, in brown, and the amount of material corresponding to additional parts of the detector infrastructure before the Muon Spectrometer at the top, in cyan. Source: The ATLAS Collaboration [10] (adapted).

One should note that, though complete η coverage is assured, for $z = 0$ (and, thus $\eta = 0$), the Electromagnetic Calorimeter has a 4 mm gap between the two halves that make up the barrel for $z > 0$ and $z < 0$, while, for $1 \lesssim |\eta| \lesssim 1.5$, there is the *crack*, the separation between the barrel and the end-cap, for

the Electromagnetic Calorimeter, or the barrel and the extended barrel, for the Tile Calorimeter.

2.3.2.3 Muon Spectrometer

The outer parts of the ATLAS detector are once again dedicated to the detection of charged particles. Given that the rest of the detector lies between this layer and the interaction point, only particles that do not interact significantly with matter and have a sufficient lifetime can reach it. The only known charged particle that fulfills these requirements is the muon, hence this layer being called a muon spectrometer.

It is built around three toroidal magnetic fields, one for each end-cap and one for the barrel, which will curve the incoming particles so their momentum, and, hence, their energy, can be determined, by another application of expression 2.7. Muon energies from ~ 3 GeV up to ~ 3 TeV can be reconstructed with precision, with the detector covering the region with $|\eta| < 2.7$.

There are four types of chamber employed in the muon spectrometer, two for precise detection, **M**onitored **D**rift **T**ubes (MDT), for $|\eta| \geq 2$, and **C**athode **S**trip **C**hambers (CSC), for $2 < |\eta| < 2.7$, and the other two mainly for the trigger system, **R**esistive **P**late **C**hambers (RPC), for the barrel, and **T**hin **G**ap **C**hambers (TGC), for the end-cap. All of these are based on the same physical principles as the Inner Detector's Transition Radiation Trackers, namely the measurement of the drift time of the electrons (and, in the case of the Cathode Strip Chambers, the ions) generated by ionization of a gas under a sufficient voltage to behave as a proportional counter to determine the position of the cause of that ionization, which, in the muon spectrometer's case, corresponds to the passing of a muon. Figure 2.7 shows the components of the muon spectrometer.

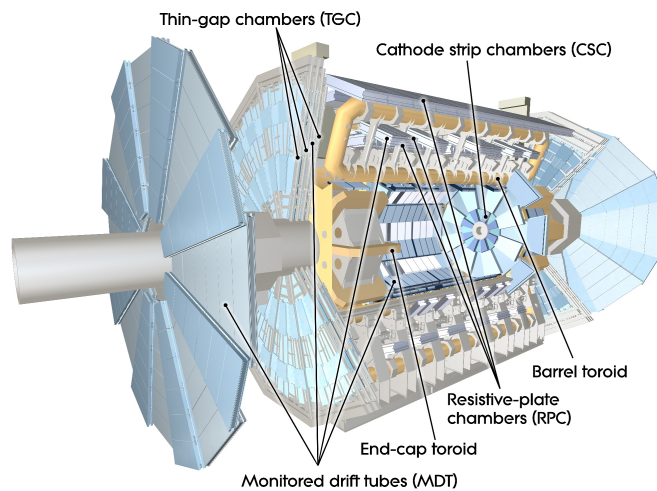


Figure 2.7: Render of the ATLAS Muon Spectrometer. Source: João Pequeno, CERN [26].

2.3.3 The ATLAS Trigger and Trigger Upgrades

As mentioned in section 2.2, the expected collision rate at the LHC is 40 MHz, which makes it unfeasible to store every event. Thus, a triggering system must be in place to filter the incoming data down to a more manageable rate for storage. Within the ATLAS trigger, two stages of event processing can be distinguished, the first performing a coarser selection using custom-made electronic circuits, some of

which interface directly with the detector hardware, and the second being based on software running on commercially available components, on a “computer farm”.

As the computational cost of processing an event depends significantly on its complexity, which is directly related to the luminosity of the detector, the ATLAS trigger has undergone a series of upgrades following the evolution of the LHC, as has been detailed in section 2.2.2. In particular, in anticipation of Run 3, *Phase I* [27] has been implemented to improve the existing trigger system, while *Phase II* [28] has been planned to accommodate the High-Luminosity LHC (which will correspond to Run 4 and beyond). Though both of these phases represent significant improvements in the performance and throughput of the trigger, the general structure remains the same.

The first stage, Level 1 (L1) up until Phase II and Level 0 (L0) after the latter, bases its assessment mainly on information from the calorimeters and the muon spectrometer, searching for electromagnetic showers, muons with large momenta in the transverse plane, missing energy also in the transverse plane or the presence of some other particles of interest (such as electrons or taus). Besides the necessary infrastructure for reading from the detectors and transferring the necessary data to the next stage of the trigger, this first stage contains four sub-systems:

- L1/L0 Calo: Uses coarse information from the calorimeters to find possible electrons, taus or photons, calculate missing transverse energies and identify jets (which are sets of particles that travel in the same general direction, typically generated by a decay).
- L1/L0 Muon: Calculates muon trajectories based on the muon spectrometers and the outermost layers of the tile calorimeter.
- Topological Processor (L1Topo)/Global Processor (GP): Applies several geometrical criteria to the positions of the particles and showers identified by the other parts of the trigger, to better select the events that may be of interest. In the case of the Global Processor, for Phase II, a more precise reconstruction of particle trajectories and overall object recognition is also performed, taking into account finer information from the calorimeters.
- Central Trigger Processor (CTP): Collects all the information from the other trigger subsystems and uses it to make a final decision regarding event acceptance. Also identifies *Regions of Interest* (RoI), portions of the detector where, based on the information gathered so far, there may be features of interest.

The second stage is either known as *High-Level Trigger* (HLT), for Phase I, or as *Event Filter* (EF), for Phase II. As stated before, this stage is handled in software and runs a variety of sophisticated algorithms to more precisely reconstruct the events that were accepted by the first stage of the trigger, possibly taking advantage of the Regions of Interest identified by the CTP to reduce the computational cost by limiting the reconstruction to those regions. Given the nature of the physical processes detected in each layer of ATLAS and the diversity of events that might be of interest, to try and list all such algorithms would be a task far beyond the scope of the present work; for now, we must mention that the reconstruction of the showers that happen within the calorimeters is typically performed through

clustering algorithms, which will be further addressed in section 2.4, given that it is precisely one of such algorithms that will be implemented and optimized.

Table 2.1: Expected output rates for the two stages of the ATLAS trigger, for both Phase I and Phase II (for the High-Luminosity LHC) of the ATLAS trigger upgrades. Data input rate for the trigger is the LHC collision rate of 40 MHz.

Phase I		Phase II	
Trigger Stage	Output Data Rate	Trigger Stage	Output Data Rate
Level 1	100 kHz	Level 0	1 MHz
High Level Trigger	1 kHz	Event Filter	10 kHz

In Table 2.1, we can see the expected output rates for the two stages of the trigger. In general, Phase II will imply a tenfold increase in the data processing rate, which increases the overall computational load by a factor of 10 as well. The increased luminosity under the HL-LHC will also increase the pile-up in the detector, making each individual event more complex, as there are more signals being registered during the event, resulting in more particles, showers and so on, which further increases the computational load on the trigger system.

For Level 0, this may be addressed by redesigning the circuits to meet the new specifications. But, for the software-based Event Filter, there are two possible, complementary approaches: on one hand, the increased computational load can be dealt with by increasing the overall computational resources available at the “farm”, and, on the other hand, the algorithms themselves should be adapted and optimized to ensure the best possible usage of the available resources.

An additional possibility, and one that is central to the objective of the present work, is the adoption of accelerators, that is, more or less specialized hardware that can offload part of the operations that would normally be performed in the CPU, with possible performance gains due to their design being better suited to some kinds of operations than the latter (which needs to offer good performance across most kinds of computational tasks). This, in a way, combines both approaches, in the sense that the available computing power increases, but typically algorithms need to be redesigned to better leverage the capabilities of the accelerators. There are several possible choices for these accelerators, with some of the most common being **F**ield **P**rogrammable **G**ate **A**rrays (FPGAs), which can implement (potentially very complex) logic circuits in a reconfigurable way, and **G**raphics **P**rocessing **U**nits (GPUs), which excel at performing the same computations over a large amount of data in parallel. It is the latter choice that is of interest for this work, and the benefits and pitfalls of GPU acceleration will be discussed in some detail in the next chapter.

2.3.4 Athena

Athena [29, 30, 31] is the software framework used within the ATLAS Experiment for interfacing with the detector and/or the hardware-based trigger, implementing the software-based trigger (“online”) and analysing the events that have been stored in greater detail without the temporal constraints of experimental operation (“offline”). It consists of a large C++ code-base with a significant additional amount of

Python code for configuration.

For the current version of Athena, labelled AthenaMT [32, 33] due to the fact that it has been designed with multi-threading in mind (contrarily to the previous versions, which focused on a more traditional serialized structuring of the code), there is essentially no distinction between online and offline processing of event data: it is perfectly possible to apply the algorithms that would be used for complete and rigorous offline event reconstruction in the trigger (or vice-versa). The calorimeter reconstruction algorithm that is the object of the present work is precisely one of these cases, being usable both online and offline, though the focus of the work is on its usability as part of the ATLAS trigger system.

Given the size and complexity of the Athena code-base, it is close to impossible to give an adequate overview of it. We will instead focus merely on the portions that are relevant for the present work, both in terms of the constraints placed upon the structure of the code and the features of Athena of which we will take advantage.

2.3.4.1 Data Processing Within Athena

Within AthenaMT, there are essentially three ways of classifying the data processing that might be employed [32]: *Services*, which are meant to exist independently of event processing and should support being called or accessed by multiple threads, *Algorithms*, which process each event in a single thread (but which, if needed, might be instantiated several times to process events in parallel), and *AlgTools*, which are specific data processing steps that may be called within an Algorithm (potentially multiple times per event). The Algorithms and AlgTools process events through their `execute` method, which receives as an argument an `EventContext` to hold information regarding the event being processed.

The implementation that results from the work here presented corresponds to an `AlgTool` meant to be called from within the clustering algorithm that is applied to the calorimeter in the trigger, `TrigCaloClusterMakerMT`. This means our implementation must be a class derived from `AthAlgTool`, and, since it will populate a collection of the clusters that can be found in the event (which, being part of the event storage model, are a `xAOD::CaloClusterContainer`), it must also derive from `CaloClusterCollectionProcessor`.

A series of algorithms specified to be executed one after the other constitutes a chain, which lies at the heart of the implementation of the ATLAS trigger within AthenaMT [34]. For the trigger, the chains typically begin with a *Filter Algorithm*, to allow for early rejection of events that do not fulfil the criteria for further processing, followed by an *Input Maker Algorithm*, to create an *Event View* to encapsulate the Regions of Interest identified in the hardware-based trigger in such a way that the algorithms to operate on them can be written the same as those that operate over the entirety of the detector. Afterwards, the desired reconstruction algorithms are run and the chain ends with a *Hypothesis Algorithm*, which, as the name implies, will test an hypothesis regarding the nature of the event being processed, based on a set of predefined physical signatures specified in the *trigger menus*.

2.3.4.2 Data Storage and Retrieval

The multi-threaded model of AthenaMT implies that every data read and especially every data write needs to be carefully considered, in particular when data dependencies between different algorithms may arise. A Service called `StoreGate` provides several facilities to handle these data accesses safely, of which we may point out `SG::ReadHandle<T>`, which provides what can be seen essentially as a read-only pointer to a stored object of type `T`, and `SG::ReadCondHandle<T>`, which is quite similar except for the fact that the object may change depending on the conditions during the event that is being processed (which is useful for properties that may change during a Run, such as the noise associated with the energy measurements in the cells of the calorimeters).

Of interest is also the `DataLink<T>` class, which may (or may not) refer to an object managed by `StoreGate`, enabling deferred reading.

2.3.4.3 Calorimeter Data

The most important class for obtaining calorimeter data is `CaloCellContainer`, which (as the name implies) holds all of the cells of all the calorimeters, which are represented through the `CaloCell` class. The cells may be indexed through their `IdentifierHash`, which are guaranteed to be unique and sequential, starting from 0 up to $N_{\text{cells}} = 187652$ (as mentioned in section 2.3.2.2). Of obvious importance (and physical meaning) is their `energy()` member function.

The `CaloNoise` class provides access to the noise associated with the energy measurements in the calorimeter, through the `getNoise` member function (which may take as arguments the `ID()` and `gain()` of a `CaloCell`). This noise must take into account both the electronic read-out system of the calorimeters and the pile-up [35].

Finally, to get the geometrical description of the calorimeters, one must use a `CaloDetDescrManager`, whose `get_element` member function will give access to a `CaloDetDescrElement` corresponding to the cell with the `IdentifierHash` provided as an argument. The `CaloDetDescrElement` holds, among other information, the cartesian and spherical coordinates of the cell and its pseudo-rapidity. Furthermore, the `getCaloCell_ID` member function of the `CaloDetDescrManager` returns a pointer to a `CaloCell_ID`, which, through its `get_neighbours` member function, can provide information on the adjacency of the cells to one another, which (as we will see in section 2.4) is crucial for the algorithm that will be studied. A final mention must be made to `LArNeighbours::neighbourOption`, which specifies the way in which the adjacency between cells will be defined.

2.3.4.4 Calorimeter Cluster Information

As has been previously mentioned and will be further explained in the next section, the algorithm that is the focus of this work is intended to assist in calorimeter event reconstruction through the formation of clusters of cells to approximate the electromagnetic showers. As such, it is important to consider the way in which the information pertaining to these clusters is stored within Athena.

As stated in section 2.3.4.1, the algorithm will have to fill a `xAOD::CaloClusterContainer`, which, as the name implies, is a container for objects of the type `xAOD::CaloCluster`. The latter is the class that actually holds the cluster information, such as the total energy, transverse energy, pseudo-rapidity and azimuthal angle, and it also includes a `CaloClusterCellLink`, which specifies which cells in the calorimeters belong to the cluster.

2.4 Topological Clustering

An important part of the reconstruction of the physical processes taking place within the calorimeter is the identification of the showers generated by the incoming particles. To achieve this, it is necessary to identify which cells within the calorimeters belong to which shower, so that its location and physical properties may be calculated. The most natural way of doing this is through *clustering algorithms*, which, as the name implies, create *clusters* of cells where the energy deposition has been sufficiently high, typically around a local maximum.

A simple way of performing this clustering is through the *Sliding-Window* algorithm [36], which considers a fixed number of cells surrounding a local maximum of the deposited energy. This is employed mostly for reconstructing showers originated by electrons or photons, especially in the hardware-based part of the trigger [28], as these showers tend to be less extensive as those generated by other, heavier particles.

A more comprehensive, though more computationally demanding, option is to consider a variable number of cells, growing the clusters outwards from cells with significant energy deposition. This may be done through the *Topological Clustering* algorithm [35, 36], which is the object of the implementation and optimization efforts of the present work.

As mentioned in 2.3.4.3, the calorimeter cells can be indexed from 0 up to N_{cells} . To implement the Topological Clustering algorithm, one must also be able to define a function $E(i)$ that returns the energy deposited at the cell with index i , as well as another function $N(i)$ that returns the noise associated with that energy measurement, which takes into account both the electronic read-out of the calorimeter and the pile-up. One must also provide three non-negative real numbers to serve as thresholds for cell classification, the *seed threshold* T_{seed} , the *growing threshold* T_{grow} and the *cell threshold* T_{cell} , with $T_{\text{seed}} > T_{\text{grow}} \geq T_{\text{cell}}$. Common choices of thresholds are $(T_{\text{seed}}, T_{\text{grow}}, T_{\text{cell}}) = (6, 3, 3)$, which may be used to reconstruct showers in the electromagnetic calorimeter [36], and the more general $(T_{\text{seed}}, T_{\text{grow}}, T_{\text{cell}}) = (4, 2, 0)$, which may be used for all calorimeters [36]. The latter choice will be adopted throughout the present work, as it is also the default used in ATLAS [35].

The Topological Clustering algorithm is as follows:

1. For every cell of the calorimeters, indexed by c :
 - Calculate the signal-to-noise ratio of the deposited energy at the cell as $\text{SNR}(c) = \frac{|E(c)|}{N(c)}$.
 - Classify the cell:

- If $\text{SNR}(c) > T_{\text{seed}}$, it is a **seed cell**.
 - If $T_{\text{seed}} \geq \text{SNR}(c) > T_{\text{grow}}$, it is a **growing cell**.
 - If $T_{\text{grow}} \geq \text{SNR}(c) > T_{\text{cell}}$, it is a **terminal cell**.
 - If $T_{\text{cell}} \geq \text{SNR}(c)$, it is an **invalid cell** and may not be part of any cluster.
2. Create the list of currently evaluated cells, the **current cell list**, and fill it with all the **seed cells**
 3. Create an empty list, the **next cell list**
 4. Sort the **current cell list** by the signal-to-noise ratio, SNR, of its cells.
 5. For every member of the **current cell list**, create a **protocluster** that contains just that cell.
 6. While the **current cell list** is not empty:
 - (a) For every cell c in the **current cell list**:
 - Let P_c designate the **protocluster** to which the cell belongs.
 - For every direct neighbour n of that cell:
 - If the neighbour cell does not belong to a **protocluster**:
 - * Add the neighbour cell to P_c .
 - * If $\text{SNR}(n) > T_{\text{grow}}$ (n is a **growing cell** or a **seed cell**), add n to the **next cell list**.
 - Else, if it belongs to a protocluster P_n and $\text{SNR}(n) > T_{\text{grow}}$ (n is a **growing cell** or a **seed cell**), merge P_c and P_n .
 - (b) Replace the contents of the **current cell list** with those of the **next cell list** and empty the latter.
 7. The **protoclusters** that remain are the final clusters and their physical properties may be calculated from the cells that comprise them.

Where cluster merging is defined as taking all the cells that belong to the cluster whose initial seed cell had the lowest signal-to-noise ratio and adding them to the other cluster.

After this cluster growing procedure, and depending on the objectives behind the analysis being performed, one could apply a minimum threshold in terms of the energies of the clusters. Another important step, but one whose implementation and optimization is outside the scope of the present work, is Cluster Splitting, which takes the clusters that have been formed by this algorithm and, taking into account local maxima of the signal-to-noise ratio within each cluster, tries to distinguish the showers originated by different incoming particles.

One important aspect that should be remembered, as it will be of some relevance later on, is that the terminal cells will belong to the cluster that can reach them in the smallest amount of steps, and, in case

two clusters can reach it in the same amount of steps, to the one that originated in the seed cell with greatest signal-to-noise ratio.

There is another detail of the algorithm that should be pointed out: the absolute energy is used for cell classification since, due to the interaction between the fluctuations associated with the random noise of the measurements and the fact that there must be some form of calibration to determine the relation between the electric signal and the actual deposited energy, some cells might register negative energies, which are not associated with a real shower. Since the fluctuations can be considered to be random, the distribution of noise should be symmetric in relation to the y axis (an even function), which means that including these cells with negative energies in the clusters helps in offsetting the additional positive energy contributions due to noise. On the other hand, one may even have *noise clusters* (both with positive and negative energies, with an energy distribution that can also be expected to be symmetric in relation to the y axis), which are not associated with any shower, being the result of cases where the fluctuations in the measurements in several adjacent cells were sufficient to pass the signal-to-noise ratio thresholds.

Chapter 3

Graphical Processing Units as Accelerators

Graphical Processing Units, as they are known today, appeared around 1980 [37] as a way to accelerate the display of text and graphics on computer screens (hence their name). First focused mostly on 2D operations given the computational power available at the time, as computer hardware and the electronic entertainment industry evolved, GPUs rapidly transitioned to providing hardware acceleration for 3D graphics rendering. Since 3D models are generally represented through polygons and the display consists (in its most basic form) of a projection of these polygons onto a 2D plane (together with the rasterization procedure that computes the colour of each pixel on the screen), this requires a significant amount of relatively simple arithmetic and trigonometric operations, between numbers that may not be integers; as the precision required for displaying 3D graphics is not too high, 32-bit floating point variables are usually chosen to represent the various quantities. One remarkable property of the computations necessary to display 3D computer graphics is that they are highly parallelizable: each polygon can be rendered separately, the colour of each pixel can be computed separately, and so on.

All of this means that the design goals of GPUs differ significantly from those of the CPUs for which programs are more commonly written. While CPUs were designed to accommodate a variety of tasks and purposes with reasonable performance, relying for most of their history on serial execution (with the multi-threaded operation that has become commonplace being a relatively recent development brought about by the physical limitations of clock speeds being reached [38], bringing an end to the decades-old trend in the evolution of processors commonly known as Moore's Law [39, 40]), GPUs have had from the start a much greater focus on parallel execution of relatively simple operations, mainly on 32-bit and especially 32-bit floating point types.

As opposed to a CPU, where memory accesses are a significant part of normal execution and branching is an usual operation, the performance of GPU operations often suffers quite significantly because of sub-optimal memory access patterns and conditional execution. Because of these and other constraints,

programming on GPUs requires a different mindset from “usual” CPU programming, and, in general, algorithms that are well suited for CPUs will likely need to be redesigned (or altogether substituted) to be performant in GPUs and vice-versa.

3.1 GPU Programming Languages

In the beginning, GPUs were not properly programmable, having a mostly fixed processing pipeline geared for their main purpose of rendering 3D graphics. Gradually, as the need for more control over the final result of that rendering arose (mostly as a consequence of the growing market for computer entertainment), the possibility of modifying the transformations applied to the geometry and pixel data was provided, first through programmable “shaders” in a very low level language that exposed the underlying functionality of the GPU, and then with some higher-level abstractions, but all of them focused mostly on the kinds of operation necessary for 3D rendering. As the computing power of GPUs grew and CPU processing speeds started to stabilize, the highly parallel design of the former made them attractive for some kinds of computational problems, which ultimately led to the development of the programming paradigm commonly known as *General-Purpose computing on GPUs*, or GPGPU.

Though evolution in this field is still ongoing, two languages, or rather, two frameworks that build upon the C/C++ family of languages (with additional bindings available for other languages, such as Fortran or Python) have established themselves as the most common options for GPU programming: **Compute Unified Device Architecture**, more commonly known as CUDA [41, 42], and `OpenCL` [43]. CUDA was developed by the Nvidia Corporation and is mostly locked to devices from that same manufacturer, while `OpenCL` is an open standard maintained by the Khronos Group that is intended for maximum compatibility and portability, but, since interfacing with the actual devices tends to be dependent on manufacturer-specific drivers, the level of support may vary¹.

Whenever there are competing standards, especially when they have an impact on a market as important as computer hardware, it can be difficult to determine without the gift of hindsight which one is the better option. Though, as a matter of principle, standards that strive to be open and support as many different platforms as possible are more desirable and more in line with the spirit of Science and Academia, there is some evidence [44, 45, 46, 47] that, in general, better performance can be leveraged in CUDA with less programming effort, though this is highly dependent on the specific task and hardware, with `OpenCL` performing as good as CUDA or even better in some other contexts.

Given the author’s own familiarity with CUDA and the fact that there have been other attempts at using this framework in the context of the ATLAS triggering software (such as [48, 49]), it was the one chosen in the present work to implement the intended GPU acceleration.

¹In particular, on devices capable of running CUDA, support for newer OpenCL features tends to be lower.

3.2 CUDA Execution Model and Architecture

The main computational unit in CUDA is the *kernel*, which describes the operations that will be performed on the GPU. When a kernel is submitted to the GPU (“launched”), two additional parameters must be provided: the block and grid dimensions. To understand the relevance of these parameters, it is important to consider the overall architecture of the GPU and the way the several tasks are performed within it.

The most important element of the GPU is the *Streaming Multiprocessor*, which will control the execution of a *block* of *threads*. Each thread is the basic parallel execution unit, but, unlike what happens in CPU-based multithreading, they are not entirely independent: each group of 32 threads, called a *warp*, tends to be in synchrony², with every instruction being executed by the whole warp; this means that, whenever there is a conditional execution (such as an `if`) where the threads within the warp follow different branches, both branches must be evaluated (though the threads that follow a different branch are disabled, so there are no side-effects of evaluating both branches except for the increased run time and energy consumption). This fact that the threads will all execute essentially the same instructions leads to the CUDA architecture being sometimes dubbed SIMT (**S**ingle **I**nstruction – **M**ultiple **T**hreads) to highlight its differences in relation to the usual SIMD (**S**ingle **I**nstruction – **M**ultiple **D**ata) model that is part of Flynn’s Taxonomy of computer architectures [50].

Each warp benefits from the possibility of *coalescing* memory accesses whenever they are aligned with the cache lines that exist within the Streaming Multiprocessor, as well as several voting, sharing or reduction operations on local memory, depending on the *Compute Capability* of the device (which, broadly speaking, characterizes both the hardware capabilities of the device and the availability of some CUDA intrinsic functions for more low-level control over the execution). These warps can then be grouped in a block, along one, two or three dimensions, though a block can contain no more than a total of 32 warps (1024 threads in total) [42].

Each block will be executed by a Streaming Multiprocessor independently of the others. By design, the order in which the blocks are executed is unspecified, which allows for significant freedom in the scheduling of their execution, helping achieve optimal utilization of the available computing power. The set of all blocks constitutes the grid, with the blocks being able to be grouped, similarly to threads, in one, two or three dimensions (though with much more generous limits on the maximum grid dimension [42]).

Another important aspect that must be taken into account is the memory hierarchy available in a CUDA-compatible device. The “main” GPU memory is split between the *Global Memory*, which holds arbitrary data and can be read and written from within each kernel, similarly to the heap-allocated memory in usual CPU programming, the *Texture Memory*, which (following from the needs of the texture mapping step in the 3D graphics pipeline) provides potentially faster access to arrays of floating point or integer

²In older architectures (Compute Capability lower than 7.0), threads within a warp are synchronized by design, while more recent devices allow for threads to be scheduled independently.

values that represent 1D, 2D or 3D grid points and interpolation along these dimensions, and the *Constant Memory*, which allows for the storage of a small amount of data (currently, up to the maximum of 64 KiB on all Compute Capabilities) that cannot be changed in the kernels, and whose access within each block may be cached more efficiently within the Streaming Multiprocessor. As for the latter, besides the existence of the caches for Global, Texture and Constant Memory that were previously mentioned, which reduce the memory access latency in comparison to data held in the “main” GPU memory, there is the possibility of allocating *Shared Memory*, which the different threads within the block being executed in the Streaming Multiprocessor can access typically with higher bandwidth and lower latency than global memory. Finally, one can also consider *local memory*, which is exclusive to each thread and (barring the memory sharing operations within the warp) cannot be accessed by other threads, corresponding to data used during the kernel execution that is not otherwise kept on registers, as determined during compilation.

One very significant consequence of the memory architecture, and, in particular, of the cache lines in the Streaming Multiprocessor and the alignment requirements of memory transfers, is that the most efficient way of storing and accessing an array of composite data types (`struct` in C/C++) is not in a contiguous array (the so-called *array of structures*), as would be common in CPU programming, but as a set of arrays of the underlying primitive data types (the so-called *structure of arrays*). This maximizes coalescing of loads and stores by the contiguous threads of a warp and minimizes unnecessary data transfers, since, by design, global memory accesses happen in 32, 64 or 128 byte chunks [42].

Figure 3.1 shows a schematic representation of the CUDA execution model and the typical device architecture, as has been described here.

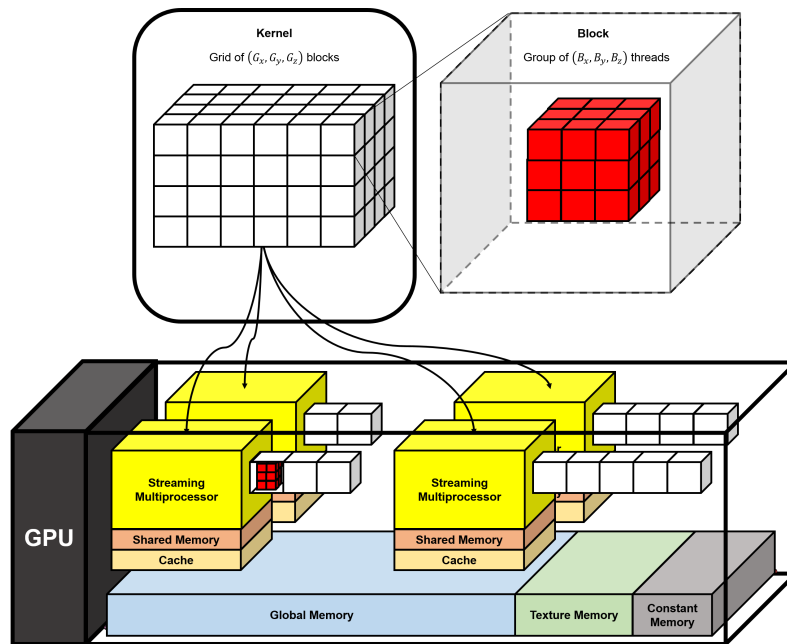


Figure 3.1: Schematic representation of the layout of the execution model and architecture of a CUDA-compatible GPU. Each kernel is comprised of multiple blocks of threads, each block is executed on a streaming multiprocessor. The streaming multiprocessors offer shared memory that can be accessed by the threads within a block as well as a local cache for global memory accesses, with the GPU also containing texture memory and constant memory that can be accessed by the streaming multiprocessors.

3.3 Topo-Automaton Clustering

Topo-Automaton Clustering [48, 51, 52] is an adapted version of the Topological Clustering algorithm described in section 2.4 that can be expected to be more suited to GPU implementation. The latter algorithm, as presented, has two potential disadvantages to GPU operation: the cluster growing outwards from seed cells is a not very parallel operation, and the maintaining of the separate lists of current cells, next cells and cells belonging to each cluster, in general, requires a memory access pattern that is sub-optimal for the GPU memory hierarchy.

If one considers the process of cluster growing as seen by a single cell, it is apparent that it only depends on that cell's direct neighbours. Of course, what happens to each neighbour cell during the cluster growing will depend on its neighbours as well, and similarly for every other cell of the calorimeter, but, in an iterative process where the clusters grow out from the seed cells until they reach the terminal cells, during each iteration, each cell only needs to check its direct neighbours. This provides a promising avenue for massive parallelization of the Topological Clustering algorithm on GPUs, especially if one evaluates each eligible pair of neighbouring cells in parallel. Given the way tag propagation is defined, these eligible pairs of neighbouring cells will correspond to those where at least one of them is a seed or growing cell and the other is not an invalid cell.

A second aspect to consider is an alternative way to identify which cells belong to a cluster, without relying on creating separate lists for each cluster. One possibility is assigning a unique tag for each cluster and having a global array that stores the tag of each cell. In this case, a cell is marked as belonging to a cluster by having its tag in the corresponding entry of the global tag array be the one that identifies that cluster. One must also consider another possible value for the tag that corresponds to cells that haven't yet been attributed to any cluster, which, for brevity, we will designate *unattributed tag*, t_u .

Taking into account both of these proposed changes, one can design an algorithm that, formally speaking, acts as a cellular automaton on the irregular grid created by the calorimeter cells and their neighbourhood relations, in which each cell has a state, that corresponds to its tag, and the transition between these states is determined by the states of each cell's neighbour (as well as the nature of the cells). Hence, the Topo-Automaton Clustering algorithm.

A final consideration must be made in regards to the stopping criterium of the algorithm. While, in Topological Clustering, the algorithm explicitly stops due to the fact that there are no further seed or growing cells to continue expanding the clusters, for Topo-Automaton Clustering, there is no explicit termination point. However, given the way tag propagation works, after a sufficient number of iterations, all the cells that could be included in a cluster will have been; in other words, an overall configuration will be reached in which further iterations do not change the tag of any cell, which should correspond (as long as the algorithm works properly) to the final result of cluster growing. Thus, the stopping criterium must be that all the cell tags remain unchanged after an iteration of tag propagation.

All in all, the Topo-Automaton Clustering algorithm can be described in the following way:

1. Create an array with as much entries as the number of cells in the calorimeter, the **cell tag array**, and initialize all of its entries to the unattributed tag. Let the tag of cell c be expressed as $\text{tag}(c)$.
2. For every cell of the calorimeters, indexed by c :
 - Calculate the signal-to-noise ratio of the deposited energy at the cell as $\text{SNR}(c) = \frac{|E(c)|}{N(c)}$.
 - Classify the cell:
 - If $\text{SNR}(c) > T_{\text{seed}}$, it is a **seed cell**.
 - If $T_{\text{seed}} \geq \text{SNR}(c) > T_{\text{grow}}$, it is a **growing cell**.
 - If $T_{\text{grow}} \geq \text{SNR}(c) > T_{\text{cell}}$, it is a **terminal cell**.
 - If $T_{\text{cell}} \geq \text{SNR}(c)$, it is an **invalid cell**.
3. For every **seed cell**, generate a unique tag to identify the protoccluster to which it belongs and assign the corresponding entry of the **cell tag array** to it. Let $P_c(t)$ identify the protoccluster that corresponds to tag t .
4. Repeat until there are no changes in the cell tags:
 - For every pair of neighbouring cells, g and n , such that $\text{SNR}(g) > T_{\text{grow}}$ (g is a **growing cell** or a **seed cell**) and $\text{SNR}(n) > T_{\text{cell}}$ (n is not an invalid cell):
 - If $\text{tag}(n) = t_u$ (n 's tag corresponds to the unattributed tag), assign to $\text{tag}(n)$ the value of $\text{tag}(g)$.
 - Else, if $\text{SNR}(n) > T_{\text{grow}}$ (n is also a **growing cell** or a **seed cell**):
 - * If $\text{tag}(g) = t_u$ (g 's tag corresponds to the unattributed tag), assign to $\text{tag}(g)$ the value of $\text{tag}(n)$
 - * Else, merge $P_c(\text{tag}(g))$ and $P_c(\text{tag}(n))$ (the protocclusters to which n and g belong, as identified by the respective tags)
 - Else (n is a **terminal cell**), if the seed cell that originated $P_c(\text{tag}(g))$ (the protoccluster to which g belongs) has a signal-to-noise ratio greater than the seed cell that originated $P_c(\text{tag}(n))$ (the protoccluster to which n belongs), assign to $\text{tag}(n)$ the value of $\text{tag}(g)$.
5. To obtain the final clusters in the same format as in the Topological Clustering and calculate their physical properties, collect all the cells whose entry in the **cell tag array** corresponds to each **protoccluster**.

Where cluster merging is defined as changing all the entries of the cell tag array whose tag corresponds to the cluster whose initial seed had the lowest signal-to-noise ratio to the tag of the other cluster.

The number of final clusters and the attribution of seed and growing cells to these clusters according to the Topo-Automaton Clustering will match those of Topological Clustering; however, by design, the

attribution of terminal cells will differ slightly: instead of the first cluster that reaches them, they will belong to the neighbouring cluster whose original seed cell has the highest signal-to-noise ratio. This design decision is motivated by the fact that, when evaluating multiple pairs of neighbours in parallel, the order in which the terminal cells' tags would be updated is essentially non-deterministic, which is undesirable for reproducibility and consistency of the results, and the alterations that are necessary for the terminal cell attribution to match the criteria of Topological Clustering without sacrificing performance in the intended GPGPU context are highly non-trivial. The impact of this difference on the physical validity of the results will be discussed in section 5.3, but, given the fact that terminal cells, by definition, have a lower signal-to-noise ratio, and, thus, tend to have lower deposited energies, they will contribute less significantly to the properties of the clusters; since the algorithm is intended particularly for the ATLAS trigger system, where absolute rigour of the results when compared to the offline algorithms is not as important as the performance, it was deemed an acceptable approximation.

Another approximation that must be mentioned is the possibility of considering the noise at each calorimeter cell constant throughout the run (or, at least, throughout a significant number of events). This helps in reducing the amount of data that must be read from the Athena data structures and transferred to the GPU when processing each event, which saves time. Once again, in section 5.3, we will study the impact of this approximation on the results, but, as section 5.4 will show, the data preparation and transfer will be a significant portion of the event processing time, which means that, once again, for the ATLAS trigger system, it can be expected to be an important approximation.

3.4 The First GPU-Accelerated Trigger Prototype

As presented in [48, 52], there has already been a prototype demonstration of the implementation of the Topo-Automaton Clustering on a GPGPU context, which found a speed-up by a factor between 1.3 and 2, depending on the nature of the events being processed. That prototype elected to implement a client-server structure, as represented in Figure 3.2, in which Athena was merely responsible for converting the relevant data into the structure-of-arrays form that is more amenable to GPU operations, sending it to the server and receiving the results, with the server managing all the GPU-related functionality, such as the necessary memory allocations and transfers and the kernel invocations.

As there were some indications that the client-server structure was detrimental to the efficiency of the code, with about 17% of the processing time being spent in inter-process communication, and, in any case, a significant part of the access to Athena data structures required changes to be compatible with current versions (in particular, due to the changes introduced to the data storage model by AthenaMT and its thread safety requirements), it was decided that the GPU accelerated version of the Topo-Automaton Clustering algorithm should be re-implemented in a more linear structure, with the code interfacing with the GPU within Athena. This is one of the purposes of the present work.

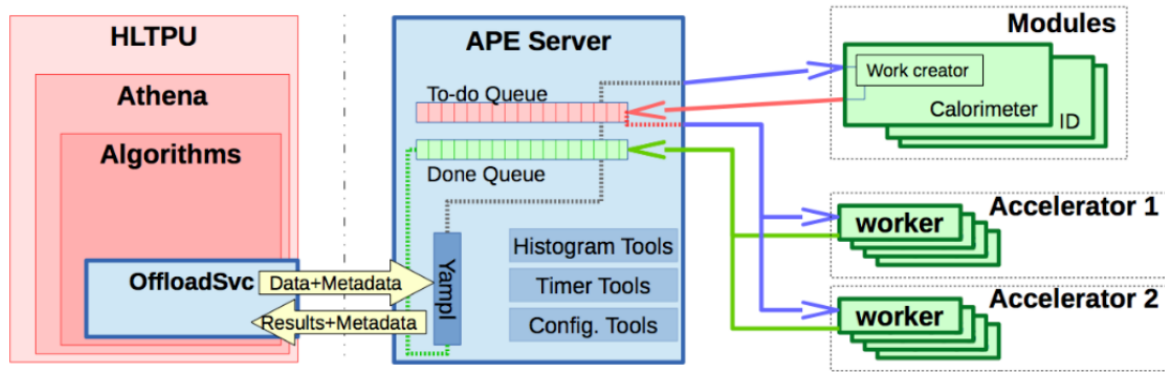


Figure 3.2: Schematic representation of the client-server structure of the previous GPU accelerated prototype, with the data outputs and inputs from and to Athena being managed by the Offload Service (*OffloadSvc*), which communicates with the **Accelerator Process Extension (APE) Server** that will manage the GPU kernel invocations and the CPU-GPU data transfers. Source: The ATLAS Collaboration, [52].

Chapter 4

Implementation and Optimization of the Topo-Automaton Clustering

4.1 Structure of the Implementation

As described in section 3.4, the GPU-accelerated prototype relied on a client-server structure to handle the interface between Athena and the GPU. One of the purposes of the present work was to rewrite the implementation of the algorithm with a more streamlined structure, where the GPU operations are explicitly queued and all the memory transfers are handled explicitly.

When processing an event, we can broadly define three stages:

1. **Preparing the Data:** converting from the Athena structures to the GPU-friendly representation and sending the data to the GPU
2. **Running the Algorithm:** performing the actual computations that will lead to the formation of the clusters
3. **Packaging the Results:** receiving the data from the GPU and converting from the GPU-friendly representation back to the Athena structures.

Since the Athena data structures are slated for a future redesign to improve support for hardware accelerators, the optimization efforts will be entirely focused on the parts of step 2. Any possible improvements to the data structures in order to make them more suitable for GPU operation, or to facilitate conversions to and from an alternative, GPU-friendly data representation (as the current implementation employs), are entirely outside the scope of the present work.

4.2 Algorithm Implementation Strategies

Upon some reflection, one can conclude that the execution of the Topo-Automaton Clustering algorithm described in section 3.3 can be separated in four mostly independent steps:

1. Calculate the signal-to-noise ratio of the energy deposited at each calorimeter cell.
2. Assign an appropriate tag to each protocluster created from the seed cells.
3. Grow the protocluster by propagating the tags throughout the eligible pairs of cells.
4. Calculate the final protocluster properties.

The implementation of the first step is straightforward from its intended effects. The implementation of the last step, on the other hand, depends on the way information regarding the protoclusters was kept during the previous steps. Thus, the key for achieving an optimized version of the algorithm lies in the freedom we will have in the implementation of steps 2 and 3, which in turn determine how step 4 must be achieved. We will now investigate the several options for these steps.

4.2.1 Tag Attribution Strategies

The previous implementation relied on explicitly creating an array of protoclusters in GPU memory, one for each cell identified as a seed, and sorting it, through the `sort_by_key` method provided by the Thrust Library [53]. Since the protoclusters were sorted by ascending order of signal-to-noise ratio of the corresponding seed cells, the desired merging behaviour (in which the cluster whose seed cell has the highest signal-to-noise ratio is kept) could be assured by making the tag an integer that corresponds to the index of the protocluster within the sorted array and taking the maximum of the neighbouring cells' tags during the growing step. This also has the advantage of facilitating the calculation of the properties of the final protoclusters, since each cell will contribute to the protocluster at the index given by its tag.

However, efficient GPU sorting algorithms are still somewhat the subject of ongoing investigation [54], and, in any case, the sorting of a large list will always take a non-negligible amount of time compared to the other operations that are expected to be performed. Therefore, it might be advantageous if the sorting step could be avoided while still preserving the ordering of the tags (which greatly simplifies the growing algorithm and prevents additional look-ups that would hurt performance in the GPU). We now present our attempts at achieving precisely that goal.

4.2.1.1 First Tagging Strategy

As we have seen, it would be desirable for the tags attributed to each protocluster be ordered according to the signal-to-noise ratio of the deposited energy in the corresponding seed cell (regardless of the use of an actual sorting step to ensure it). A second, obvious, requirement is that the tags must be unique for each protocluster (by definition, else the algorithm would not function properly). This second requirement can be taken into consideration by making the tag calculation also dependent on the index

of the seed cell. Thus, we must construct a function, tag , such that, given any two valid cell indices i_1 and i_2 and two seed cell signal-to-noise ratios of r_1 and r_2 (thus satisfying $r_1, r_2 > T_{\text{seed}}$),

$$\text{tag}(r_1, i_1) > \text{tag}(r_2, i_2) \Leftrightarrow r_1 > r_2 \quad \wedge \quad \text{tag}(r_1, i_1) \neq \text{tag}(r_1, i_2) \quad (4.1)$$

To satisfy this, we may introduce a monotonically increasing function over the reals greater than T_{seed} , $f(r)$, such that $f(r) \geq 1$ for all valid r , and an injective function over the integers from 0 to the total number of cells in the calorimeter (N_{cells}), $g(i)$, such that $g(i) \geq 0$ for all valid i , to write tag as:

$$\text{tag}(r, i) = f(r) \times \max_{\text{all valid indices } j} (g(j)) + g(i) \quad (4.2)$$

For brevity, let $M = \max_{\text{all valid indices } j} (g(j))$. We will now bring into consideration the fact that, in practice, the precision with which the algorithm will work is not infinite. In particular, a floating point format, as described by the IEEE-754 standard [55], must be used to represent the signal-to-noise ratios, which means that care must be taken to ensure that the inevitable rounding does not prevent any of the requirements from being satisfied. Let us, for the moment, assume f and g can be chosen to satisfy the constraints placed upon them (at least for a sufficiently broad range of possible inputs, namely in what regards the monotonicity of f). Considering a binary floating point format similar to those described in the standard, with b bits in the mantissa, we will have the following condition:

$$\text{tag}(r, i) \neq \text{tag}(r, j) \Leftrightarrow |g(j) - g(i)| > 2^{\lfloor \log_2(f(r) \times M) \rfloor - b} \Leftarrow r_{\text{max}} < f^{-1} \left(\frac{|g(j) - g(i)|}{M} 2^b \right) \quad (4.3)$$

Where f^{-1} is the inverse of f (which must exist since f is monotonically increasing) and r_{max} denotes the maximum signal-to-noise ratio to guarantee unicity of the tags despite the seed cells having the same signal-to-noise ratio.

For simplicity, computational efficiency and minimization of potential rounding issues, choosing f and g to be affine functions seems reasonable. Let $f(r) = \alpha r + \beta$ and $g(i) = ci + d$. The previous condition then becomes:

$$r_{\text{max}} < \frac{|j - i|}{\alpha N_{\text{cells}} + \alpha \frac{d}{c}} 2^b - \frac{\beta}{\alpha} \quad (4.4)$$

The definition of f implies $f(T_{\text{seed}}) \geq 1 \Leftrightarrow \alpha \geq 1/T_{\text{seed}}$. The definition of g implies $g(0) \geq 0 \Leftrightarrow d \geq 0$. The most natural choice to maximize r_{max} then becomes $\alpha = 1/T_{\text{seed}}$, $\beta = 0$, $c = 1$, $d = 0$. Thus, we have:

$$\text{tag}(r, i) = \frac{r}{T_{\text{seed}}} \times N_{\text{cells}} + i \quad (4.5)$$

For completeness' sake, we provide in Table 4.1 a brief list of the maximum signal-to-noise ratio for which tag unicity can assured, given seed cells with indices separated by a given Δi , for both 32-bit and 64-bit binary floating point formats, which have, respectively, $b = 23$ and $b = 53$, and the two most common choices for the seed threshold, $T_{\text{seed}} = 4$ (which is the case we are interested in) and $T_{\text{seed}} = 6$.

Based on a sampling of the events used for testing during the development process, less than 1% of

Table 4.1: Maximum signal-to-noise ratio, $r_{\max}$, for guaranteed tag unicity for seed cells with same signal-to-noise ratio and indices separated by Δi , for the two most common choices of the seed threshold, T_{seed} . b is the number of binary digits in the mantissa of the floating point representation, with $b = 23$ and $b = 53$ corresponding to 32-bit and 64-bit floating points, respectively.

Δi	$r_{\max}$			
	$T_{\text{seed}} = 4$		$T_{\text{seed}} = 6$	
	$b = 23$	$b = 53$	$b = 23$	$b = 53$
1	178.81	1.91×10^{11}	268.21	2.87×10^{11}
2	357.62	3.83×10^{11}	536.43	5.75×10^{11}
4	715.24	7.67×10^{11}	1072.87	11.51×10^{11}
8	1430.49	15.35×10^{11}	2145.74	23.03×10^{11}
16	2860.99	30.71×10^{11}	4291.48	46.07×10^{11}
32	5721.98	61.43×10^{11}	8582.97	92.15×10^{11}

the signal-to-noise ratios of the seed cells were greater than 100 and in no case did different cells have exactly the same ratio, which means that even with 32-bit floating point representations one can be reasonably confident that no tag collisions should happen in general. Given that 32-bit operations tend to be favoured in the design of GPUs, that would seem to be the most logical choice.

There is a final caveat of this tag attribution scheme. As we will see, the implementation of the algorithm requires the use of atomic maximum operations on global memory, which the CUDA API only provides for integer types, thereby preventing the direct use of the floating point values that result from expression 4.5 to represent the ordering of the clusters. Fortunately, the design of the IEEE-754 standard has the important property that the binary representation of two non-negative floating point numbers, when taken as the representation of an integer, will compare in the same way as the original numbers. Since, by design, all the tags that are generated will be greater than zero, we can simply interpret and store the resulting bit pattern as a 32-bit integer and use it throughout the remainder of the algorithm. Thus, our first tagging strategy can be written, in a C-like pseudo-code, as:

```
int32_t first_tag( float signal_to_noise,
                  int seed_cell_index )
{
    return float_as_int32( signal_to_noise / seed_threshold * NCells + seed_cell_index );
}
```

4.2.1.2 Second Tagging Strategy

Though a case could be made for the adoption of 64-bit tags in the strategy described above to absolutely ensure their unicity beyond any reasonable doubt, there is an alternative approach that can better take advantage of the additional range of possible values. While the previous strategy maintains the proper ordering of the protoclusters without requiring a sorting step, it has a very relevant shortcoming: there is no straightforward way to associate additional information to the tag other than the fact that the protocluster exists. In other words, the calculation of cluster properties will be dependent on creating and

maintaining some mapping from the protoccluster tag to a position in the array of protoccluster properties, which (regardless of the particular implementation chosen to do this) is also not particularly suited to GPU operation.

There is a simple solution to this problem: as in the previously used tagging scheme with the sorting step, we create an array of protocclusters in GPU memory while attributing the tags, but, rather than ordering the array, we simply store the index that corresponds to the protoccluster associated with the seed cell in the 32 less significant bits of the tag. The 32 most significant bits can either correspond to the bit pattern of the raw floating point representation of the signal-to-noise ratio, for simplicity, or, as could be done to facilitate comparisons between the the different strategies, the result of the first tagging strategy. This preserves the ordering, as the most significant bits will be compared first, and completely assures tag unicity (since the least significant 32 bits will be unique for each seed cell), while allowing fast access to the protoccluster properties to speed up the calculations. The only possible downside is the typically lower efficiency of 64-bit operations (atomic and otherwise) on GPUs, which is why the performance of both tagging strategies will be compared.

Once again, using the C-like pseudo-code,¹

```
uint64_t second_tag( float signal_to_noise ,
                    int seed_cell_index ,
                    int cluster_array_index )
{
    uint64_t tag = float_as_int32(signal_to_noise); //or first_tag(signal_to_noise , seed_cell_index);
    return ((tag << 32) | cluster_array_index);
}
```

And, to recover the array index, it is simply a matter of truncating the tag to the least significant bits (which, given the rules for unsigned type narrowing conversions in C/C++, can simply be written in those languages as a cast to a 32-bit unsigned integer).

4.2.1.3 Special Tag Values for Unassigned Cells

Both tag attribution strategies that were described refer only to the calculation of the tags that will identify each seed cell and, by extension, each initial protoccluster. As we have seen in section 3.3, the state of every cell must be initialized to a value that marks them as not being part of any cluster, thereby ensuring correct propagation of the tags, namely to growing and terminal cells. Since tag propagation happens by taking the maximum value of all neighbouring cells, this means that growing and terminal cells must have their state initialized to a value that is lower than the minimum possible protoccluster tag.

¹This implementation in particular is written to be completely portable (barring the `float_as_int32` function, which is provided by the CUDA Math API as an intrinsic, `__float_as_int`), standards-compliant and to invoke no undefined behaviour in both C and C++. Type punning is undefined behaviour in C++ standards later than C++11, but (upon some quick tests) it would appear the CUDA compiler treats that unspecified behaviour much in the same way as in the C11 standard and has the expected effects on the punned data; given that CUDA-compatible GPUs are specified to be little-endian [42], and given the 32-bit alignment requirements that should dispense with the need for padding (which is a second source of undefined and/or unspecified behaviour), this allows us to consider a non-portable, but possibly clearer, alternative implementation, in which the `uint64_t` is type punned with a `struct` consisting of two `int32_t`, with the first being the index of the cluster in the array and the second the signal-to-noise ratio (or the result of the first tagging strategy), thus setting the least and most significant bits, respectively, to the desired values.

Furthermore, since memory access is one of the main potential performance bottlenecks in GPU programming, it makes sense to give meaningful values for these placeholder states, such that the classification of the cell can be determined without incurring an additional memory access. Since, by construction, the bitwise representations of the protoccluster tags will have values greater than N_{cells} (much higher, in the case of the second tagging strategy, since the part that corresponds to the signal-to-noise ratio is in the 32 most significant bits), we may consider the following placeholder states:

- $\text{tag} = 0$, for invalid cells
- $\text{tag} = 1$, for terminal cells
- $\text{tag} = 2$, for growing cells

Thus, rather than just a single value for the unattributed tag mentioned in the description of the Topo-Automaton Clustering algorithm, we will have these three.

4.2.2 Cluster Growing Strategies

Cluster growing is unequivocally the most important part of the algorithm, and the one in which it can be expected to spend most of its execution time, hence the importance of considering and comparing the advantages of different implementation strategies. We will take into account essentially two sets of features where these strategies may vary: the ways in which the valid cell pairs for tag propagation are listed and the ways in which the merging of protocclusters is performed.

4.2.2.1 An Improvement for Tag Propagation

Before detailing the variations that were considered, we must describe an additional improvement on the direct implementation of the algorithm, which will be employed in all of them not only due to its performance advantages, but also due to the simplification it brings to the code. As has been previously described, the clusters can only grow through cells whose signal-to-noise ratio is greater than T_{grow} , which implies that, when propagating through all the possible cell pairs, some additional form of book-keeping besides the cluster tag must be kept to ensure terminal cells do not propagate their tag back to other cells. However, since the propagation of the tags to the terminal cells will not affect in any way the tags of the remaining cells, we can opt for a simpler alternative, in which we propagate first (iteratively) through all growing and seed cells until the clusters can no longer keep growing and only propagate to the terminal cells at the end (in a single step).

Since, as per 4.2.1.3, we can determine the type of each cell before propagation by its tag, it is easy to ensure that only cells with the desired classification have their tags updated without incurring additional memory accesses to read the signal-to-noise ratio of the cells, only their tag (which would need to be read in any case). Furthermore, as we will see, the way in which we enumerate the cell pairs may also enable us to implicitly ensure this correct propagation, dispensing with additional checks.

4.2.2.2 Valid Cell Pair List Creation Strategies

4.2.2.2.1 Selective Cell Pairs

The first approach, following directly the previous implementation, relies on creating, every time an event is processed, a list of the cell pairs in which at least one of them has a signal-to-noise ratio greater than T_{grow} and the other is not an invalid cell. These pairs can be described simply by the indices of the two cells in question, kept in a structure-of-arrays to be more amenable to GPU processing.

Since the choice of “first” and “second” elements of the pair is essentially arbitrary, we can simplify the algorithm for tag propagation by ensuring that the first element is always a growing or seed cell. As the list of pairs is created by checking the neighbours of all cells that have a signal-to-noise ratio greater than T_{grow} , this ensures that all the growing and seed cells will appear as the first element of at least one pair and that all the terminal cells to which a tag can be propagated will only appear as second elements of at least one pair.

Given that the propagation is done by taking the maximum over the tags of all neighbouring cells, and since, as described in section 4.2.1.3, terminal cells are initialized with a tag that is lower than that of seed and growing cells, the iterative part of tag propagation (between growing and seed cells) can be done by propagating the tag of the second element of each pair to the first. As for the final propagation step (to terminal cells), since it is only performed after all the possible propagations between growing and seed cells have been done, it can be described as a tag propagation from the first element of each pair to the second: if the second element is a growing or seed cell, they will already be part of the same cluster as long as all the neighbourhood relations between cells are bijective (which intuitively seems to be a more than reasonable assumption), while the terminal cells will belong to the neighbouring cluster whose seed cell has the highest signal-to-noise ratio, as intended.

4.2.2.2.2 Fixed Cell Pairs

Given the massive parallelism inherent to GPU processing, it could be possible to achieve a better usage of all available resources by always considering all possible pairs of cells, regardless of their classification, rather than only those that will indeed contribute to cluster growth.

In this case, the list of cell pairs is created by considering all the neighbours of all the cells, with no guarantees being made regarding the nature of the cells assigned to be the first and second elements of the pairs. As such, it only needs to be performed once, before processing any events, rather than for each event being processed.

Though the choice of this strategy means that explicit checks for the nature of the cells must be made during tag propagation, since there is no implicit property of the cell pairs upon which we can rely to simplify the implementation, it is worthwhile to measure if the time saved by not constructing the list of pairs for each event can compensate for the performance penalty incurred by these checks.

4.2.2.2.3 Implicit Cell Pairs

A third alternative (that requires a slight redesign of the tag propagation algorithm and a somewhat more sophisticated implementation) is to try to leverage the inherent synchronization and memory access coalescing of GPU warps when checking the neighbours of a given cell, which opens up interesting possibilities in terms of performing the maximum among all neighbours' tags using warp-based reductions instead of resorting to atomic operations to global memory across multiple threads. The main disadvantage of this is that some of the threads in the warp will not perform useful work, since the maximum number of neighbours that any cell can have (26) is smaller than the number of threads per warp (32).

In this strategy, each warp is uniquely associated with one calorimeter cell and its constituent threads will read the current tag of the neighbour that corresponds to its index within the warp (up to the number of neighbours of the cell associated with the warp). Then, a warp-based reduction is employed to calculate the maximum over all the neighbour tags held in thread-local memory, which, given the CUDA memory model, could result in better performance.

4.2.2.3 Protocluster Merging Strategies

4.2.2.3.1 Explicit Merging

The strategy that is conceptually closer to the description of the Topological Clustering algorithm is to include an explicit step in which the tag of the protoclusters that should be merged is properly updated. This requires us to maintain some form of mapping from all the protocluster tags that were created from all the existing seed cells, which we might call pre-merge tags, to the corresponding tags after a merge has happened, which we will naturally designate post-merge tags. Then, after each iteration of the tag propagation, every cell is updated with the post-merge tag that corresponds to its current (pre-merge) tag.

When adopting the second tagging strategy, since the tags contain within themselves the position of the protocluster within the array of all existing protoclusters, this mapping from pre-merge to post-merge tags may be achieved simply by creating an array that is initially filled with all existing tags and, when a merge occurs, writing the larger tag involved in the merge in the position that corresponds to the smaller tag. Then, each cell would check the position of the array that corresponds to its current tag and update it to the value found there.

However, the first tagging strategy does not allow us to easily derive a position within an array from the tag. This means that the mapping would require either a linear search of the array of all existing tags per each cell, which is far from efficient, or the use of a more complex data structure to maintain a map from each pre-merge tag to the corresponding post-merge tag. Although this can be achieved, commonly through hash maps (and [56, 57, 58] outline some possible GPU-friendly algorithms), there would still be some inevitable overhead while accessing these data structures, not to mention the additional time requirements for developing or finding an appropriate implementation and validating it. Thus, the inclu-

sion of an explicit merging step will not be considered for the versions of our implementation that adopt the first tagging strategy.

4.2.2.3.2 Implicit Merging Through Tag Propagation

Fortunately, the nature of the algorithm provides us with an alternative to the explicit merging step.

If we let the tags be propagated normally through the cellular automaton, whenever two protocusters fulfil the conditions for merging (a cell with signal-to-noise ratio greater than T_{grow} from one cluster borders another from the other), the higher tag will be propagated to the cells of the cluster with the lower tag. Since all of these will be connected (given that the lower tag was previously propagated to them), after a sufficient number of iterations, the final configuration will be exactly the same as if the cluster merging was performed explicitly: they will all have the higher tag.

Naturally, this comes at the cost of a potential increase in the number of iterations that are necessary to complete the tag propagation. For the second tagging strategy, for which we have a viable way of using an explicit merging step, it will be interesting to compare the performance impact of that increase when compared to the merging.

4.3 Data Structures

We now provide an overview of the data structures that are used in the implementation. Section A.2 of Appendix A contains more details on the corresponding C++ (and/or CUDA) code.

4.3.1 Object Wrappers

For more convenient syntax, and following the Resource Acquisition Is Initialization (RAII) principle that lies at the core of sensible memory management strategies in C++, we created a set of helper classes that encapsulate allocation, deallocation, copy and/or movement of objects that reside in CPU or GPU memory. These can be seen essentially as simplified and customized versions of `std::unique_ptr`, with constructors and assignment operators between them to perform the appropriate data transfers between the different memory spaces.

A possible, although less expressive, alternative would be the *unified memory* provided by CUDA since version 6.0 [42], which performs automatic data transfers between CPU and GPU memory without explicit calls to `cudaMemcpy`, though this would still require careful attention to allocation and deallocation, as well as introducing a possible source of errors in that explicit synchronization through `cudaDeviceSynchronize` is needed when accessing memory whose contents were changed in the GPU from the CPU. It must also be said that, in the not too distant future, as support for GPU operations and GPU resource management within the Athena framework increases, a more general and robust substitute for these helper classes is likely to become available. In any case, the relevant classes are:

- `Helpers::CPU_object<T>`: holds an object of type `T` in CPU memory.

- `Helpers::CUDA_object<T>`: holds an object of type `T` in GPU memory.
- `Helpers::CUDA_kernel_object<T>`: stores a non-owning pointer to an object of type `T` in GPU memory (in practice, equivalent in every way to a raw `T *` with the added benefit of only being constructible/assignable from a `CUDA_object<T>`, which means the object is guaranteed to reside in GPU memory).

Besides the aforementioned assignment operators and constructors, `CPU_object` and `CUDA_object` provide the `allocate()` and `deallocate()` functions so explicit memory management may be performed, as well as a `valid()` function to check if the held object has been allocated and can be safely accessed.

4.3.2 General Data Structures

When designing a GPU implementation of an algorithm, it is important to remember that a structure-of-arrays is a more suitable form of organizing data than the more natural array-of-structures, since in most cases it maximizes memory access throughput, as mentioned in section 3.2.

The second constraint that dictates the layout and composition of the data structures will be the expected lifetime of the underlying information and the way it will be accessed, especially when support for multi-threaded execution might need to be included in the future. In our particular case, we may distinguish between the information that will be initialized once and kept invariant during a run, which we will call *constant data*, and the one that will need to be read or updated for each event, the *event data*. Thus, in general, we will have a `ConstantDataHolder` and an `EventDataHolder` to manage the several components of each type of data through the object wrappers introduced before.

By definition, constant data should be safe to be accessed by different threads (or `CUDAevent` streams) simultaneously, since it does not change, while the event data must be stored and accessed separately for each event. Though, in the future, more close integration with the Athena memory management system might be desirable, the intersection of these systems with GPU memory management is not yet clear, so this must be taken care of locally by our implementation.

4.3.2.1 Constant Data

The first component of the constant data will be the information pertaining to the geometric configuration of the calorimeter, namely the position of the cells (in both Cartesian coordinates and η - ϕ) and their neighbours, which is stored as an array of cell indices for each cell (as the maximum number of direct neighbours for each cell is 26, as can be seen by simple geometrical considerations). This corresponds to a structure-of-arrays called `GeometryArr`.

As described in section 3.3, for the signal-to-noise ratio calculation, we will assume a constant noise for all events, which makes it also part of the constant data. Anticipating the possibility of that assumption needing to be changed sometime in the future, we will store the cell noise in a separate object, `CellNoiseArr`, so the code can more easily be changed to consider it part of the event data. This is simply an array of 32-bit floating point values, one per calorimeter cell.

4.3.2.2 Event Data

The data that underlies the whole analysis we intend to do is the energy that has been deposited on the calorimeter cells in each event. We store it in `CellEnergyArr` as an array of floating points, to allow for a simple transfer to the GPU. Then, the states of the calorimeter cells throughout tag propagation are held in the `CellStateArr` structure, which contains both the signal-to-noise ratio (which the algorithm, in the ideal case, seldom accesses after cell classification has taken place, but is needed at some points depending on the implementation strategy) and the tag array.

The final cluster properties will be stored in a `ClusterInfoArr`, which holds the cluster energy, transverse energy, pseudo-rapidity and azimuthal angle, together with the initial seed cell's index, signal-to-noise ratio and coordinates for easier access, in a structure-of-arrays. As the number of clusters will be variable from event to event, though the array is allocated once with a size that provides a very generous upper bound on the total number of clusters, we must also store the number of currently existing clusters.

For the strategies that include an explicit merging step, `ClusterMergeTable`, the array that implements the pre-merge to post-merge tag map, will also belong to the event data: it will correspond simply to an array of post-merge tags, since these strategies are only implemented together with the second tagging strategy, which provides a way for each tag to refer to the index of the cluster within the `ClusterInfoArr`. That same index can be used here so that each pre-merge tag refers univocally to a post-merge tag. This means that the size of this array can simply be the same as the maximum size of the arrays in `ClusterInfoArr`, with the tags themselves ensuring no out-of-bounds accesses occur.

4.3.2.3 Data Whose Nature Depends on the Chosen Strategy

Depending on the choice that was made regarding the creation of the cell pair list, the latter can either belong to the constant data (for fixed pairs) or to the event data (for selective pairs), or not need to exist at all (for implicit pairs). When it does exist, regardless of its nature, it will correspond to a `PairsArr` that holds two arrays of cell indices, together with the size of these arrays (which depends on the event, for selective pairs, and also on the geometry of the detector, which cannot be easily known at compile time). The maximum possible size of these arrays, which is also the size that is allocated when an object of type `PairsArr` is created, is given by $N_{\text{cells}} \times N_{\text{max neighbours}}$, where $N_{\text{max neighbours}} = 26$ is the maximum number of direct neighbours that each cell may have.

4.3.3 The `TrigTopoAutomatonCluster` Class

Our algorithm is meant to be executed within the Athena framework. Its coupling to that framework happens through the `TrigTopoAutomatonCluster` class, whose implementation details largely result from the needs of interfacing with Athena and thus are of little relevance for the present discussion.

Following from what was mentioned in section 2.3.4.1, `TrigTopoAutomatonCluster` must be derived from both `AthAlgTool` and `CaloClusterCollectionProcessor` and implement an `execute` method,

which will perform the desired event processing.

By design, the class will keep a member `ConstantDataHolder` object to hold the data throughout the several events.

4.4 Implementation

We will now present the actual implementation of the algorithm by describing in some detail what happens when an event is processed. As there are some subtle design decisions with impact on performance that can be more directly expressed through the source code rather than a textual explanation, interested readers may refer to section A.3 of Appendix A for a more detailed picture of what happens.

4.4.1 Outline

As mentioned in section 4.1, there are three stages in event processing. For implementation purposes, the middle one (which corresponds to the algorithm itself) may be further split into four parts:

1. **Preparing the Data:** converting from the Athena structures to the GPU-friendly data representation
2. Running the Algorithm
 - (a) **Signal-to-Noise Calculation:** Calculate the signal-to-noise ratio, classify the cells and attribute an appropriate tag to those that are seed cells.
 - (b) **Pair Creation:** If using selective pairs, build the list of all pairs that allow cluster growth (if not, there is nothing to be done at this step).
 - (c) **Cluster Growing:** Propagate the tags.
 - (d) **Final Cluster Properties Calculation:** Calculate the cluster properties.
3. **Packaging the Results:** converting from the GPU-friendly representation back to the Athena structures.

It must be pointed out once more that the optimization focuses entirely and exclusively on the “Running the Algorithm” part of event execution.

4.4.2 Preparing the Data

Data preparation has two main components: preparing the constant data and preparing the event data.

By definition, constant data does not change throughout all event processing, so its preparation is only necessary when processing an event for the first time. When doing it, anticipating the future need to run our algorithm in a multi-threaded scenario, a `std::mutex` is acquired to prevent data races from other threads and it is only released after the data is successfully sent to the GPU.

On the other hand, a new `EventDataHolder` is created each time the event is processed (and destroyed at the end of processing). Though, in the current single-threaded execution model, the unnecessary allocations and deallocations have a detrimental impact on performance, that can be considered part of the time taken by the data conversions, in the sense that it does not directly concern the implementation of the algorithm itself, which is the real object of the present work. In the future, some solution involving more sophisticated memory management might be developed to alleviate this.

4.4.2.1 Constant Data Preparation

Given the previous definition of constant data, there are at least two structures that must be filled: the description of the geometry and the cell noise array. In case the fixed pair creation strategy is being used, the list of all possible cell pairs must also be constructed at this step.

The geometry description and noise array are filled by iterating through all the cells, using the `Athena` classes described in section 2.3.4.3, and accessing their coordinates, neighbours and noise information.

The data is then sent to the GPU, which is ensured by the copy assignment operators of the previously described object wrappers.

The construction of all cell pairs, if necessary, is done already in GPU memory, and simply corresponds to a kernel where each thread will add all the possible pairings of a calorimeter cell to its neighbours, as enumerated in the `GeometryArr`, to the list of pairs in the `PairsArr`.

4.4.2.2 Event Data Preparation

Similarly to what was done to prepare the constant data, we fill the array of the cell energies by iterating through every cell and getting its energy, and send it to the GPU through the object wrappers.

However, there is an additional operation that must be performed to ensure correct behaviour of the algorithm: we must zero out the number of existing clusters and, if using the selective pairs strategy, the number of cell pairs, since, in general, allocation does not initialize memory (this could possibly be done together with the allocation itself in the constructor of the `EventDataHolder`, but, conceptually, it belongs in the data preparation step).

4.4.3 Signal-to-Noise Calculation

The signal-to-noise calculation step obviously involves calculating the ratio itself, but (since `CUDA` kernel calls have some overhead) it makes sense to group the cell classification and tag attribution here as well.

The step is implemented by simply calling the relevant kernel, with at least as many threads as there are calorimeter cells (thus the grid size will be determined by the block size, which will be one of the parameters of the optimization study that will be performed, for this and all other steps). Each thread will correspond to a single cell, accessing the respective entries of the `CellEnergyArr` and the `CellNoiseArr` to calculate the ratio, write it to the `CellStateArr` and then, depending on the classification of the cell,

either set the appropriate special tag value, or, if it is a seed cell, calculate the unique tag that will correspond to the protocluster.

For the second tagging strategy, this also requires updating the `ClusterInfoArr`, in particular incrementing the cluster count, which should be done with an atomic operation to ensure thread-safety. In this case, it makes sense to initialize the cluster properties to 0 (or to the relevant seed cell values) in this kernel as well.

4.4.4 Pair Creation

This step is only necessary when the selective pairs strategy is adopted. It will once again correspond to the invocation of a single CUDA kernel, with as many threads as there are cells, and each thread will concern itself with a given calorimeter cell and its possible pairings with its neighbours.

Given what was outlined in section 4.2.2.2.1 regarding the selective pair creation strategy, if the cell to which the thread corresponds has a signal-to-noise ratio less than or equal to T_{grow} (that is, if it is an invalid or a terminal cell), the thread stops execution as it will not contribute to tag propagation. Otherwise, its neighbours, as described in the `GeometryArr`, will be checked, with those that constitute valid candidates for being added to a cluster (signal-to-noise ratio greater than T_{cell}) being added to an array held in thread-local memory. Once all the neighbours have been checked, the number of pairs in the `PairsArr` will be atomically increased by the size of that array of valid neighbours, and the pairs will be added, with the cell that corresponds to the thread (and which we ensured was either a growing or a seed cell) being placed always as the first element of the pair.

Though this would mean that potentially redundant pairs are being added when two growing or seed cells neighbour each other, the improvement for tag propagation described before actually requires the presence of all growing or seed cells as the first element of a pair. Given that memory is not a constraint, since the maximum size of the pair array is allocated at once, and there is no additional duplication of work given the meaning of the first and second element of the pair, these apparently redundant pairs do not have the detrimental impact on performance that might seem at first glance.

It must be pointed out that the implementation for this step is the same for both tagging strategies and, since no part of it depends on the tags of the cells, the performance should be exactly the same.

4.4.5 Cluster Growing

As mentioned in section 4.2.2.1, the best approach to cluster growing involves performing a series of iterations in which the tags are propagated among the cells that have a signal-to-noise ratio greater than T_{grow} and, when the stopping criterium of no cell changing tag during an iteration is satisfied, a final propagation step to the terminal cells. This (together with the nature of the propagation algorithm in itself) requires some form of global synchronization between iterations, which, given the constraints of the CUDA programming model, does not have an obvious, easy solution:

- **Single Kernel:** The simplest approach is to perform every computation in a single kernel and rely

on intra-block synchronization. While straightforward and supported by any CUDA-enabled hardware, this has the significant disadvantage of limiting the maximum number of concurrent threads to the maximum block size, which is 1024, as mentioned in section 3.2, significantly reducing the extent to which we can take advantage of the capabilities of the GPU.

- **Cooperative Groups:** CUDA 9.0 introduced the concept of cooperative groups, which provide a mechanism for explicit inter-block synchronization. However, for such a synchronization to be possible, not only the number of thread blocks needs to be carefully controlled so all of them can be executed at the same time in the available streaming multiprocessors, but some rather stringent requirements on the compute capability of the hardware and on the CUDA platform that is being used must also be fulfilled [42]. This significantly hampers portability.
- **Multiple Kernel Launches:** The alternative for inter-block synchronization is launching multiple kernels, which, by design, provide an implicit point of synchronization. However, each kernel launch carries with it some overhead, and, furthermore, since, at each step, the stopping criterium must be checked and some flag must be (re)set accordingly, there will need to be data transfers between the GPU and the CPU at each step, which is far from desirable. Nevertheless, this does not impose any constraint on the kernel launch parameters or on the GPU, ensuring maximum portability.
- **Dynamic Parallelism:** CUDA 5.0 introduced the concept of dynamic parallelism, which allows kernels to be launched from within kernels in the GPU. This requires hardware with compute capability greater at least 3.5, a much laxer requirement than that of cooperative groups, and does not directly impose any constraints on the kernel launch parameters. By leveraging this possibility, we can avoid communicating with the CPU during the execution of the algorithm, since the flag for the stopping criterium will be checked directly in GPU memory, and, if necessary, new kernels will be launched.

Given the ultimate goal of futurely integrating this algorithm in the ATLAS trigger system, and given that the author was not aware of the specifications of the hardware that will be available in that environment, it seemed reasonable to exclude, for the time being, the use of cooperative groups. On the other hand, since devices with compute capability 3.5 came out in 2012 and current versions of the CUDA SDK have already started dropping support of earlier models, it also seemed reasonable to consider that dynamic parallelism will be possible with the available hardware, and thus this was the chosen solution. We must keep in mind, however, that here lies room for future improvement, not only through the possible use of cooperative groups for inter-block synchronization, but also through a more profound redesign of the algorithm to reduce the dependency on that same synchronization. In any case, both possibilities can be considered beyond the scope of the present work.

The current implementation of the cluster growing step will then consist of a synchronization kernel that is launched with a single thread, which will keep launching the kernel that corresponds to the iterative part of tag propagation (and, if adopting a strategy with explicit merging, the merge step) until the stopping

criterion is fulfilled (which is checked through a flag that signals if there have been tag changes in the propagation step). After that stopping criterion is reached, the synchronization kernel will call the final part of the growing process, which will propagate the tags to the terminal cells.

We will now provide some additional detail on the implementation of the iterative and final parts of the growing process, for the different choices of pair creation and merging strategies.

4.4.5.1 Strategies With Selective or Fixed Pairs

The implementations that are simplest to describe are those that include an explicit list of cell pairs, be it selective or fixed. In these cases, the iterative and final parts correspond to kernels that are launched with at least as many threads as the number of valid pairs stored in the `PairsArr`, with each thread being responsible for one of these pairs.

Following from section 4.2.2.1, the iterative part propagates tags to growing or seed cells, and, given the way the tags (and especially the special tag values) have been defined, the classification of the neighbouring cell from which that tag comes is not relevant, as a valid protocluster tag will never be replaced with a special tag. For the case of selective pairs, this allows a nearly branchless kernel, as the first elements of each pair are all growing or seed cells, thus we simply propagate from the second; with fixed pairs, we must check explicitly that the first element of the pair has a signal-to-noise ratio greater than T_{grow} (which we can do by checking that its tag is at least the special tag value attributed to growing cells) and only propagate in that case. Since different pairs may refer to the same cells, all tag updates are done through atomic maximum operations on the tag array contained in the `CellStateArr`. To check for the stopping criterium, a global flag is set (also atomically) whenever the first element of the pair will change tags, and, in case the strategy involves a merge step, the tag in the appropriate entry of the `ClusterMergeTable` is set (also via atomic maximum) to the expected post-merge tag.

As for the final part, given that propagation is now intended to happen to terminal cells, we simply propagate from the first elements of the `PairsArr` to the second, except in the cases where, with fixed pairs, the first element is a terminal cell as well (to prevent clusters from growing from terminal cell to another, which the algorithm forbids).

4.4.5.2 Strategies With Implicit Pairs

The implicit pair creation strategy, as described in section 4.2.2.2.3, requires a more sophisticated implementation, since the core idea is reducing the amount of atomic operations performed on global memory by doing the maximum over the neighbouring cells in thread-local memory. This means that some form of reduction of the maximum operation over the warp must be employed, so it is important to consider the different strategies for achieving this:

- **Fully Intrinsic Warp-Based Reduction:** for devices with a CUDA Compute Capability of at least 8.0, there is an intrinsic function, `__reduce_max_sync`, that performs this reduction in what one could expect to be the faster and most efficient way. However, this places almost as strict a requirement

on the possible devices as the cooperative groups solution for kernel synchronization, which means that this strategy will not be considered, for the same reasons as the latter.

- **Warp-Based Reduction With Intrinsic Shuffles:** CUDA 9.0 introduced another intrinsic function, `__shfl_down_sync`, which, for devices of Compute Capability of at least 3.0, allows exchanges of values held in the thread's local memory with other threads in the warp (with lower indices). This can be used to write a reduction by sending the memory from the threads with higher indices down in decreasing strides, until the first thread of the warp has operated on all the values held in local memory [59]. This performs the reduction in $\mathcal{O}(\log_2(N))$ operations, where N is the number of threads that hold meaningful values.
- **Explicit Warp-Based Reduction:** following generally the same algorithm as the previous reduction strategy, but with the values being accessed through shared memory. The additional loads and stores to shared memory should result in a decrease in performance, but it is still a valid strategy, especially for compatibility (though the use of dynamic parallelism requires a device with Compute Capability of at least 3.5, which means that, if our implementation is able to be run, the intrinsic shuffles will be supported as well).

Regardless of the choice of the reduction strategy, the implementation is based on a kernel that is launched with as many warps as there are cells in the calorimeter (so, with $32 \times N_{\text{cells}}$ threads in total), with each warp being responsible for one calorimeter cell.

Within the warp, each thread (with an index within the warp lower than the number of neighbours of the cell) will load the tag of one of the cell's neighbours, as described in the `GeometryArr`, and then the warp-based reduction is performed to get the maximum among the neighbours' tags (together with the cell's current tag), which is then written atomically to the corresponding entry in the tag array of the `CellStateArr`.

For the iterative part of tag propagation, this will only be performed for growing or seed cells, following the usual reasoning, while the final propagation step only occurs for terminal cells and neighbours which are not growing or seed cells are ignored, also as usual, to prevent clusters growing between terminal cells.

4.4.6 Final Cluster Properties Calculation

Once the clusters have finished growing, the only part of the algorithm that remains to be done is calculating their properties, namely their energy, transverse energy and average pseudo-rapidity (η) and azimuthal angle (ϕ). This is done by considering the energy contributions from each cell of the cluster, as well as their η and ϕ coordinates weighted by the cell's absolute energy.

For the second tagging strategy, this is quite straightforward, as the cluster tags were constructed specifically so one could know to which cluster the cell would contribute: one simply launches a kernel with as many threads as there are calorimeter cells, where each thread will check if the corresponding cell belongs to a cluster and, if it does, add the contributions to the appropriate elements of the arrays in the

ClusterInfoArr.

From what was previously discussed in section 4.2.1 and section 4.2.2.3, the first tagging strategy, quite on the contrary, does not provide any direct way to associate a tag with a position in the list of all existing clusters. This implies that, for every cell that belongs to a cluster, we must search through that list to find the right cluster to which it will contribute its properties. Since this needs to happen only once per seed cell, it seems justifiable to not incur the overhead (not only in terms of the runtime, but especially in terms of development time) of creating a more complex data structure, simply searching linearly through the cluster array instead.

In an effort to minimize the performance impact of the linear search through the cluster array, contrary to what happens in the second tagging strategy, only the final clusters are included in that array. To that end, before summing the contributions from the different cells, a cluster finding kernel is run, in which each thread will check if a cell is the seed cell that originated the cluster to which it belongs by comparing its tag to what would be the tag as computed from its index and signal-to-noise ratio. If they match, a new cluster is created (through an atomic increment of the cluster count in ClusterInfoArr) and initialized, with its properties being set to 0 and the necessary seed cell information being stored.

In any case, once all cell contributions have been summed, the cluster coordinates are divided by the sum of the absolute energies of the cluster cells to get the average, and the transverse energy is calculated from that same sum and η . For the second tagging strategy, the clusters that have been merged with others throughout the cluster growing stage of the algorithm, to whom no cells contribute and, thus, whose sum of the absolute energies of the cells is zero, must be signalled so they can be excluded from the list of final clusters. For simplicity, this will correspond to setting the seed cell signal-to-noise ratio to T_{cell} , which is obviously invalid.

All of this is done in the final kernel, in which each cluster will be processed by a separate thread.

4.4.7 Packaging the Results

After the final cluster properties have been calculated, it is time to re-integrate that information into Athena. We must first copy the ClusterInfoArr and the array that holds each cell's tag back to CPU-accessible memory and build an array of CaloClusterCellLink such that each element corresponds to a valid cluster (that is, if using the second tagging strategy, we must check that the clusters have a seed cell has a signal-to-noise ratio of the seed cell greater than T_{seed}). We then add to the appropriate element of that array of CaloClusterCellLink the cells that belong to the respective cluster, as identified by the cells' tags. Finally, we populate the xAOD::CaloClusterContainer with the several xAOD::CaloCluster that have the previously filled CaloClusterCellLink and the properties that were calculated in the previous step. This concludes the processing of an event.

Chapter 5

Topo-Automaton Clustering Optimization, Validation and Performance

5.1 Experimental Conditions

5.1.1 Run System

All the results were obtained on the remote server provided by INCD, on a machine running CentOS 7.9.2009 equipped with an AMD EPYC 7552 CPU and a Tesla V100S GPU, using CUDA 10.2, GCC 8.3 and version 22.0.24 of Athena.

For the particular case of the optimization measurements, the code was compiled with most optimizations turned on, namely through the `-O2` compiler flag, to yield a better approximation of their expectable performance under actual use.

5.1.2 Samples

The results were obtained from two different samples, both resulting from simulations done with the Pythia [60] Monte Carlo event generator. Those samples are (at the moment of writing) readily available in the CernVM File System (`cvmfs`) as part of the validation samples for the nightly builds of Athena.

- $t\bar{t}$ Events¹: A proton-proton collision that has resulted in the formation of a top quark-antiquark pair. In particular, the events that form this sample correspond to cases in which the final state contains at least one lepton (*non fully hadronic*), and have a pile-up of 80 collisions per bunch crossing. There is a total of 3000 events in this sample.

¹`mc15_13TeV.410000.PowhegPythiaEvtGen.P2012.ttbar_hdamp172p5_nonallhad.recon.RD0.e3698.s2608.s2183.r7195`

- Jet Events²: A proton-proton collision that has resulted in the formation of a quark-antiquark or a gluon-antigluon pair that originate sprays of particles travelling in the same direction (which form the eponymous *jets*). In particular, the events that form this sample correspond to cases in which there are two jets (di-jet events), and have a pile-up of 20 collisions per bunch crossing. There is a total of 500 events in this sample.

These two samples were chosen because the type of events contained in them correspond to what one could expect to be two important limits to be studied: $t\bar{t}$ events are expected to give rise to a high number of clusters throughout the calorimeter, which means that the computational cost of processing them tends to be higher, increasing the impact of any performance gains that may be achieved in the GPU, while jets tend to produce less wide and less widespread clusters, which means that the computational cost of processing them on the CPU tends to be lower, which may allow the data conversion and transfer overheads associated with GPU acceleration to offset the speed-up of the actual processing. The fact that the jet sample has a lower pile-up will tend to further decrease the number of clusters per event, which makes it even more of a limiting case.

It must be said that none of the samples are representative of the conditions expected to be found in the High-Luminosity LHC (as described in section 2.2.2, the expected pile-up is 200 collisions per bunch crossing), but we can expect that the performance comparisons between the GPU and the CPU algorithms in the present conditions will give a lower bound to the improvement under those conditions. In other words, in events with higher pile-up, the algorithm can be expected to perform at least as well as what will be shown here (and likely better) when compared to the currently used CPU implementation, hardware differences notwithstanding.

5.2 Optimization Studies

5.2.1 Optimization Procedure

To be as thorough as possible in the comparisons between the implementation alternatives described in the previous chapter, we will take advantage of the entirety of the available events from both samples to run the relevant parts of the event processing. For convenience and speed, this is achieved through a stand-alone program which uses the event information that is saved for validation purposes (as described in next section), thus enabling us to skip the rest of the event processing that takes place within Athena. Furthermore, in an attempt to improve the statistical properties of the time measurements (without incurring too long a processing time), we will measure the time taken to process each event twice. We will be considering the two event types separately since it is not inconceivable that an implementation proves itself advantageous in only one kind of event, given some possible subtle dependencies on the properties of the distribution of cells along the calorimeter.

As hinted at during the description of the implementations, there is a significant degree of freedom in the choice of the thread block size for each kernel invocation, that one does not have any reasons to stipulate

²`valid3.147917.Pythia8_AU2CT10_jetjet_JZ7W.recon.RD0.e3099_s2578_r6596_tid05293007_00`

a priori. In fact, it may even happen that different implementations become more or less advantageous in respect to each other depending on the choice of block size. As such, to determine the most optimal choice, one must also consider all the possible block sizes, which, as of current hardware, correspond to all integer multiples of 32 not greater than 1024 [42], giving us a total of 32 possible sizes. This is compounded by the fact that some processing steps correspond to two kernels³, which means that, in some cases, we will need to consider 1024 possible combinations of block sizes, besides the alternative implementations themselves.

As a final note, we must point out that any results from this optimization procedure are largely dependent on the properties of the hardware in which these tests will be performed, which means that the conclusions cannot be readily generalized to all possible situations. Nevertheless, if a very significant difference between alternative implementations is found, one could reasonably expect the performance advantage to remain as long as the changes in hardware are not too extreme.

5.2.2 Optimization Results

Tables 5.1, 5.2, 5.3 and 5.4 outline the choices of block size(s) that result in the two best average execution times for each considered implementation. The average, t_{avg} , and standard deviation, σ_t , of the execution times have been calculated based on the two repetitions of all the events of the corresponding samples. For the tables where the implementations depend on multiple kernels, as the second block size generally has a much smaller ($\lesssim 1 \mu s$) influence on the execution time, the second best choice corresponds to the first choice that has a different first block size (as the 31 other possible choices for the second block size typically result in an execution time that, within the standard deviation, is indistinguishable from the best possible choice).

Table 5.1: Optimal parameters for signal-to-noise ratio calculation and tag initialization, for both event types and both tagging strategies. The implementations are described in section 4.4.3. The bold row corresponds to the strategy that was selected for the final implementation. The average, t_{avg} , and standard deviation, σ_t , of the execution times have been calculated based on the two repetitions of all the events of the corresponding samples.

Tagging Strategy	$t\bar{t}$ Events			Jet Events		
	Block Size	t_{avg} (μs)	σ_t (μs)	Block Size	t_{avg} (μs)	σ_t (μs)
1st	64	55	6	64	55	3
	736	55	6	704	55	7
2nd	64	58	10	736	58	3
	704	59	9	64	58	11

There are two things that become quite clear from Table 5.1: the execution time of the signal-to-noise calculation step is largely independent of the type of the event (as expected, since it operates over all cells) and the first tagging strategy has a slight advantage (also as expected, since the second

³In fact, there are some cases where three kernels are executed, but, in those cases, two of the kernels are sufficiently similar that the same block size can be considered for both without affecting the optimization efforts in any meaningful way (in the most relevant case, cluster growing, where the kernels in question are the iterative part of tag propagation between grow and seed cells and the final tag propagation to terminal cells, this is compounded by the fact that one of the kernels will be run several times while the other is only run once).

tagging strategy creates the array of clusters at this step), though by all means not very significant given that the execution times of the two strategies are entirely within one standard deviation of each other. Another interesting detail is that the second tagging strategy tends to have slightly larger variations in the execution times, which is likely due to the atomic operations necessary to update the size of the protocluster array in a thread-safe way whenever a seed cell is identified. It is also worth mentioning (though the table does not show it, as it only contains the two best cases) that the average execution time does not change appreciably with the choice of block size, with the difference between the best and worst case being less than $3\text{ }\mu\text{s}$, thus within one standard deviation for all cases.

Table 5.2: Optimal parameters for cell pair creation, for both event types and both tagging strategies. The implementations are described in section 4.4.4. The bold row corresponds to the strategy that was selected for the final implementation. The average, t_{avg} , and standard deviation, σ_t , of the execution times have been calculated based on the two repetitions of all the events of the corresponding samples.

Tagging Strategy	$t\bar{t}$ Events			Jet Events		
	Block Size	t_{avg} (μs)	σ_t (μs)	Block Size	t_{avg} (μs)	σ_t (μs)
1st	32	85	8	64	69	7
	64	87	7	32	69	6
2nd	32	85	9	64	69	8
	64	87	7	96	70	7

Table 5.2 shows us that there is no appreciable difference between the implementations, which serves mostly to reassure us that this optimization procedure is working properly, given that the underlying kernel is exactly the same for both tagging strategies and the data access pattern (through structures-of-arrays, as detailed in section 4.3) means that the changes in the tag size will not affect its behaviour in any way. As could be expected, execution takes longer for the denser $t\bar{t}$ events since the amount of work to be performed depends on the number of cells whose signal-to-noise ratio is above T_{grow} . It is important to recall that this step is only performed with the selective cell pairs strategy; in the fixed pairs strategy, the list is created before event processing begins, while the implicit pairs strategy accesses the neighbouring cells directly during the cluster growing step. As such, when considering the overall event processing time for the different alternatives, the time taken for pair creation must only be accounted for when the selective cell pairs strategy is employed.

The main conclusion that should be drawn from Table 5.3 is that the addition of an explicit merging step reduces the execution time significantly for the denser events, which means that solutions with the second tagging strategy should be favoured. Furthermore, the variability of the execution time between events (and, somewhat less expressively, between event types) is reduced. All of this can be attributed to the fact that, with an implicit merging step, the total number of iterations needed for tag propagation is proportional to the distance (in the sense of number of connected neighbours) between the seed cell and the farthest cell of the largest final cluster, while, with an explicit merging step, it will be proportional only to the largest distance between a seed cell and a terminal cell or a growing cell of another cluster. In the most extreme conceivable case of two clusters that merge only through a cell at their outer radius, this would represent a factor of 2 on the execution time.

Table 5.3: Optimal parameters for cluster growing, for both event types and the strategy combinations that were chosen to be tested. The implementations are described in section 4.4.5. The bold row corresponds to the strategy that was selected for the final implementation. The average, t_{avg} , and standard deviation, σ_t , of the execution times have been calculated based on the two repetitions of all the events of the corresponding samples.

Strategy			$t\bar{t}$ Events				Jet Events			
Tag	Pair Creation	Additional Information	Block Size		t_{avg} (μs)	σ_t (μs)	Block Size		t_{avg} (μs)	σ_t (μs)
			1	2			1	2		
1st	Selective	Implicit Merging	32	64	853	352	1024	512	437	305
			64	256	986	441	32	512	525	152
2nd	Selective	Implicit Merging	32	128	839	334	1024	608	439	302
			64	128	987	441	32	160	524	150
2nd	Fixed	Implicit Merging	704	512	1237	523	704	160	760	270
			736	512	1240	522	736	352	766	271
2nd	Selective	Explicit Merging	32	192	667	136	32	736	462	138
			64	448	751	162	64	128	495	161
2nd	Implicit	Explicit Warp-Based Reduction	192	512	1117	317	192	384	700	209
			160	320	1300	357	320	448	716	230
2nd	Implicit	Warp-Based Reduction With Intrinsic Shuffles	256	1024	928	219	288	704	615	196
			288	416	933	225	320	672	615	196

The factor of ~ 1.25 that can be found for the $t\bar{t}$ sample seems reasonable given that most clusters would tend to have more intersections than just one cell. However, the performance for the jet events seems not to improve with the inclusion of the merge step (in fact, for the best choice of block sizes, the execution time increases by about $30\mu\text{s}$ when compared with the equivalent implementation with an implicit merge step); the most likely explanation lies in the fact that the clusters will tend to be smaller and fewer in these kinds of events, which nullifies the advantage of the merge step (which, not being instantaneous, means there will be some time spent on doing an essentially unnecessary operation). Nevertheless, as the events upon which we expect the algorithm to be run will have an even higher pile-up than the $t\bar{t}$ sample and the execution time for the jet sample increases by less than 6%, it seems reasonable to assume the inclusion of an explicit merge step will still prove advantageous, even if a significant fraction of real events will correspond to a collision where jets are formed.

The strategies that try to forego the creation of a specific list of cell pairs per event do not seem to be advantageous under any circumstances, especially considering that the cell pair creation step takes less than $100\mu\text{s}$. This is likely due to the fact that more threads do not perform useful work, either because the cell pair they are considering does not contribute to tag propagation (for fixed cell pairs) or because they refer to a non-existent neighbour of the cell the warp is considering (for implicit cell pairs). In the latter case, it is conceivable that the use of the fully intrinsic warp-based reduction mentioned in section 4.4.5.2 might improve the execution time, though it seems unreasonable to assume that improvement to be sufficient to overcome the $250\mu\text{s}$ difference in execution time between the best implicit cell pairs method (with intrinsic shuffles) and the best overall strategy (selective cell pairs with explicit merging).

It must be said that Table 5.3 does not show all the possible combinations of implementation strategies, only those that were considered the most relevant. The execution times of the other combinations could be estimated by assuming that each separate choice contributes a constant factor to the execution time; for instance, the average execution time for a fixed pairs strategy with explicit merging can be estimated by multiplying the execution time in the third row by the ratio between those of the fourth and the second rows.

Table 5.4: Optimal parameters for the final cluster properties calculation, for both event types and both tagging strategies. The implementations are described in section 4.4.6. The bold row corresponds to the strategy that was selected for the final implementation. The average, t_{avg} , and standard deviation, σ_t , of the execution times have been calculated based on the two repetitions of all the events of the corresponding samples.

Tagging Strategy	$t\bar{t}$ Events				Jet Events			
	Block Size		t_{avg} (μs)	σ_t (μs)	Block Size		t_{avg} (μs)	σ_t (μs)
	1	2			1	2		
1st	128	704	254	24	96	544	146	42
	96	64	255	22	128	704	146	43
2nd	960	128	125	22	736	384	112	5
	1024	64	125	23	64	96	112	5

Finally, Table 5.4 confirms the suspicion that motivated the development of the second tagging strategy: if we are able to know the address of the cluster directly through its tag, the reconstruction of the cluster properties can be done much faster. This reasoning is valid for both event types, though it becomes more relevant the more dense the events are, which is particularly beneficial given the expected increase in event density when dealing with real data.

From all of what was said before, we can arrive at a final implementation that should correspond to the following strategies:

- Second tagging strategy
- Selective cell pairs
- Explicit merge step during cluster growing

5.3 Physical Validation of the Results

5.3.1 Validation Procedure

To ensure that our implementation is able to adequately reconstruct the properties of the clusters and maintain the physical validity of the results, a comparison needs to be made with the values that come from the CPU implementation.

It is important to point out that the different implementation strategies all lead to exactly the same results, with the due exception of small differences in the final reconstructed cluster properties that arise from the intrinsic lack of associativity of floating point additions (in this case due to the non-deterministic order

in which the contributions from the different cells will be summed up, depending on the scheduling of the execution of the thread blocks within the GPU). This means that we only need to check one of the possible implementations, in this case, the final implementation, as defined in the previous section, since this is the one we are interested in having implemented within Athena.

As mentioned before, we will need to save some information from each event to perform this comparison. We will store the energy deposited at each cell, the list of final clusters with their respective properties (E , E_T , η and ϕ) and the final state of the automaton, that is, the array that expresses to which cluster each cell belongs (which corresponds to `CellStateArr` in our implementation). We must also offload (when the first event is processed) the geometry, which is taken to be constant throughout all events, as is, for our algorithm, the noise associated with the energy measurements.

This is achieved through the addition of a step in the relevant trigger chain, after the cluster growing algorithm is executed, where the relevant information is accessed and stored in a binary format, which is then read and processed by an external script that produces the plots that we will show throughout this section.

One thing that should be pointed out is that we will be comparing the results of our algorithm, which assumes a constant noise, with ones produced by the CPU algorithm that may consider the noise to change during the run. This means that, *a priori*, we can expect some additional differences other than in the terminal cells (that the algorithms, by construction, assign according to different criteria). While using the same values for the noise in both algorithms would provide us with a clearer notion of how close the GPU implementation reproduces the results of the CPU, keeping the CPU implementation with a varying noise allows us to also evaluate the soundness of the constant noise approximation, at least for the samples being used here.

5.3.2 Cluster Matching

One of the most significant requirements for the comparison between the results of our GPU implementation and those of the CPU implementation to be possible is that the clusters identified in each of them are properly matched, that is, that one is properly able to find the most similar clusters among the list of final clusters produced in each event by the implementations. Since the results will be greatly influenced by this matching, special care must be taken to ensure it is done as rigorously and thoroughly as possible. It could be expected that the seed cells of the clusters would be the same between the CPU and GPU implementations, providing a simple and easy way to match the clusters; however, at a certain point of the development process, significant differences between the implementations were being found, leading to a significant fraction of mismatched clusters, which motivated the adoption of a more complex matching algorithm. Though those differences were later discovered to be due to several otherwise undetected implementation and configuration issues, as the more complex matching algorithm was already implemented, it made sense to continue its use.

The matching is done through what can be seen as a variant of the Gale-Shapley Algorithm [61], which

is a solution to the Stable Matching problem (also known as Stable Marriage Problem): there are n elements of a type (in our case, clusters identified in our GPU implementation) to be matched to n elements of another type (clusters in the CPU implementation), and it is possible to know the order by which each element of a type ranks the elements of the other in terms of its “preference” to be matched to them. The goal, then, is to construct a list of pairs of elements of each type, such that, for all of those elements, it is impossible to find a possible pairing for which the preference between the elements would be greater. In our case, we must also take into account the possibility that the number of GPU clusters and CPU clusters might not be the same and that some clusters might be tied in terms of preference, which makes our problem an example of Stable Matching with Unequal Numbers and Indifference.

Unequal numbers do not constitute a problem to the formal definition of the Gale-Shapley algorithm, since one can introduce additional, “dummy” elements to the less numerous set, with their preferences being set lower than all remaining elements, and then consider the elements that were matched with them to be unmatched at the end. As for indifference, there are different approaches that might be taken, since, depending on the requirements one wishes to place upon the properties of the final solutions, the existence of a solution is not guaranteed [62], but, for our purposes, it is sufficient to consider the condition used in the classic Stable Matching problem, namely that no possible pairing has greater preference than those returned as the solution, which corresponds to *weak stability*. This has the significant advantage that the algorithm is always guaranteed to arrive at a solution.

Our actual implementation must also take into account that we are not interested in simply calculating the best possible match between the clusters, instead, we should only consider the pairings for which there is sufficient similarity to reasonably justify that the clusters actually refer to the same physical phenomenon. This minimum similarity criterium, $S_{\min}$, was set after some consideration to the value of $S_{\min} = 0.9$, in an attempt to find a balance between false negative matchings (which could happen if $S_{\min}$ was too high) and false positives (for too low values of $S_{\min}$).

Let us now describe how the similarity calculation was defined. Let C_1 and C_2 be two clusters, let all the cells be indexed by an integer i , with the signal-to-noise ratio of cell i being r_i , and let w_t , w_g and w_s be three weights that must be specified for each kind of cell (in our case, $(w_t; w_g; w_s) = (1; 250; 5000)$ seems like a reasonable choice). The raw similarity between the clusters, $\mathcal{S}_{\mathcal{R}}(C_1, C_2)$, will be given by:

$$\mathcal{S}_{\mathcal{R}}(C_1, C_2) = \sum_i \xi(C_1, i) \xi(C_2, i) \omega(r_i) r_i \quad (5.1)$$

Where:

$$\xi(C, i) = \begin{cases} 1, & \text{cell } i \in C \\ 0, & \text{otherwise} \end{cases} \quad \omega(r) = \begin{cases} w_s, & r > T_{\text{seed}} \\ w_g, & T_{\text{seed}} \geq r > T_{\text{grow}} \\ w_t, & T_{\text{grow}} \geq r > T_{\text{cell}} \\ 0, & \text{otherwise} \end{cases} \quad (5.2)$$

From this, we can define the similarity between two clusters, C_1 and C_2 , relative to one of them, $S(C_1, C_2; C_1)$, as:

$$S(C_1, C_2; C_1) = \frac{\mathcal{S}_{\mathcal{R}}(C_1, C_2)}{\mathcal{S}_{\mathcal{R}}(C_1, C_1)} \quad (5.3)$$

Our cluster matching process, then, will correspond to the following steps:

1. Calculate the raw similarity between the CPU and GPU clusters.
2. For each GPU cluster G , sort the list of CPU clusters, C_j , by their similarity to the GPU cluster, $S(G, C_j; G)$. Exclude from that list those with similarities lower than $S_{\min}$. This will be the list of possible CPU cluster matches for each GPU cluster.
3. Create a list clusters under consideration and fill it with the GPU clusters. Repeat while there is at least an unmatched cluster under consideration with a non-empty list of possible matches:
 - For each GPU cluster under consideration, G ,
 - If the list of possible CPU cluster matches is empty, remove G from the list of clusters under consideration.
 - Else, check the status of the first element in the CPU cluster matches list, C_0 :
 - If C_0 has been matched with a cluster, G' (which may be the same as G):
 - If $S(C_0, G'; C_0) \geq S(C_0, G; C_0)$, remove C_0 from the list of possible matches of G .
 - Else, remove C_0 from the list of possible matches of G' and match it with G .
 - Else, match C_0 with G .

This is guaranteed to arrive at a solution in $\mathcal{O}(N_{\text{clusters max}}^2)$, where $N_{\text{clusters max}}$ is the maximum between the number of clusters identified by both algorithms for the event in question. In practice, for all the events that were considered, the matching was mostly completed in the first iteration, with three iterations being the worst case, which is consistent with the previous observation that this matching algorithm is more complex than what is strictly necessary, given the cluster similarities that one can *a priori* expect.

5.3.3 Validation of the Cluster Identification

The first step of the validation of the results is making sure that a majority of the clusters is being correctly identified by our GPU algorithm. In particular, we will be interested in verifying that the number of cells that are attributed to different (that is, non-matching) clusters is minimal, with the differences being mostly contained in the terminal cells, since we know the algorithms differ explicitly in the way terminal cell attribution is done.

Figure 5.1 shows the differences that can be found between the number of clusters identified by our GPU implementation and those identified by the existing CPU algorithm in each event. For reference, the total number of clusters identified per event is shown in Figure 5.2.

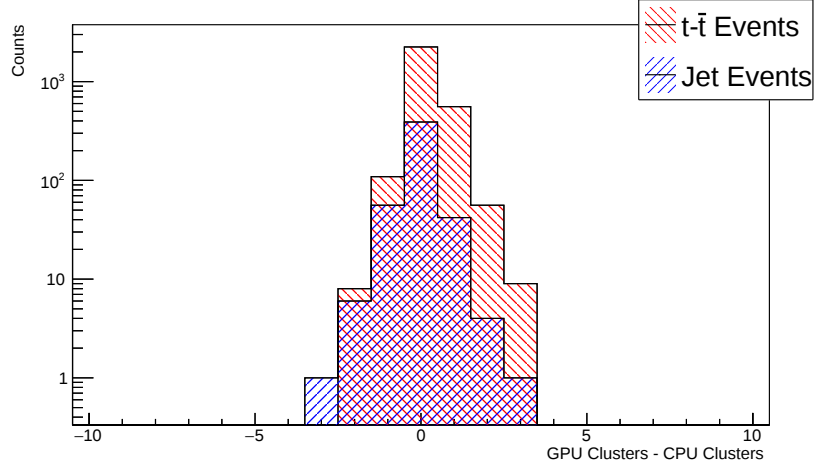


Figure 5.1: Distribution of the difference between the number of clusters identified by the GPU implementation and by the CPU implementation in each event, for $t\bar{t}$ and di-jet events.

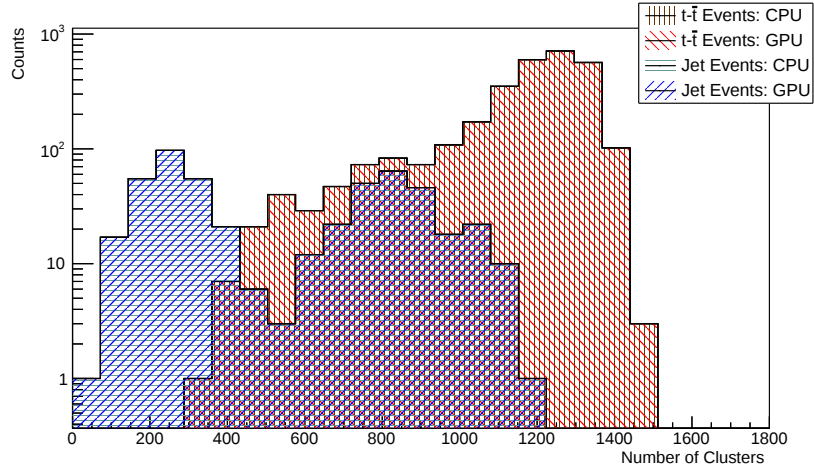


Figure 5.2: Distribution of the total number of clusters identified in each event, for $t\bar{t}$ and di-jet events and for the GPU and CPU implementations.

Though the number of identified clusters could be expected not to be the same in all events given the possibly different noise definitions, it would seem reasonable to posit that the distribution would be symmetrical in relation to the y axis, which is not the case: there tend to be more clusters identified by the GPU in $t\bar{t}$ events while there are less clusters in jet events.

This is somewhat troubling and warrants additional investigation, at the very least to exclude the possibility that this bias is an artefact of the choice of samples, but also, for instance, to check for some potential dependency on the regions of the calorimeter where those clusters are (not) being found. Even so, it must be said that the differences in the number of identified clusters are not very significant when compared to the the total number of clusters.

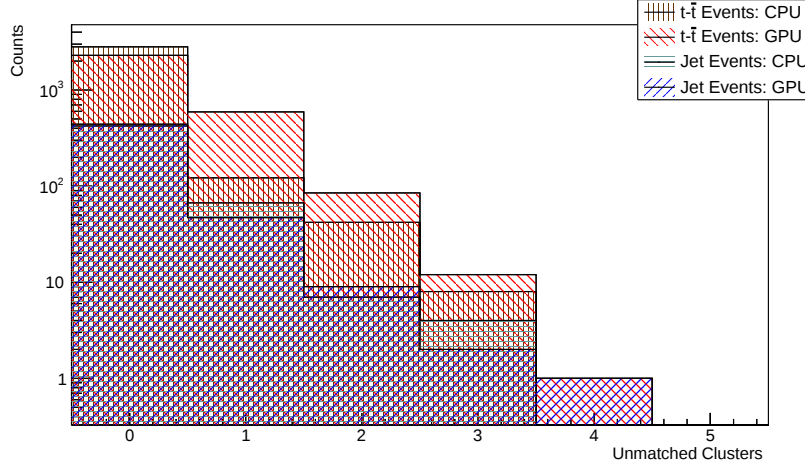


Figure 5.3: Distribution of the number of clusters that have not been matched in each event, for $t\bar{t}$ and di-jet events and for the GPU and CPU implementations.

In any case, those differences in the number of identified clusters imply that there will be several events in which a perfect matching between the CPU and the GPU clusters is logically impossible (by direct application of the Pigeonhole Principle). In fact, as Figure 5.3 illustrates, there is a small, but non-negligible number of events where one or two clusters from one implementation have not been matched with clusters from the other. In section 5.3.6, we will study the properties of these unmatched clusters to ensure that their existence does not pose a meaningful risk of loss of physical significance of the results, in particular due to them being consistent with noise clusters. For now, we must remark that, following from the asymmetry in Figure 5.1, for $t\bar{t}$ events, there are more unmatched clusters from the GPU implementation, while, for jet events, there tend to be slightly more unmatched clusters from the CPU implementation.

Let us now analyse the number of cells that have been attributed to a different cluster by each implementation. Figure 5.4 shows the total number of cells per cluster that, for a given implementation, were not attributed to the cluster that matches with the one under consideration, according to the matching process described in the previous sub-section (or, for short, *differently attributed cells*). Figure 5.5 shows only the growing or seed cells among them.

If we inspect Figure 5.6 and Figure 5.7, which represent the fraction of clusters that, for a given number of differently attributed cells, have more than that number of differently attributed cells (in essence, if $f(n)$ is the number of clusters that have n differently attributed cells, we are representing $g(n) = \left(\int_n^{+\infty} f(t)dt \right) / \left(\int_0^{+\infty} f(t)dt \right)$), it is noticeable that, while about 99% of the clusters differ by less than 25 cells in total, more than 99.9% of the clusters have exactly the same seed and growing cells and more than 99.99% differ by less than 2 seed or growing cells, which is in accordance with the expectation that the differences lie mostly in the attribution of terminal cells. In general, the results seem to support the conclusion that, despite not being perfect, the reconstruction of the clusters in the GPU implementation is close to that of the CPU implementation even with the potential differences in the noise.

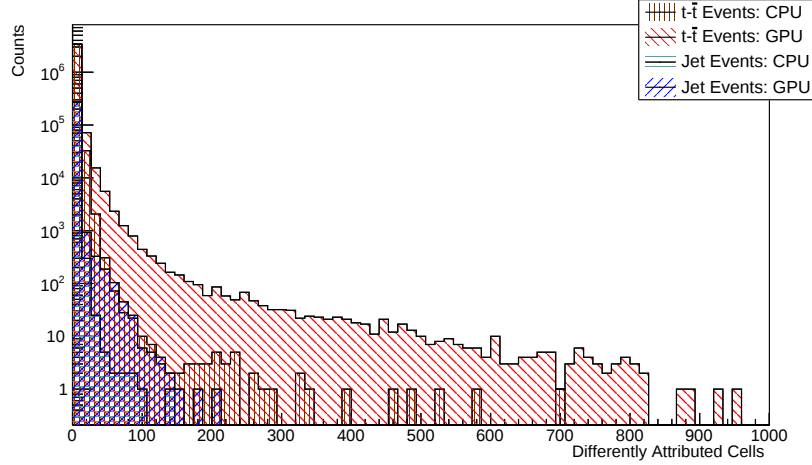


Figure 5.4: Distribution of the number of cells per cluster that have been attributed to different clusters (according to the cluster matching algorithm) by the GPU and CPU implementations, for $t\bar{t}$ and di-jet events and both implementations. This number is obtained by checking every cell of every cluster identified in an event by a given implementation and counting those that, in the other implementation, have not been attributed to the cluster that matches with the cluster under consideration, which also includes cells that have not been attributed to any cluster at all in the other implementation.

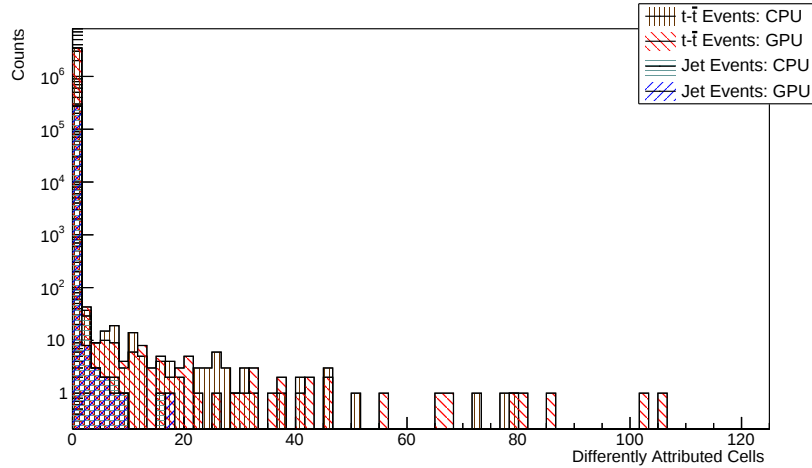


Figure 5.5: Distribution of the number of growing or seed cells per cluster that have been attributed to different clusters (according to the cluster matching algorithm) by the GPU and CPU implementations, for $t\bar{t}$ and di-jet events and both implementations. Obtained similarly to Figure 5.4, with the only difference being that only the growing or seed cells are considered.

It should be pointed out that, as can be seen most expressively in Figure 5.6, the number of terminal cells in clusters that have been identified by the GPU that do not belong to the matching cluster in the CPU is significantly higher than the converse. As that number also includes cells that belong to a cluster in an implementation and do not belong to any cluster in the other, this suggests that the GPU implementation may be including more cells in the clusters, in general. In any case, it not inconceivable that this issue is related with the causes of the aforementioned bias towards identifying more clusters in the GPU in the $t\bar{t}$ samples.

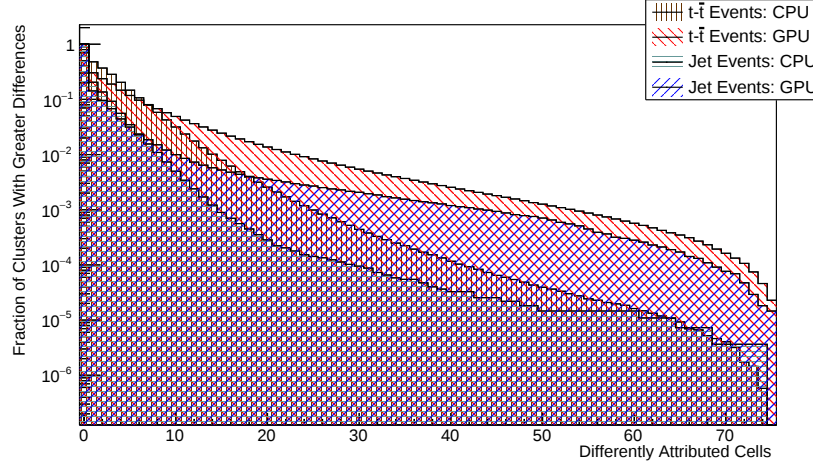


Figure 5.6: Fraction of clusters that, for a given number of differently attributed cells, have more than that number of cells attributed to different clusters (as determined in Figure 5.4), for $t\bar{t}$ and jet events and the GPU and CPU implementations.

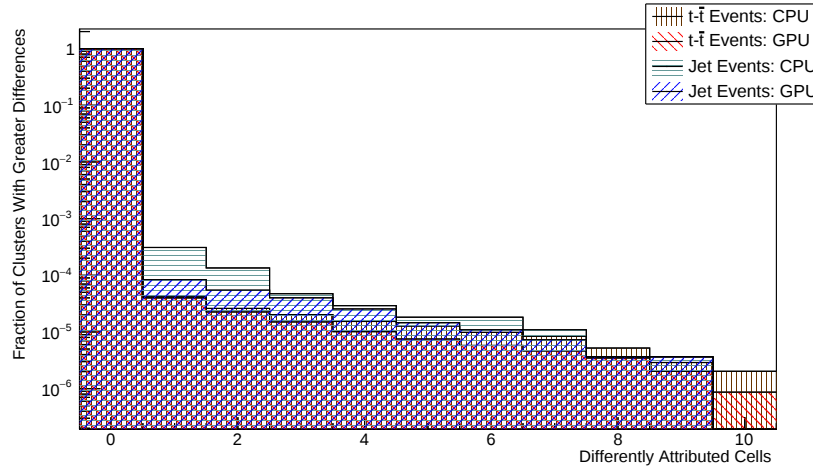


Figure 5.7: Fraction of clusters that, for a given number of differently attributed cells, have more than that number of growing or seed cells attributed to different clusters (as determined in Figure 5.5), for $t\bar{t}$ and jet events and the GPU and CPU implementations.

5.3.4 Validation of the Physical Properties Reconstruction

From the previous validation step, we concluded that the clusters we are identifying on the GPU correspond reasonably well to those from the reference implementation on the CPU. We must now make sure that the differences between the implementations do not result in significant changes in the reconstructed physical properties of the clusters.

Let us begin by observing the energy and transverse energy distributions of all of the clusters identified in all events. As shown in Figure 5.8 and Figure 5.9, the CPU and GPU algorithms seem to be mostly consistent.

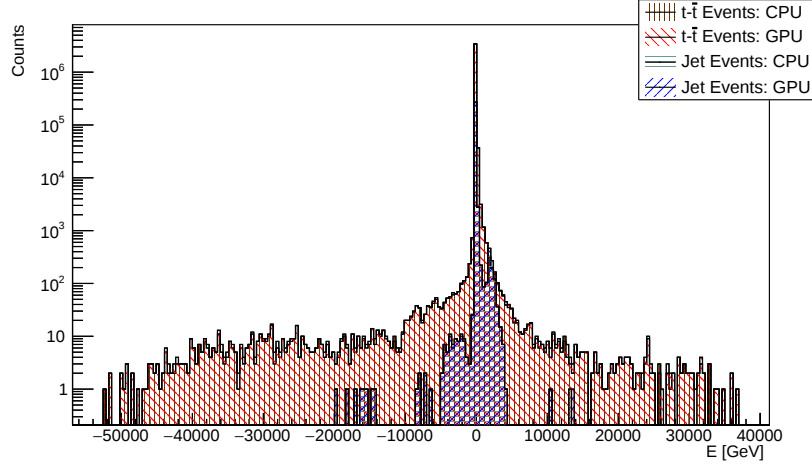


Figure 5.8: Distribution of the energies of all of the clusters in $t\bar{t}$ and di-jet events, as reconstructed by the GPU and CPU implementations.

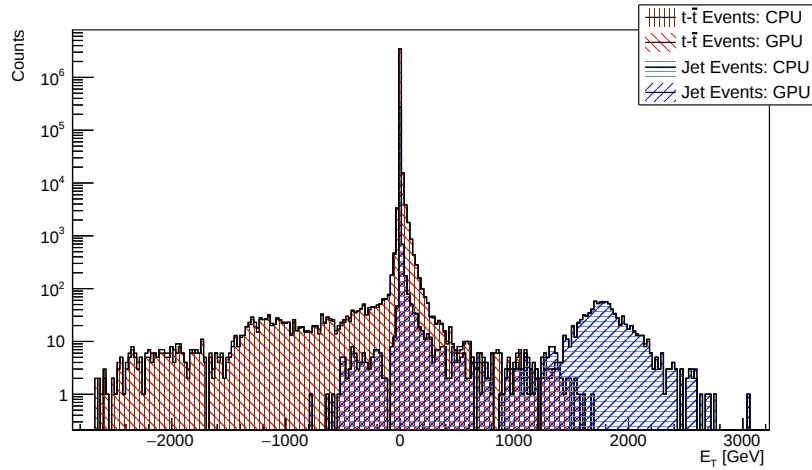


Figure 5.9: Distribution of the transverse energies of all of the clusters in $t\bar{t}$ and di-jet events, as reconstructed by the GPU and CPU implementations.

However, given the broad range of cluster energies that the events contain, it is far more illustrative to plot the variations between the energies of matching clusters. As we can see in Figure 5.10 and Figure 5.11, those energy differences are not too significant when compared to the overall energy distributions, though they are still far from being zero.

Let us now take a look at the coordinates that have been attributed to the clusters. The distributions of the reconstructed values for η and ϕ are sufficiently similar to prevent any meaningful comparison, so we must once again analyse the differences between matched clusters.

The pseudo-rapidity is reconstructed quite acceptably, since, as shown in Figure 5.12, the variations fall within the $[-0.2; 0.2]$ interval.

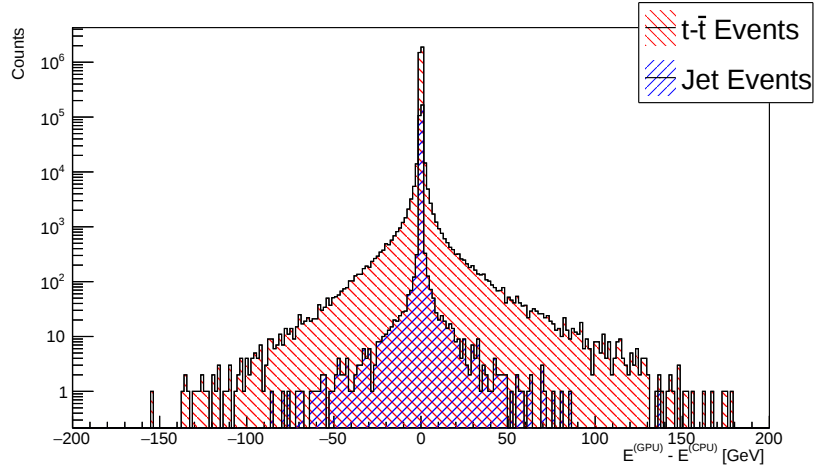


Figure 5.10: Distribution of the difference between the energy as reconstructed in the GPU and as reconstructed in the CPU, for all matched clusters in $t\bar{t}$ and di-jet events.

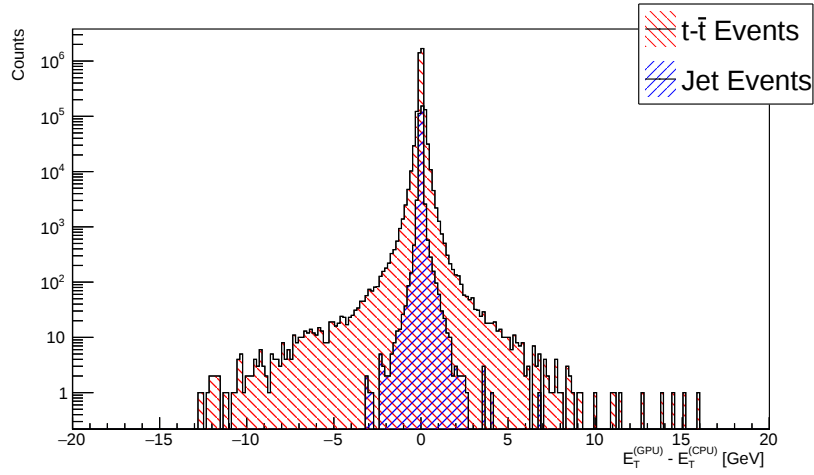


Figure 5.11: Distribution of the difference between the transverse energy as reconstructed in the GPU and as reconstructed in the CPU, for all matched clusters in $t\bar{t}$ and di-jet events.

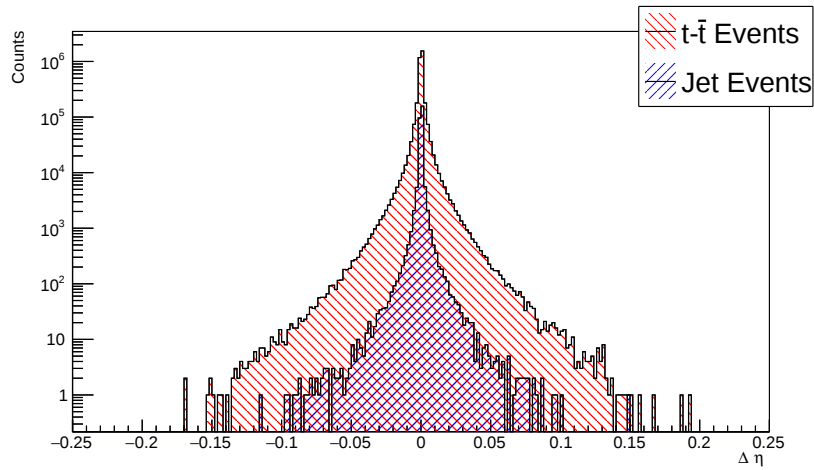


Figure 5.12: Distribution of the difference between the pseudo-rapidity (η) as reconstructed in the GPU and as reconstructed in the CPU, for all matched clusters in $t\bar{t}$ and di-jet events.

However, the azimuthal angle is more problematic. As Figure 5.13 shows, although the vast majority of the matched cluster pairs are reconstructed with similar values of ϕ , all possible values for the difference between the angles can be found, as there are counts throughout the entire $[-\pi; \pi]$ interval. Though the calculation of this property (and of the difference between GPU and CPU clusters) may be somewhat more sensitive to the rounding associated with floating point arithmetic due to the inherent wrap-around of angular quantities, that does not seem to be a sufficient explanation for this discrepancy and further investigation is necessary to better understand it.

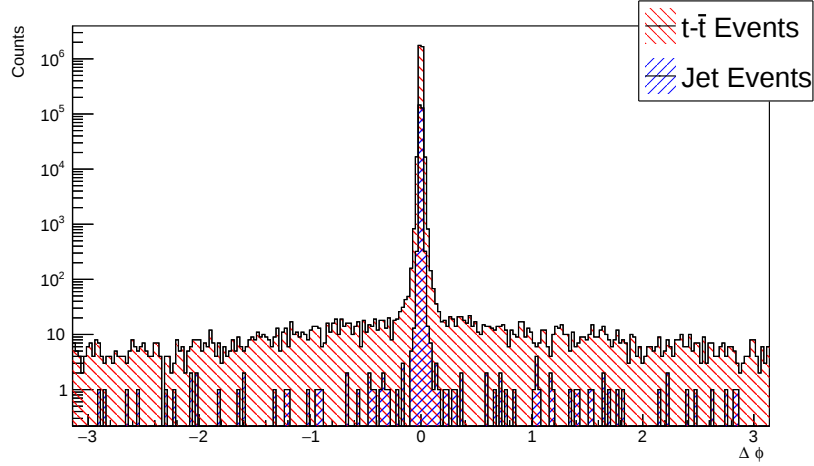


Figure 5.13: Distribution of the difference between the azimuthal angle (ϕ) as reconstructed in the GPU and as reconstructed in the CPU, for all matched clusters in $t\bar{t}$ and di-jet events.

Naturally, given the definition of distance between clusters as $\Delta R = \sqrt{(\Delta\eta)^2 + (\Delta\phi)^2}$, it is unsurprising that the distance can be as high as π , as we can see in Figure 5.14, while the majority of the cluster pairs have ΔR less than 0.2, as Figure 5.15 shows quite clearly.

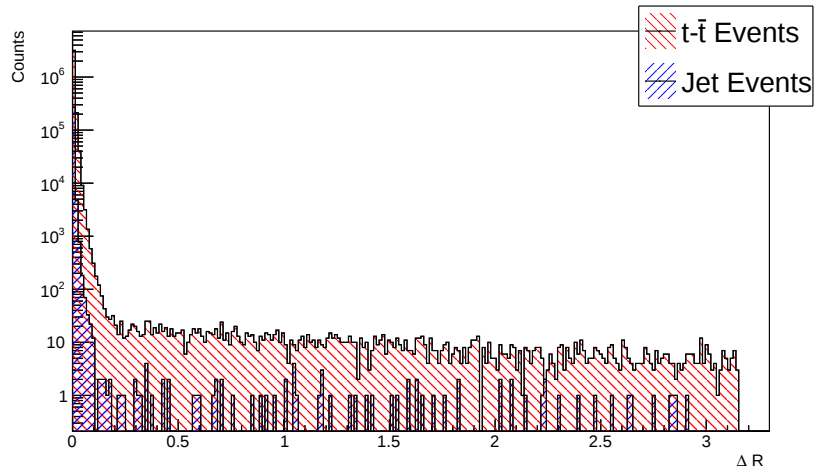


Figure 5.14: Distribution of the distance between pairs of matched clusters, defined as $\Delta R = \sqrt{(\Delta\eta)^2 + (\Delta\phi)^2}$, for all matched pairs in $t\bar{t}$ and di-jet events.

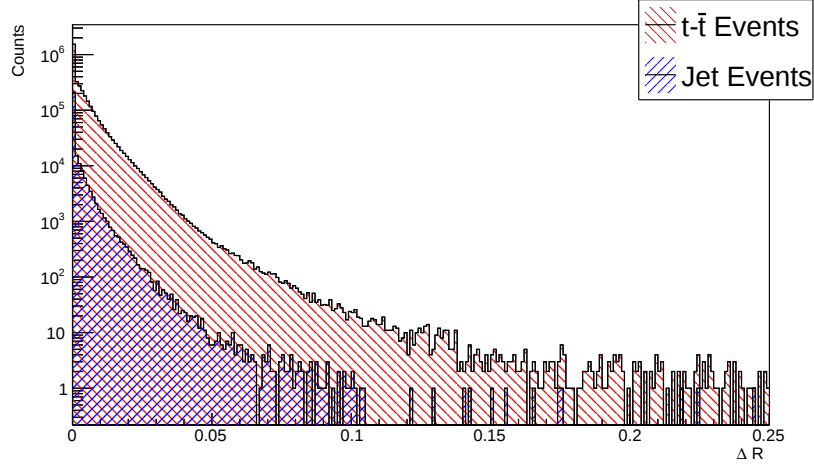


Figure 5.15: Zoomed in version of Figure 5.14, to show the behaviour closer to the origin.

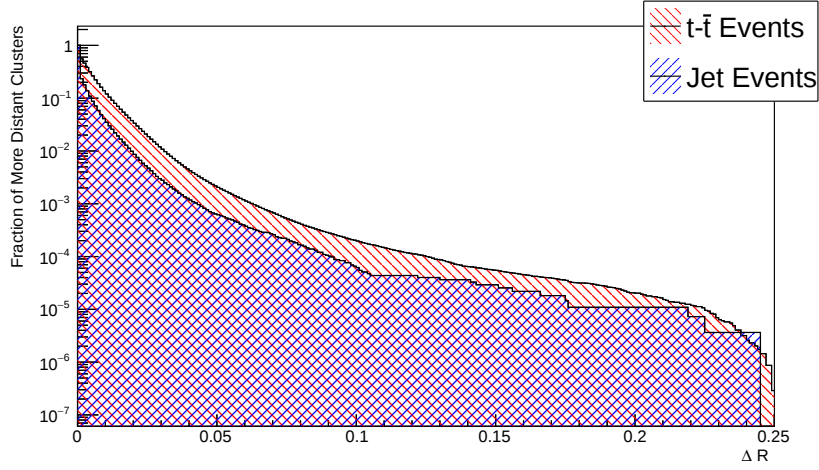


Figure 5.16: Fraction of pairs of matched clusters that, for a value of ΔR , have a greater distance than that value, for $t\bar{t}$ and jet events.

Figure 5.16 shows the fraction of the pairs of matched clusters that, for a given value of ΔR , have a greater distance between them than that (similarly to Figure 5.6 and Figure 5.7, if $f(r)$ is the number of clusters with $\Delta R = r$, we are representing $g(r) = \left(\int_r^{+\infty} f(t)dt \right) / \left(\int_0^{+\infty} f(t)dt \right)$). We can conclude that 99.9% of the clusters have $\Delta R < 0.07$ and 99.99% of the clusters have $\Delta R < 0.13$.

Taking all of this into account, it can be said that, despite not being perfect, the reconstruction of the cluster properties in the GPU implementation is close to that of the CPU implementation.

5.3.5 Effect of Cell Differences

We will now see that the differences identified in the previous validation steps can be attributed almost entirely to the differences in the attribution of terminal cells, by plotting the differences between implementations only for the pairs of clusters that have exactly the same cells⁴.

⁴Since the differences in the attribution of seed or growing cells are much smaller, though not non-existent due to the different noises, doing the plots for clusters that have just the same seed cells or the same seed and growing cells results in distributions

As we can see from Figure 5.17 and Figure 5.18, the energy and transverse energy are reconstructed with very small differences, lower than 0.05% of their values, which is consistent with what could be expected to be caused by floating point approximations.

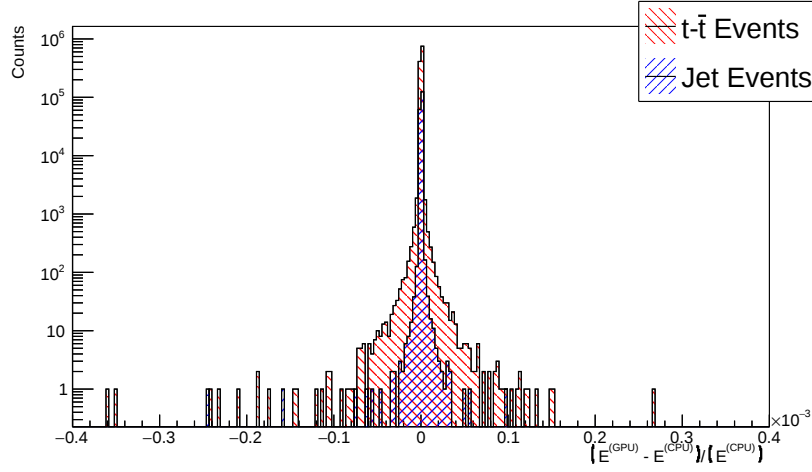


Figure 5.17: Distribution of the relative difference between the energy reconstructed by the GPU implementation and the one from the CPU implementation for pairs of matched clusters that have exactly the same cells, for all events of the two samples. The relative energy difference is calculated with regard to the CPU implementation.

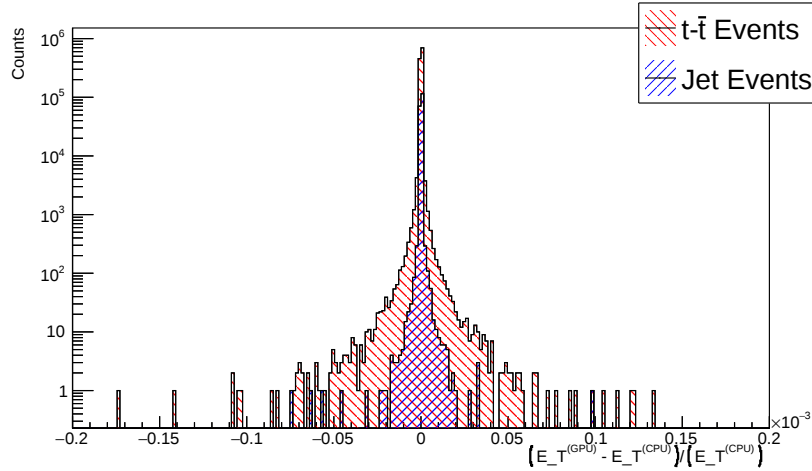


Figure 5.18: Distribution of the relative difference between the transverse energy reconstructed by the GPU implementation and the one from the CPU implementation for pairs of matched clusters that have exactly the same cells, for all events of the two samples. The relative transverse energy difference is calculated with regard to the CPU implementation.

The pseudo-rapidity is reconstructed with great consistency between the two implementations, with the differences being entirely negligible, as Figure 5.19 demonstrates. On the other hand, even within clusters with exactly the same cells, there are some pairs of clusters where the azimuthal angle appears not to be reconstructed correctly, as shown in Figure 5.20. The existence of pairs with $\Delta\phi = \pm\frac{\pi}{2}$ suggests that there may be issues with the angle wrap-around, although there is also at least one other pair, with $\Delta\phi \sim 0.9$, whose origin is not so easy to determine. All of this fall within the scope of the previously suggested further investigations.

that are very similar to the ones from the previous sub-section.

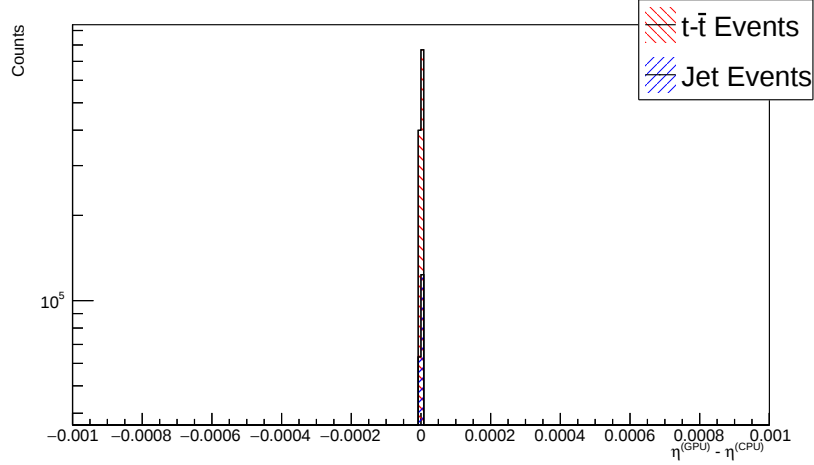


Figure 5.19: Distribution of the differences between the pseudo-rapidity (η) as reconstructed in the GPU and as reconstructed in the CPU, for pairs of matched clusters that have exactly the same cells, for all events of the two samples.

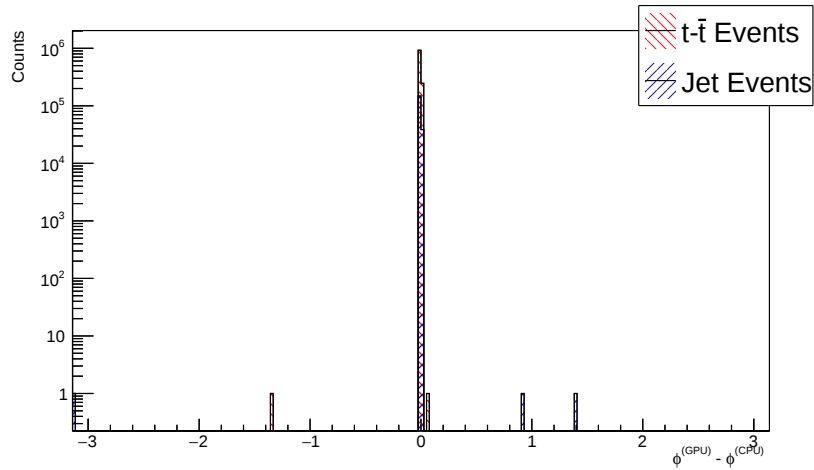


Figure 5.20: Distribution of the differences between the azimuthal angle (ϕ) as reconstructed in the GPU and as reconstructed in the CPU, for pairs of matched clusters that have exactly the same cells, for all events of the two samples.

In short, the issues with ϕ notwithstanding, it is indeed true that the terminal cell attribution is responsible for most of the differences in the reconstructed cluster properties, which allows us to be reasonably sure that the cluster reconstruction is being done as it should be.

5.3.6 Properties of the Unmatched Clusters

Finally, we will investigate the properties of the unmatched clusters, to ensure that the clusters that are not properly identified (either by their presence or absence) will not have much impact in the overall event reconstruction. The simple fact that their number is quite small can be, by itself, a good reason to support that claim, but we are also interested in ensuring that we are not excluding important clusters, both in terms of their size and their energy.

We begin by checking the distribution of unmatched cluster sizes, in Figure 5.21. When compared with the overall cluster size distribution for the same range, which is shown in Figure 5.22, we can see that, in general, the unmatched clusters constitute a negligible fraction of the total number of clusters with the same size. Nevertheless, there are a few unmatched clusters that could be said to be large, as 99% of all clusters have less than 400 cells, which is undesirable. However, it should also be said that some clusters (that have been matched) can have more than 8000 cells, and, in the case of the $t\bar{t}$ sample, up to 50000 cells, which means that the largest clusters tend to be correctly matched.

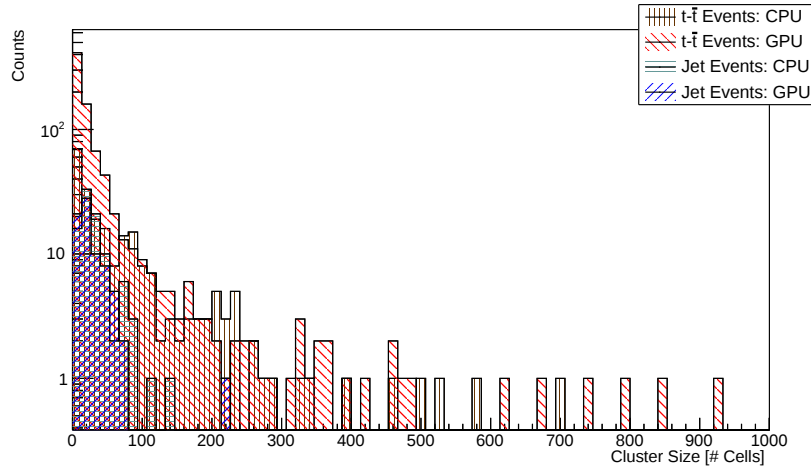


Figure 5.21: Distribution of the number of cells of the clusters identified by one implementation that have not been matched to clusters identified by the other implementation, for the $t\bar{t}$ and di-jet events.

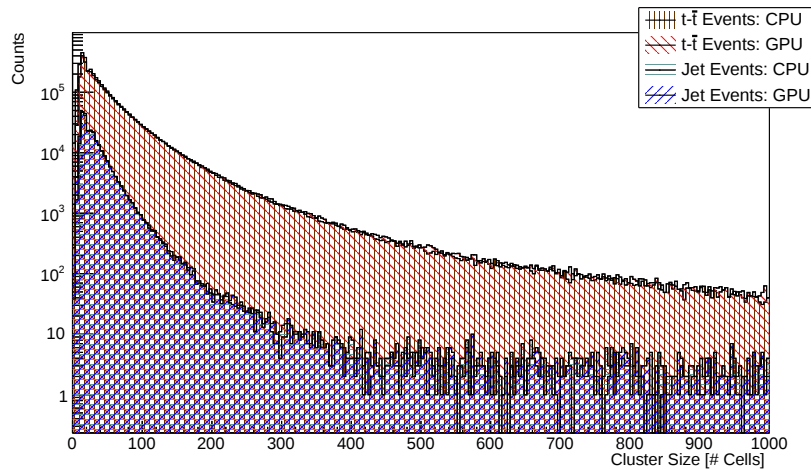


Figure 5.22: Section of the distribution of the number of cells of all clusters, for the GPU and CPU implementations and $t\bar{t}$ and di-jet events, with the same range as Figure 5.21.

As for the energy and transverse energy distributions, which can be seen in Figure 5.23 and Figure 5.24, a comparison with Figure 5.8 and Figure 5.9 makes it clear that the unmatched clusters do not represent a significant fraction of the overall event energy.

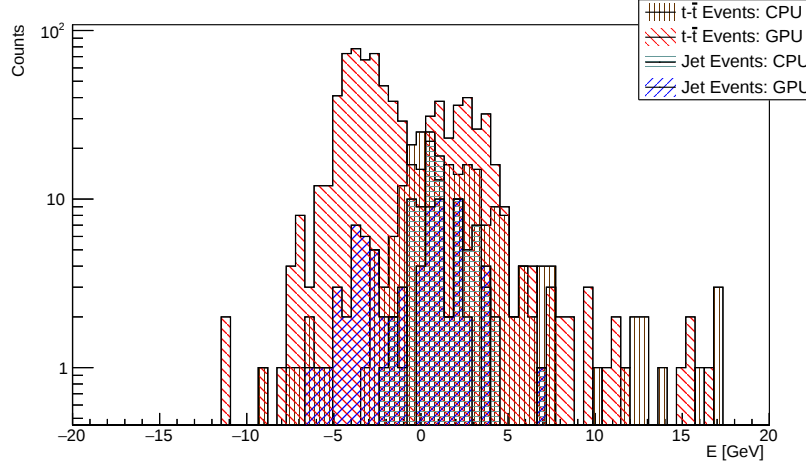


Figure 5.23: Distribution of the energies of the clusters identified by one implementation that have not been matched to clusters identified by the other implementation, for the $t\bar{t}$ and di-jet events.

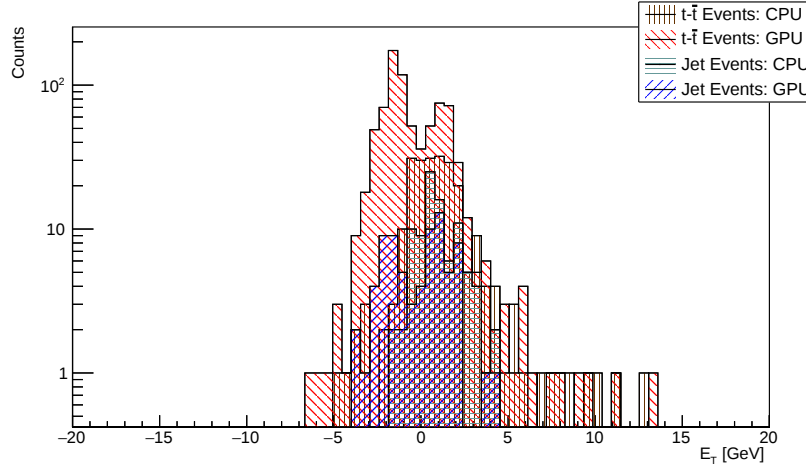


Figure 5.24: Distribution of the transverse energies of the clusters identified by one implementation that have not been matched to clusters identified by the other implementation, for the $t\bar{t}$ and di-jet events.

It is particularly relevant to note that the distributions are roughly symmetrical in relation to the line defined by $x = 0$, with approximately gaussian peaks at $E \simeq \pm 3$ GeV or $E_T \simeq \pm 1$ GeV. This, together with the low energies involved, suggests that most of the unmatched clusters do correspond to noise clusters, thus not representing any actual physical phenomena of interest, being mostly a consequence of the fact that the noise considered in the CPU implementation may vary from event to event.

For completeness' sake, we can also check the possible spatial dependency of the identification of these clusters. In terms of azimuthal angle, Figure 5.25 suggests there is no discernible dependency, which is reassuring as most phenomena (including the noise) should be isotropic in ϕ .

As for the pseudo-rapidity, the unmatched clusters seem to mostly contained within the $[-1.5; 1.5]$ interval, which is within the barrel, with the distribution displaying three peaks centered around $\eta = 0$ and ± 1.1 . Following from the final remark in section 2.3.2.2, the peaks with $\eta = \pm 1.1$ may more or less

correspond to the crack region of the calorimeters, at $\eta = 0$ there is the small gap between the halves of the barrel EM calorimeter.

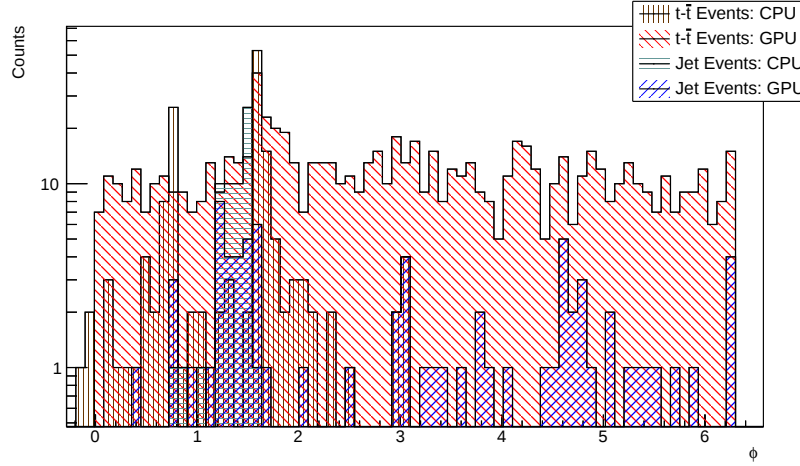


Figure 5.25: Distribution of the azimuthal angles (ϕ) of the clusters identified by one implementation that have not been matched to clusters identified by the other implementation, for the $t\bar{t}$ and di-jet events.

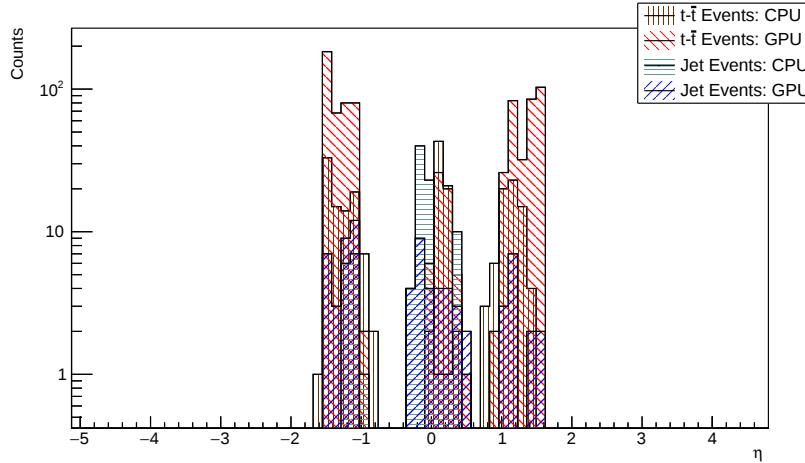


Figure 5.26: Distribution of the pseudo-rapidities (η) of the clusters identified by one implementation that have not been matched to clusters identified by the other implementation, for the $t\bar{t}$ and di-jet events.

All in all, the plots support the notion that the unmatched clusters do not represent a significant loss of physical significance of the results, thereby confirming that, in general, our implementation is sufficiently close to the CPU reference even with the constant noise approximation.

5.4 Performance of the Topo-Automaton Clustering

We will now analyse the event processing time of the GPU implementation, both in comparison to the CPU and in terms of its constituent parts, to better understand the current performance of our code and its potential for future improvement.

5.4.1 Comparison with the CPU Algorithm

Figure 5.27 shows the time taken to process each event, for both samples and both implementations.

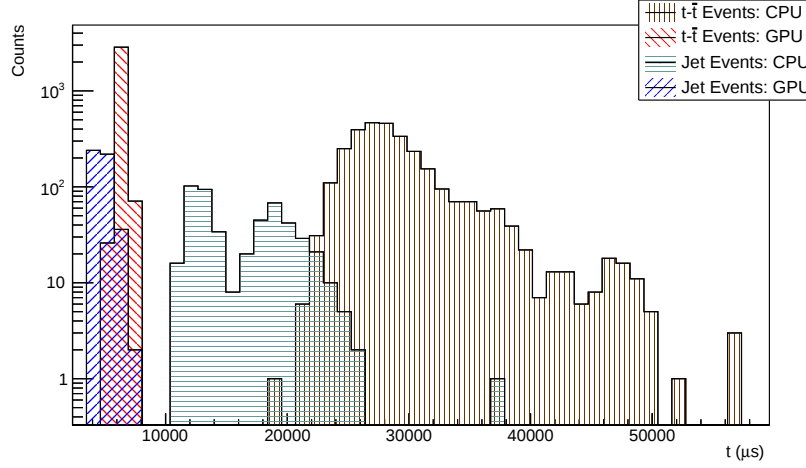


Figure 5.27: Distribution of the time taken to process each event, for $t\bar{t}$ and di-jet events and implementations. These times take into account memory access and transfers, and, in the case of the GPU implementation, the translation between the Athena data structures and those used in our implementation, as described in section 4.3.

It is immediately apparent that the GPU implementation processes the events much quicker, which means that the main objective of this work was achieved. In fact, if we look at the ratio between the GPU and CPU event processing times, in Figure 5.28, we can see that processing an event on the GPU never takes more than 45% of the time necessary to do it on the CPU. On average, the GPU implementation improves the event processing time by a factor of about 5.5 on the $t\bar{t}$ sample and by a factor of about 3.5 on the jet sample, as Figure 5.29 shows.

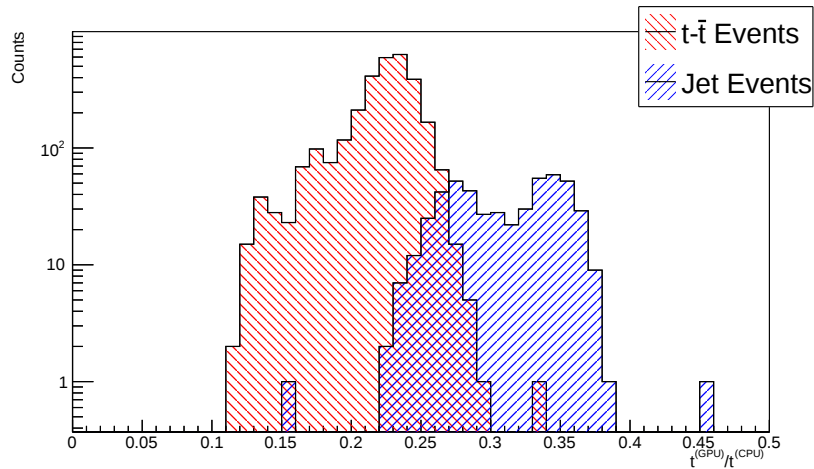


Figure 5.28: Distribution of the ratios between the time taken by the GPU implementation to process an event and that taken by the CPU implementation to do the same.

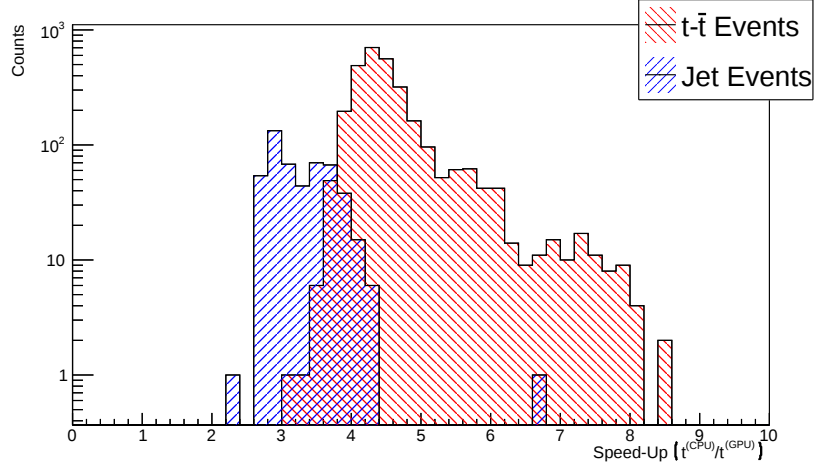


Figure 5.29: Distribution of the speed-up achieved by the GPU implementation in comparison to the CPU implementation in each event of each type, which is the same as the distribution of the ratios between the time taken by the CPU implementation to process an event and that taken by the GPU implementation to do the same (the inverse of the abscissas of Figure 5.28).

If we represent the event processing time for the GPU implementation in a more adequate scale, as in Figure 5.30, we can see that, despite the $t\bar{t}$ sample having a slightly higher processing time in general and in average, the distributions of both samples overlap more closely than for the CPU implementation, where the $t\bar{t}$ events take much more time to be processed. We will explore this behaviour in the next sub-section.

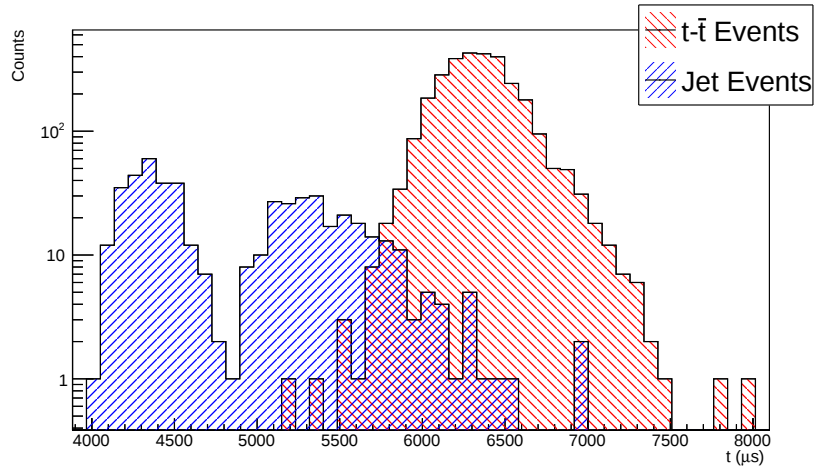


Figure 5.30: Distribution of the time taken by the GPU implementation to fully process an event. Essentially, a zoomed-in version of Figure 5.27 to better highlight the behaviour of the GPU implementation across both event types.

5.4.2 Decomposition of the Processing Times of the GPU Implementation

As a way to estimate the potential for future improvements that still remains in the current implementation, it is interesting to determine the fraction of the total processing time that is taken up by each step of the algorithm. Following the list of steps given in section 4.4.1, and taking the average of their respective execution times over all the events of each sample for simplicity, we can construct Table 5.5.

Table 5.5: Decomposition of the average GPU event processing time, for $t\bar{t}$ and di-jet events. Times are averaged over all events of the respective sample, with the errors being calculated as the standard deviation over all events as well.

Step		$t\bar{t}$ Events		Jet Events	
Preparing the Data	Input Data Conversion	$1736 \pm 62 \mu\text{s}$ (27.34%)	$1057 \pm 43 \mu\text{s}$ (16.66%)	$1785 \pm 47 \mu\text{s}$ (36.41%)	$1106 \pm 44 \mu\text{s}$ (22.56%)
	Input Data Transfer		$678 \pm 19 \mu\text{s}$ (10.68%)		$679 \pm 18 \mu\text{s}$ (13.85%)
Running the Algorithm	Signal-to-Noise Ratio Calculation	$1251 \pm 274 \mu\text{s}$ (19.71%)	$37 \pm 2 \mu\text{s}$ (0.58%)	$824 \pm 235 \mu\text{s}$ (16.80%)	$36 \pm 2 \mu\text{s}$ (0.73%)
	Pair Array Creation		$58 \pm 8 \mu\text{s}$ (0.91%)		$36 \pm 7 \mu\text{s}$ (0.74%)
	Cluster Growing		$1033 \pm 224 \mu\text{s}$ (16.28%)		$650 \pm 235 \mu\text{s}$ (13.27%)
	Final Cluster Properties Calculation		$123 \pm 40 \mu\text{s}$ (1.94%)		$101 \pm 7 \mu\text{s}$ (2.06%)
Packaging the Results	Output Data Transfer	$3362 \pm 280 \mu\text{s}$ (52.96%)	$1257 \pm 252 \mu\text{s}$ (19.79%)	$2294 \pm 484 \mu\text{s}$ (46.79%)	$1251 \pm 482 \mu\text{s}$ (25.52%)
	Output Data Conversion		$2105 \pm 28 \mu\text{s}$ (33.16%)		$1042 \pm 45 \mu\text{s}$ (21.27%)
Total		$6352 \pm 252 \mu\text{s}$		$4906 \pm 614 \mu\text{s}$	

The first and foremost conclusion is that the fraction of the event processing time that was the object of the present work, the step labelled “Running the Algorithm”, corresponds to less than 20% for both event types. Taking into account the kind of parallelism exhibited by this problem, whose size is fixed, the most appropriate model for its scalability is Amdahl’s Law rather than Gustafson’s [63], which implies that the maximum possible speed-up, $s_{\max}$, would have an upper bound given by:

$$s_{\max} \geq \frac{1}{1 - 0.2} = 1.25 \quad (5.4)$$

This means that, by algorithmic changes alone, the maximum improvement that could be seen on the run times would not be greater than 25%. In other words, the data from Table 5.5 suggests that, for the time being, further improvements to the algorithm itself will not result in appreciable improvements in the event processing times if they are not accompanied by other changes, such as in the data model employed within Athena, so as to reduce on the time spent on the data conversions.

Two other properties of the time decomposition need to be addressed. The most puzzling is that, if

the measurements for the expected execution times in section 5.2.2 are compared to the corresponding steps in Table 5.5, we can see that they do not quite match, as Table 5.6 shows clearly.

Table 5.6: Comparison between expected and measured execution times of the GPU event processing steps. Based on the results from tables 5.1, 5.2, 5.3 and 5.4 and the measurements from table 5.5.

Step	$t\bar{t}$ Events		Jet Events	
	$t_{\text{avg}}^{(\text{expected})}$ (μs)	$t_{\text{avg}}^{(\text{measured})}$ (μs)	$t_{\text{avg}}^{(\text{expected})}$ (μs)	$t_{\text{avg}}^{(\text{measured})}$ (μs)
Signal-to-Noise Ratio Calculation	58 ± 10	37 ± 2	58 ± 3	36 ± 2
Pair Array Creation	85 ± 9	58 ± 8	69 ± 8	36 ± 7
Cluster Growing	667 ± 136	1033 ± 224	462 ± 138	650 ± 235
Final Cluster Properties Calculation	125 ± 22	123 ± 40	112 ± 5	101 ± 7

Part of the explanation may lie in the different compiler options used within the Athena build system, when compared to the standalone program that was used for the optimization studies. Perhaps the remote server environment is also partly to blame. In any case, even though there is no fixed relation between the expected and measured execution times, the relation between the $t\bar{t}$ and jet execution times is roughly the same, which allows us to conjecture that the arguments presented in section 5.2.2 for the choice of implementations should still remain generally valid.

Secondly, but unsurprisingly, there is a noticeable difference between the output data conversion times for both samples. As this step entails populating a list of the clusters that were found in the GPU in CPU memory and filling those clusters with the appropriate cells, it will tend to take longer for events with more clusters and/or larger clusters, both of which apply to the $t\bar{t}$ sample.

As for the data transfer from and to the GPU, the times are consistent across both implementations, which is precisely what was expected since the size of the objects being transferred is fixed.

Chapter 6

Conclusions

In the course of the present work, an implementation of the Topo-Automaton Clustering algorithm using the `CUDA` GPU programming framework was successfully developed and integrated with Athena, the software-based trigger of the ATLAS experiment. Different optimization strategies were considered and their performance was compared based on two sets of events (di-jets and $t\bar{t}$) to select the final fastest strategy.

The results of this final strategy were compared with the CPU-based implementation of the Topological Clustering algorithm, which is what the ATLAS trigger currently uses to reconstruct the showers in the calorimeters. Through this comparison, the impact of the differences between the algorithms (namely in the attribution of terminal cells to the clusters) and of the approximations made for the GPU implementation (namely that the noise in the calorimeter cells is constant throughout the events) on the physical validity of the reconstruction is determined. Though some differences were found, 99.99% of the clusters reconstructed in the GPU have a distance of $\Delta R < 0.13$ in relation to those found by the CPU implementation, and the energies and transverse energies are also consistent between both implementations, which suggests physical validity is maintained.

Finally, the execution time of the GPU implementation of Topo-Automaton Clustering was compared to the Topological Clustering in the CPU, yielding an overall improvement by a factor of 3.5 in di-jet events and of 5.5 on $t\bar{t}$ events, on average, which proves that GPU acceleration is a very promising strategy for the shower reconstruction.

We will now go into some more detail regarding each of these points, as well as some possible avenues for future work.

6.1 Optimization of the Algorithm

The optimization studies of the algorithm provide interesting data regarding the relative advantages of the different implementation strategies. Though the results were obtained for a particular combination

of hardware, we would be justified in expecting the ordering of the strategies in terms of their run time to be roughly the same for other systems.

One of the main unknowns was the effect of the use of 64-bit variables necessary for the second tagging strategy on the performance of the algorithm, given that the architecture of GPUs tends to be more focused on 32-bit operation. However, upon more close inspection of the CUDA arithmetic instructions specification [42], the less favourable 64-bit operations are in fact integer multiplications and divisions, as 64-bit arithmetic is emulated using 32-bit instructions. Since most 64-bit operations in our algorithm correspond to maxima or bit-wise composition (which can even be expressed as two 32-bit stores), it is quite likely that the performance overhead is negligible, as the optimization results seem to support, in particular, the first two entries of Table 5.2, for which the only change is in the tagging strategy and whose execution times are statistically indistinguishable. However, this might not necessarily be the case for all hardware, in particular for older hardware, as more recent devices have been built with greater support for 64-bit operations.

However, even if there were some additional overhead associated with 64-bit tags, the possibility of recovering the cluster index directly from the tag allows for a much faster cluster reconstruction, and, above all, for the inclusion of an explicit merge step, which represents a significant reduction of the execution time of the algorithm, as shown by Table 5.3.

As mentioned in section 5.2.2, this lack of an impact on performance is in great deal a consequence of the fact that, for implicit cluster merges, the tag must propagate from the point where the clusters touch back to all of the other cells. Given an approximately spherical cluster with d cells in its diameter that will be merged to another cluster whose center is r cells away from the point where the clusters touch, the merge will be concluded $\mathcal{O}(d + r)$ iterations after the start of tag propagation, while an explicit merge will only take $\mathcal{O}(\min(\frac{d}{2}, r))$ iterations to be concluded.

Another relevant (and not necessarily expected) conclusion is that the creation of a specific list containing only the pairs that can propagate tags at each event seems to be the most sensible strategy, as the performance penalty of considering all possible pairs of cells during tag propagation is greater than the time saved by not building the specific list of pairs. This is likely due to the fact that, with fixed pairs, the classification of the cells needs to be checked at each step to prevent propagation between terminal cells, which constitutes a measurable overhead, and, in any case, the increased number of pairs to be checked does not result in additional performance from the point in which the maximum possible number of concurrent blocks are being executed in the GPU.

The fact that the more sophisticated strategies that forewent the creation of an explicit list of pairs and tried to minimize atomic operations on global memory actually pessimized the run time is also somewhat unexpected. This might be due to the implementation being sub-optimal for GPU operation, while global memory atomic operations have been increasing in efficiency with later GPU architectures (as well as the CUDA compiler's capabilities of optimizing them). It could be interesting to try to repeat the measurements with other, possibly older, hardware.

6.2 Validity of Physical Reconstruction

From the results shown in the previous chapter, one of the main conclusions is that the GPU implementation of Topo-Automaton Clustering is able to reconstruct most of the clusters that are identified by the existing CPU implementation with reasonable accuracy, with 99.9% of the clusters having $\Delta R < 0.07$ and 99.99% of the clusters having $\Delta R < 0.13$ in relation to the CPU reconstruction. Nevertheless, there are some indications that the azimuthal angle reconstruction may have some problems that will need to be diagnosed, possibly related to some interplay between angle wrap-around and the finite precision of the floating point format.

In terms of the energy and transverse energy reconstruction, though the values do not match perfectly, that can be mostly attributed to the fact that the algorithm in the CPU implementation assigns terminal cells to the clusters using a different criterium, together with the approximation employed in the GPU implementation that the noise at each calorimeter cell is constant throughout all events.

Though there have been some events in which some clusters have only been identified by one of the implementations, those discrepancies are also likely caused by the aforementioned constant noise approximation, since their energy distribution is consistent with noise clusters.

Taking all of this into account, the results support the claim that the implementation of the Topo-Automaton Clustering algorithm does not incur a meaningful loss of physical significance despite the differences from the reference implementation and is thus suited for use in the ATLAS Trigger. It would be important, however, to test it more thoroughly with larger samples, preferably with higher pile-up.

6.3 Time Performance of the Algorithm

The improvement of the overall event processing time by a factor of 3.5 in di-jet events and 5.5 on $t\bar{t}$ events proves the feasibility of employing GPU acceleration in the cluster growing process in the ATLAS trigger. As the pile-up conditions of the samples that were used to measure the performance of the implementations were quite low in comparison to the expectations for the High-Luminosity Upgrade, and given that the speed-up tended to be greater for the denser $t\bar{t}$ events, it is reasonable to assume that the factors of 3.5 and 5.5 will represent a lower bound to the speed-up achieved in events with higher pile-up. Naturally, it will be relevant to verify this when appropriate samples are available.

However, and somewhat worryingly, what is most relevant for future developments is the fact that less than 20% of the event processing time is spent on the algorithm itself. This suggests that any further efforts to optimize the algorithm would have diminishing returns, as Amdahl's Law predicts an upper bound on the speed-up of about 1.25 in relation to current run times, all other conditions being the same.

Extrapolating from these results, it would seem that one of the most significant bottlenecks for GPU acceleration is the data model used within Athena, which means that, if development efforts were dedicated

to finding a more GPU-friendly representation (which is doubtlessly a daunting task, since changes in that data model would have repercussions throughout the entire codebase), GPU acceleration in general would be a more attractive option for other parts of the event reconstruction. An alternative approach would be performing several steps of the reconstruction in the GPU, which could cut down on the data transfers given the fact that the necessary information for a step would already be present in the GPU memory as a consequence of the previous steps. One particularly relevant case in which it makes sense to do this is cluster splitting, which must be done immediately after cluster growing and only depends on its results.

6.4 Future Work

Based on all that was done in the present work, a few avenues for future improvement present themselves:

- Regarding the physical validity of the algorithm:
 - Investigating the possible bias in the cluster identification when compared to the CPU implementation
 - Investigating the issues with the azimuthal angle calculation
 - Comparing the current terminal cell attribution and the CPU cell attribution with the “truth” particles from Monte-Carlo to determine which one is preferable
 - Considering possible alternatives for the terminal cell attribution criterium to increase the verisimilitude of the results
- Regarding the usability of the algorithm:
 - Including at least some of the configurability of the CPU algorithm (possibly through different classes/class templates rather than run-time specified values, to dispense with additional memory accesses to control the behaviour of our implementation)
 - Testing multi-threaded operation, with several events being processed in parallel, and implementing the necessary changes if or when necessary
- Regarding the optimization of the implementation:
 - Comparing the results of the optimization procedure to those obtained on a greater variety of hardware configurations (including older systems)
 - Measuring the speed-up when processing events with higher pile-up, closer to the operating conditions of the High-Luminosity LHC
 - Considering the possibility of joining the algorithm with other GPU-accelerated reconstruction steps (namely cluster splitting) to reduce data transfers to and from the GPU

- Despite the diminishing returns, considering additional alternative implementations that could reduce the number of atomic operations during tag propagation and/or the number of kernel

Regarding this latter point, a few comments must be made, based on the cursory exploration of these alternatives that the author has done in the context of the development of the present work. Due to the fact that most of the cells tend to have many neighbours (averaging around 13 neighbours per cell) which, to make matters worse, are more or less spread throughout the different parts of the calorimeter system without any widespread regularity, there is no obvious way of grouping the cells such that most cells that neighbour each other can be processed by the same thread block in shared memory. Furthermore, the lack of inter-block synchronization (except through the use of cooperative groups, which, as justified in section 4.4.5, does not seem reasonable at the moment) prevents the efficient implementation of strategies that minimize atomic operations by maintaining some form of cache at block-level, given that all the possible synchronization strategies (as discussed in section 4.4.5) rely on launching multiple kernels, shared memory does not persist between kernel invocations and copying from and to shared memory has an overhead that is very likely greater than simply accessing global memory.

It is also interesting to consider the potential of more significant changes to the algorithm to achieve greater optimization. It is possible to formulate the cluster growing process as a particular form of graph traversal, with the cells acting as the graph's nodes and their neighbourhood relations as its edges. Due to the applicability of graph-based approaches to a wide variety of problems, efficient GPU-based graph traversal algorithms are an active area of study (see, for example, [64, 65, 66, 67]), and, even though the scope of these algorithms, especially in terms of the expected size of the graphs, tends to be somewhat different from what would be necessary for cluster growing, it might still be worthwhile to consider these alternatives.

References

- [1] H. Damerau, A. Funken, R. Garoby, et al. **LHC Injectors Upgrade, Technical Design Report**. Dec. 2014. doi: 10.17181/CERN.7NHR.6HGC. URL: <https://cds.cern.ch/record/1976692>.
- [2] Y. Kadi, M. A. Fraser, and A. Papageorgiou-Koufidou. **HIE-ISOLDE: Technical Design Report for the Energy Upgrade**. CERN Yellow Reports: Monographs. CERN, Geneva, May 2018. doi: 10.23731/CYRM-2018-001. URL: <https://cds.cern.ch/record/2635892>.
- [3] S. A. Baird, D. Berlin, J. Boillot, et al. **Design Study of the Antiproton Decelerator: AD**. Nov. 1996. doi: 10.17181/CERN.KNXM.AYKR. URL: <https://cds.cern.ch/record/317704>.
- [4] A. Caldwell, E. Gschwendtner, K. Lotov, et al. **AWAKE Design Report: A Proton-Driven Plasma Wakefield Acceleration Experiment at CERN**. Technical report, Apr. 2013. URL: <https://cds.cern.ch/record/1537318>.
- [5] E. Mobs. **The CERN Accelerator Complex - 2019. Complexe des Accélérateurs du CERN - 2019**, Jul. 2019. URL: <https://cds.cern.ch/record/2684277>.
- [6] L. Evans and P. Bryant. **LHC Machine**. *Journal of Instrumentation*, 3(08):S08001–S08001, Aug. 2008. doi: 10.1088/1748-0221/3/08/s08001.
- [7] S. Redaelli, I. Agapov, R. Calaga, et al. **First Beam Based Aperture Measurements in the Arcs of the CERN Large Hadron Collider**. May 2009. URL: <https://cds.cern.ch/record/1211048>.
- [8] The ALICE Collaboration. **ALICE: Technical Proposal for a Large Ion collider Experiment at the CERN LHC**. LHC technical proposal. CERN, Geneva, 1995. URL: <https://cds.cern.ch/record/293391>.
- [9] The ATLAS Collaboration. **ATLAS: Technical Proposal for a General-Purpose pp Experiment at the Large Hadron Collider at CERN**. LHC technical proposal. CERN, Geneva, 1994. URL: <https://cds.cern.ch/record/290968>.
- [10] The ATLAS Collaboration. **The ATLAS Experiment at the CERN Large Hadron Collider**. *JINST*, 3:S08003. 437 p, 2008. doi: 10.1088/1748-0221/3/08/S08003. URL: <https://cds.cern.ch/record/1129811>.
- [11] The CMS Collaboration. **CMS, the Compact Muon Solenoid: Technical Proposal**. LHC technical proposal. CERN, Geneva, 1994. URL: <https://cds.cern.ch/record/290969>.

- [12] The CMS Collaboration. ***The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment.*** *JINST*, 3:S08004. 361 p, 2008. doi: 10.1088/1748-0221/3/08/S08004. URL: <https://cds.cern.ch/record/1129810>.
- [13] A. A. Alves, L. M. Andrade, F. Barbosa-Ademarlaudo, et al. ***The LHCb Detector at the LHC.*** *JINST*, 3:S08005, 2008. doi: 10.1088/1748-0221/3/08/S08005. URL: <https://cds.cern.ch/record/1129809>.
- [14] O. Adriani, L. Bonechi, M. Bongi, et al. ***LHCf Experiment: Technical Design Report.*** Technical design report. LHCf. CERN, Geneva, 2006. URL: <https://cds.cern.ch/record/926196>.
- [15] J. Pinfold, R. Soluk, Y. Yao, et al. ***Technical Design Report of the MoEDAL Experiment.*** Technical report, Jun. 2009. URL: <https://cds.cern.ch/record/1181486>.
- [16] V. Berardi, M. G. Catanesi, E. Radicioni, et al. ***Total Cross-Section, Elastic Scattering and Diffraction Dissociation at the Large Hadron Collider at CERN: TOTEM Technical Design Report.*** Technical design report. TOTEM. CERN, Geneva, 2004. URL: <https://cds.cern.ch/record/704349>.
- [17] A. Ariga, T. Ariga, J. Boyd, et al. ***Technical Proposal: FASER, the Forward Search Experiment at the LHC.*** Technical report, CERN, Geneva, Dec. 2018. URL: <https://cds.cern.ch/record/2651328>.
- [18] W. Herr and B. Muratori. ***Concept of Luminosity.*** 2006. doi: 10.5170/CERN-2006-002.361. URL: <https://cds.cern.ch/record/941318>.
- [19] E. Courant and H. Snyder. ***Theory of the Alternating-Gradient Synchrotron.*** *Annals of Physics*, 3(1):1–48, 1958. ISSN 0003-4916. doi: 10.1016/0003-4916(58)90012-5.
- [20] G. Apollinari, I. Béjar Alonso, O. Brüning, et al. ***High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1.*** 2017. doi: 10.23731/CYRM-2017-004. URL: <http://cds.cern.ch/record/2284929>.
- [21] J. Pequenão. ***Computer Generated Image of the Whole ATLAS Detector,*** Mar. 2008. URL: <https://cds.cern.ch/record/1095924>.
- [22] C.-Y. Wong. ***Introduction to High-Energy Heavy-Ion Collisions.*** World Scientific Pub Co Inc, 1994. ISBN 9810202636; 9789810202637.
- [23] J. Pequenão. ***Computer Generated Image of the ATLAS Inner Detector,*** Mar. 2008. URL: <https://cds.cern.ch/record/1095926>.
- [24] The ATLAS Collaboration. ***Expected Tracking Performance of the ATLAS Inner Tracker at the HL-LHC.*** Technical report, CERN, Geneva, Mar. 2019. URL: <https://cds.cern.ch/record/2669540>.

- [25] J. Pequeno. **Computer Generated Image of the ATLAS Calorimeter**, Mar. 2008. URL: <https://cds.cern.ch/record/1095927>.
- [26] J. Pequeno. **Computer Generated Image of the ATLAS Muons Subsystem**, Mar. 2008. URL: <https://cds.cern.ch/record/1095929>.
- [27] The ATLAS Collaboration. **Technical Design Report for the Phase-I Upgrade of the ATLAS TDAQ System**. Technical report, Sep. 2013. URL: <https://cds.cern.ch/record/1602235>.
- [28] The ATLAS Collaboration. **Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System**. (CERN-LHCC-2017-020. ATLAS-TDR-029), Sep. 2017. URL: <https://cds.cern.ch/record/2285584>.
- [29] P. Calafiura, W. Lavrijsen, C. Leggett, et al. **The Athena Control Framework in Production, New Developments and Lessons Learned**. 2005. doi: 10.5170/CERN-2005-002.456. URL: <https://cds.cern.ch/record/865624>.
- [30] The ATLAS Collaboration. **ATLAS Athena Guide**, retrieved 24 Jun. 2021. URL: <https://web.archive.org/web/20210624221115/https://atlassoftwaredocs.web.cern.ch/athena>.
- [31] The ATLAS Collaboration. **Athena**, May 2021. URL: <https://doi.org/10.5281/zenodo.4772550>.
- [32] The ATLAS Collaboration. **Multi-Threaded Software Framework Development for the ATLAS Experiment**. Technical report, CERN, Geneva, Mar. 2016. URL: <https://cds.cern.ch/record/2142775>.
- [33] The ATLAS Collaboration. **AthenaMT: Upgrading the ATLAS Software Framework for the Many-Core World with Multi-Threading**. Technical report, CERN, Geneva, Jan. 2017. URL: <https://cds.cern.ch/record/2242909>.
- [34] The ATLAS Collaboration. **ATLAS High Level Trigger within the Multi-Threaded Software Framework AthenaMT**. Technical report, CERN, Geneva, May 2019. URL: <https://cds.cern.ch/record/2674286>.
- [35] The ATLAS Collaboration. **Topological Cell Clustering in the ATLAS Calorimeters and its Performance in LHC Run 1**. *The European Physical Journal C*, 77(7), Jul. 2017. ISSN 1434-6052. doi: 10.1140/epjc/s10052-017-5004-5.
- [36] The ATLAS Collaboration. **Calorimeter Clustering Algorithms: Description and Performance**. (ATL-LARG-PUB-2008-002. ATL-COM-LARG-2008-003), Apr. 2008. URL: <https://cds.cern.ch/record/1099735>.
- [37] T. S. Crow. **Evolution of the Graphical Processing Unit**. Master's thesis, University of Nevada Reno, 2004.
- [38] T. N. Theis and H.-S. P. Wong. **The End of Moore's Law: A New Beginning for Information Technology**. *Computing in Science & Engineering*, 19(2):41–50, 2017.

- [39] G. E. Moore. ***Cramming More Components onto Integrated Circuits***, 1965.
- [40] G. E. Moore. ***Progress in Digital Integrated Electronics***. In *Electron Devices Meeting*, volume 21, pages 11–13. Washington, DC, 1975.
- [41] J. Nickolls, I. Buck, M. Garland, and K. Skadron. ***Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For?*** *Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500.
- [42] The NVIDIA Corporation. ***CUDA C++ Programming Guide***, retrieved 23 Jul. 2021. URL: <https://web.archive.org/web/20210723052038/https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [43] J. E. Stone, D. Gohara, and G. Shi. ***OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems***. *Computing in Science Engineering*, 12(3):66–73, 2010. doi: 10.1109/MCSE.2010.69.
- [44] J. Fang, A. L. Varbanescu, and H. Sips. ***A Comprehensive Performance Comparison of CUDA and OpenCL***. In *2011 International Conference on Parallel Processing*, pages 216–225, 2011. doi: 10.1109/ICPP.2011.45.
- [45] K. Karimi, N. G. Dickson, and F. Hamze. ***A Performance Comparison of CUDA and OpenCL***. 2011.
- [46] S. Memeti, L. Li, S. Pillana, et al. ***Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption***. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing, ARMS-CC '17*, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351164. doi: 10.1145/3110355.3110356.
- [47] A. Asaduzzaman, A. Trent, S. Osborne, et al. ***Impact of CUDA and OpenCL on Parallel and Distributed Computing***. In *2021 8th International Conference on Electrical and Electronics Engineering (ICEEE)*, pages 238–242, 2021. doi: 10.1109/ICEEE52452.2021.9415927.
- [48] A. T. Delgado and D. Emelianov. ***ATLAS Trigger Algorithms for General Purpose Graphics Processor Units***. In *2016 IEEE Nuclear Science Symposium, Medical Imaging Conference and Room-Temperature Semiconductor Detector Workshop (NSS/MIC/RTSD)*, pages 1–6, 2016. doi: 10.1109/NSSMIC.2016.8069670.
- [49] The ATLAS Collaboration. ***GPU-Based Tracking Algorithms for the ATLAS High-Level Trigger***. (ATL-DAQ-PROC-2012-006), May 2012. doi: 10.1088/1742-6596/396/1/012018. URL: <https://cds.cern.ch/record/1450130>.
- [50] M. Flynn. ***Very High-Speed Computing Systems***. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. doi: 10.1109/PROC.1966.5273.

- [51] A. Delgado. **Development of Jet Trigger Algorithms for the ATLAS Experiment at the LHC**. PhD thesis.
- [52] The ATLAS Collaboration. **Multi-threaded Algorithms for GPGPU in the ATLAS High Level Trigger**. (ATL-DAQ-PROC-2016-045. 3), Dec. 2016. doi: 10.1088/1742-6596/898/3/032003. URL: <https://cds.cern.ch/record/2239823>.
- [53] The NVIDIA Corporation. **CUDA Toolkit Documentation - Thrust**, retrieved 23 Jul. 2021. URL: <https://web.archive.org/web/20210723205256/https://docs.nvidia.com/cuda/thrust/index.html>.
- [54] D. I. Arkhipov, D. Wu, K. Li, and A. C. Regan. **Sorting with GPUs: A Survey**. 2017.
- [55] **IEEE Standard for Floating-Point Arithmetic**. IEEE STD 754-2019 (Revision of IEEE 754-2008), pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.
- [56] S. Ashkiani, M. Farach-Colton, and J. Owens. **A Dynamic Hash Table for the GPU**. 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 419–429. 05 2018. doi: 10.1109/IPDPS.2018.00052.
- [57] D. Alcantara, V. Volkov, S. Sengupta, et al. **Building an Efficient Hash Table on the GPU**. GPU Computing Gems Jade Edition. 08 2011. ISBN 9780123859631. doi: 10.1016/B978-0-12-385963-1.00004-6.
- [58] D. Alcantara, A. Sharf, F. Abbasinejad, et al. **Real-Time Parallel Hashing on the GPU**. ACM Transactions on Graphics, 28:154:1–, 12 2009. doi: 10.1145/1661412.1618500.
- [59] W. Armour. **Warp Shuffles, Reduction and Scan Operations – Lecture Notes**, 2019. Retrieved 13 Apr. 2021. URL: https://web.archive.org/web/20210413214801/https://people.maths.ox.ac.uk/gilesm/cuda/2019/lecture_04.pdf.
- [60] T. Sjöstrand, S. Ask, J. R. Christiansen, et al. **An Introduction to PYTHIA 8.2**. Computer Physics Communications, 191:159–177, Jun. 2015. ISSN 0010-4655. doi: 10.1016/j.cpc.2015.01.024.
- [61] D. Gale and L. S. Shapley. **College Admissions and the Stability of Marriage**. The American Mathematical Monthly, 69(1):9–15, 1962. ISSN 00029890, 19300972. doi: 10.2307/2312726.
- [62] R. W. Irving. **Stable Marriage and Indifference**. Discrete Applied Mathematics, 48(3):261–272, 1994. ISSN 0166-218X. doi: 10.1016/0166-218X(92)00179-P.
- [63] J. L. Gustafson. **Reevaluating Amdahl’s Law**. Communications of the ACM, 31(5):532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415.
- [64] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. **CuSha: Vertex-Centric Graph Processing on GPUs**. In Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC ’14, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327497. doi: 10.1145/2600212.2600227.

- [65] Y. Wang, Y. Pan, A. Davidson, et al. **Gunrock: GPU Graph Analytics**. *ACM Trans. Parallel Comput.*, 4(1), Aug. 2017. ISSN 2329-4949. doi: 10.1145/3108140.
- [66] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao. **Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing**. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, page 622–636. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450349116. URL: <https://doi.org/10.1145/3173162.3173180>.
- [67] P. Wang, L. Zhang, C. Li, and M. Guo. **Excavating the Potential of GPU for Accelerating Graph Traversal**. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 221–230, 2019. doi: 10.1109/IPDPS.2019.00032.
- [68] C. Marcelloni. **View of the ATLAS Detector during July 2007. Vue Détecteur ATLAS en Juillet 2007**, Jul. 2007. URL: <https://cds.cern.ch/record/1057769>.
- [69] The ATLAS Experiment. **A Rack of ATLAS TDAQ Components**, Apr. 2014. URL: <https://cds.cern.ch/record/1696907>.
- [70] The NVIDIA Corporation. **Tesla V100S**, retrieved 28 Mar. 2021. URL: https://web.archive.org/web/20210328004828/https://www.nvidia.com/content/dam/en-zz/es_em/Solutions/Data-Center/tesla-v100/data-center-tesla-v100-nvlink-625-ud@2x.jpg.
- [71] S. Mehlhase. **ATLAS Logo**, Jan. 2014. URL: <https://cds.cern.ch/record/1170735>.

References [68, 69, 70, 71] provided most of the elements of the cover image.

Appendix A

Code for the Topo-Automaton Clustering Implementation

A.1 General Definitions

First of all, we should specify the choice that we are making for the thresholds:

```
constexpr float CellThreshold = 0.0f;  
constexpr float GrowThreshold = 2.0f;  
constexpr float SeedThreshold = 4.0f;
```

We must define the bounds for the arrays that make up the data structures (as well as some temporaries used in some strategies, namely with the intrinsic pairs strategy):

```
constexpr int NMaxNeighbours = 26;  
//The maximum number of neighbours of a cell  
constexpr int NCaloCells = 187652;  
//The total number of cells in the calorimeter  
constexpr int NMaxClusters = 100000;  
//A very lax upper bound on the maximum number of clusters  
constexpr int NMaxPairs = NMaxNeighbours * NCaloCells;  
//The maximum number of possible cell pairs  
constexpr int WarpSize = 32;  
//The size of a warp of threads  
constexpr int MaxBlockSize = 1024;  
//The maximum number of threads per block
```

Following what was presented in section 4.2.1.3, we must also keep in mind the special tag values:

```
constexpr int GrowTag = 2;  
constexpr int TerminalTag = 1;  
constexpr int InvalidTag = 0;  
//Integer promotion rules ensure this will be equally valid  
//when the tags are unsigned long long (i. e. second tagging strategy)
```

Since, at some points, we will need to deal with trigonometric variables, we provide a type-safe definition of π to prevent unexpected conversions to and from `double` (as could happen if using the usual macro definition provided by most common implementations of the C++ standard library, `M_PI`):

```
template <class T>
constexpr T pi = static_cast<T>(/* pi with 512 significant digits */);
// This will round pi to the maximum available precision in the type
// and will be perfectly portable and extendable
// (except in the unlikely case that the type supports even greater precision)
```

Finally, for properly calculating the grid sizes for kernels that operate over arrays of size N with a fixed block size B , it is useful to consider $\lceil N/B \rceil$, which can be implemented as:

```
__host__ __device__ inline constexpr
int int_ceil_div(const int num, const int denom)
{
    return num / denom + (num % denom != 0);
}
```

For shortness of notation whenever the code is (mostly) the same for both choices of tagging strategy, let `tag_type` be `int` (corresponding in CUDA to `int32_t`) if adopting the first tagging strategy or `unsigned long long int` (corresponding to `uint64_t`) if adopting the second tagging strategy.

A.2 Data Structures

A.2.1 Object Wrappers

The idea behind the implementation is that the `Helpers::X_object<T>` (where X is `CPU`, `CUDA` or `CUDA_kernel`) are specializations of a more general template `Helpers::SimpleHolder<T, MemoryContext, holds_memory>`.

In that template, `holds_memory` controls if the object will perform memory allocations, de-allocations and copies, when `true`, or, if `false`, will only act as a non-owning pointer to other memory. This is used to give `Helpers::CUDA_kernel_object` its intended behaviour of acting as an explicitly non-owning pointer and makes for clearer code.

As for `MemoryContext`, this corresponds to a class that identifies the location where the memory corresponding to the object should reside. This will be used by a more general `Helpers::Manager<T>` class whose static `allocate<Context>`, `deallocate<Context>`, `copy<SourceContext, DestContext>` and `move<SourceContext, DestContext>` perform the obvious memory operations for objects of type `T` in the specified memory locations. Definitions for such operations in CPU and GPU memory are provided, using `new`, `delete` and `std::memcpy` and `cudaMalloc`, `cudaFree` and `cudaMemcpy`. This system is of course extensible for other possible memory spaces, though that is not relevant for the present work.

As for the classes themselves, they are in essence simply:

```
template <class T, class MemoryContext, bool holds_memory>
struct Helpers::SimpleHolder
{
private:
    T* m_object;
```

```

public:
    const T& operator *() const { return *m_object; }
    T& operator *() { return *m_object; }
    const T* operator ->() const { return m_object; }
    T* operator ->() { return m_object; }
    bool valid() const { return m_object != nullptr; }
    operator const T*() const { return m_object; }
    operator T*() { return m_object; }
    //Then follow several copy-constructors, move-constructors,
    //assignment operators and move operators
    //to perform the expected conversions
    //between objects with possibly different MemoryContexts
    //...
    //And, when holds_memory is false,
    void clear() { Manager::template deallocate<Context>(m_object); }
    void allocate() { if(!m_object) m_object = Manager::template allocate<Context>(1); }
};

```

The actual implementation is somewhat more complicated due to the fact that the CUDA version in use limits us to C++14, which means that the differences between the templates cannot rely on `if constexpr` to specify the different behaviours, instead needing to be different specializations of the base template. As all of this happens at compile time, there is no performance impact.

The template specializations that result in the helper classes are as follows:

- `Helpers::CPU_object<T>` → `Helpers::SimpleHolder<T, MemoryContext::CPU, true>`
- `Helpers::CUDA_object<T>` → `Helpers::SimpleHolder<T, MemoryContext::CUDAGPU, true>`
- `Helpers::CUDA_kernel_object<T>` → `Helpers::SimpleHolder<T, MemoryContext::CUDAGPU, false>`

A.2.2 General Data Structures

```

class ConstantDataHolder
{
public:

    void sendGeoGPUData(const bool clear_CPU = true);
    //Sends the GeometryArr to the GPU (and optionally clears it from the CPU)
    void sendNoise(const bool clear_CPU = true);
    //Sends the CellNoiseArr to the GPU (and optionally clears it from the CPU)

    void prepGeoGPUData(LArNeighbours::neighbourOption, const bool, const bool);
    // Fills the GeometryArr in CPU memory.
    void prepNoise(const CaloNoise *NoiseTool,
                  SG::ReadHandle<CaloCellContainer>& pCaloCellContainer);
    // Fills the CellNoiseArr in CPU memory.

    Helpers::CPU_object<GeometryArr> m_geometry;
    Helpers::CPU_object<CellNoiseArr> m_cell_noise;
    Helpers::CUDA_object<GeometryArr> m_geometry_dev;

```

```

Helpers::CUDA_object<CellNoiseArr> m_cell_noise_dev;

//void prepPairs();
//Helpers::CUDA_object<PairsArr> m_pairs_dev;
//If using fixed pairs.
};

class EventDataHolder
{
public:
    void sendData(const bool clear_CPU = false);
    //Sends the CellEnergyArr to the GPU (and optionally clears it from the CPU)
    void returnData(const bool clear_GPU = false);
    //Returns the ClusterInfoArr to the CPU (and optionally clears it from the GPU)

    void prepData(SG::ReadHandle<CaloCellContainer>& pCaloCellContainer);
    //Fills the CellEnergyArr in CPU memory with the relevant values.

    Helpers::CPU_object<CellEnergyArr> m_cell_energy;
    Helpers::CPU_object<ClusterInfoArr> m_clusters;

    Helpers::CUDA_object<CellEnergyArr> m_cell_energy_dev;
    Helpers::CUDA_object<CellStateArr> m_cell_state_dev;
    Helpers::CUDA_object<ClusterInfoArr> m_clusters_dev;

    Helpers::CUDA_object<ClusterMergeTable> m_temp_table_dev;
    //If using an explicit merging step.

    //Temporary required for the method.
    Helpers::CUDA_object<int> m_temp_count_dev;

    Helpers::CUDA_object<PairsArr> m_pairs_dev;
    //If using selective pairs.
};

```

One should note that `LArNeighbours::neighbourOption`, `CaloNoise` and `SG::ReadHandle <CaloCellContainer>` are the same Athena classes described in section 2.3.4.

A.2.2.1 Constant Data

```

struct GeometryArr
{
    int    caloSample[NCaloCells];
    //The ID of the calorimeter sample the cell belongs to.
    float  x[NCaloCells];
    float  y[NCaloCells];
    float  z[NCaloCells];

    //All of these four are currently unused,
    //but they may be helpful for future implementations.
}

```

```

float eta[NCaloCells];
float phi[NCaloCells];
int nNeighbours[NCaloCells];
//The number of neighbours of each cell.
int neighbours[NCaloCells][NMaxNeighbours];
};

struct CellNoiseArr
{
    float noise[NCaloCells];
};

```

A.2.2.2 Event Data

```

struct CellEnergyArr
{
    float energy[NCaloCells];
};

struct CellStateArr
{
    float cellSNR[NCaloCells];
    tag_type clusterTag[NCaloCells];
};

struct ClusterInfoArr
{
    int clusterNumber;
    int seedCellID[NMaxClusters];
    float seedCellSNR[NMaxClusters];
    float seedCellEta[NMaxClusters];
    float seedCellPhi[NMaxClusters];

    tag_type clusterTag[NMaxClusters];
    //This is only truly relevant for the first tagging strategy
    //and could be omitted if using the second strategy.

    float clusterEnergy[NMaxClusters];

    float clusterEt[NMaxClusters];
    //This will also be temporarily used to hold the sum of absolute energies of all the cells.

    float clusterEta[NMaxClusters];
    float clusterPhi[NMaxClusters];
};

struct ClusterMergeTable
{
    tag_type new_tag[NMaxClusters];
};

```

A.2.2.3 Data Whose Nature Depends on the Chosen Strategy

```
struct PairsArr
{
    int pairsNumber;
    //The total number of cell pairs that are being considered
    int cellID[NMaxPairs];
    int neighbourID[NMaxPairs];
};
```

A.3 Implementation

A.3.1 Preparing the Data

A.3.1.1 Constant Data Preparation

If using a fixed list of cell pairs for the cluster growing algorithm, the code to build it is as follows:

```
void prepPairs()
{
    int zeroer = 0;
    cudaMemcpy(&(m_pairs_dev->pairsNumber), &zeroer, sizeof(int), cudaMemcpyHostToDevice);
    const int block_size = 256;
    //We could try to find a more optimal block size,
    //but since this will be executed just once,
    //it's of little consequence.
    const int grid_size = int_ceil_div(NCaloCells, block_size);
    fixed_cell_pairs_kernel<<<grid_size, block_size>>>(m_pairs_dev, m_geometry_dev);
}

__global__ void fixed_cell_pairs_kernel(Helpers::CUDA_kernel_object<PairsArr> neighbour_pairs,
                                       const Helpers::CUDA_kernel_object<GeometryArr> geometry)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < NCaloCells)
    {
        const int num_neighs = geometry->nNeighbours[index];
        const int n = atomicAdd(&(neighbour_pairs->pairsNumber), nNeighbours);

        //atomicAdd returns the value before the addition;
        //thus, this will be the position after what was previously the last element.

        for (int i = 0; i < num_neighs; ++i)
        {
            neighbour_pairs->cellID[n + i] = index;
            neighbour_pairs->neighbourID[n + i] = geometry->neighbours[index][i];
        }
    }
}
```

A.3.1.2 Event Data Preparation

To zero-out the counts in the cluster properties and cell pairs arrays:

```
int zeroer = 0;
cudaMemcpy(&(m_clusters_dev->clusterNumber), &zeroer, sizeof(zeroer), cudaMemcpyHostToDevice);
cudaMemcpy(&(m_pairs_dev->pairsNumber), &zeroer, sizeof(zeroer), cudaMemcpyHostToDevice);
// If using selective pairs...
```

A.3.2 Signal-to-Noise Calculation

For shortness of notation, the parts of code between `/*(2)*/` are only present when the second tagging strategy is adopted, with the rest of the code being the same regardless of the tagging strategy.

```
//grid_size = int_ceil_div(NCaloCells, block_size);
__global__
void signal_to_noise_kernel( Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                             /*(2)*/ Helpers::CUDA_kernel_object<ClusterInfoArr> clusters_arr, /*(2)*/
                             const Helpers::CUDA_kernel_object<CellEnergyArr> energy_arr,
                             const Helpers::CUDA_kernel_object<CellNoiseArr> noise_arr,
                             /*(2)*/ const Helpers::CUDA_kernel_object<GeometryArr> geometry /*(2)*/ )
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < NCaloCells)
    {
        float sig_noise_ratio;
        const float cell_energy = energy_arr->energy[index];
        const float cell_noise = noise_arr->noise[index];
        if (cell_noise > 0.0f)
        {
            sig_noise_ratio = fabsf(cell_energy / cell_noise);
        }
        else
        {
            sig_noise_ratio = 0.00001f;
            // It's what's done in the CPU implementation in these cases.
        }
        state_arr->cellSNR[index] = sig_noise_ratio;
        if (sig_noise_ratio > SeedThreshold)
        {
            /*(2)*/ const int n = atomicAdd(&(clusters_arr->clusterNumber), 1); /*(2)*/

            const tag_type tag = calculate_tag(n, sig_noise_ratio);

            /*(2)*/ clusters_arr->seedCellID[n] = index; /*(2)*/
            /*(2)*/ clusters_arr->seedCellSNR[n] = sig_noise_ratio; /*(2)*/
            /*(2)*/ clusters_arr->seedCellEta[n] = geometry->eta[index]; /*(2)*/
            /*(2)*/ clusters_arr->seedCellPhi[n] = geometry->phi[index]; /*(2)*/
            /*(2)*/ clusters_arr->clusterEnergy[n] = 0.f; /*(2)*/
        }
    }
}
```

```

/* (2) */ clusters_arr->clusterEt[n] = 0.f;          /* (2) */
/* (2) */ clusters_arr->clusterEta[n] = 0.f;         /* (2) */
/* (2) */ clusters_arr->clusterPhi[n] = 0.f;         /* (2) */

    state_arr->clusterTag[index] = tag;
}
else if (sig_noise_ratio > GrowThreshold)
{
    state_arr->clusterTag[index] = GrowTag;
}
else if (sig_noise_ratio > CellThreshold)
{
    state_arr->clusterTag[index] = TerminalTag;
}
else
{
    state_arr->clusterTag[index] = InvalidTag;
}
}
}

```

Where calculate_tag is defined according to adopted tagging strategy:

```

//First Tagging Stragey:
__device__ inline
int calculate_tag_1(const int index, const float SNR)
{
    return __float_as_int((SNR/SeedThreshold) * NCaloCells + index);
    //CUDA intrinsic that does the bit reinterpretation in a typesafe way.
}

//Second Tagging Strategy:
__device__ inline
unsigned long long int calculate_tag_2(const int index, const float SNR)
{
    unsigned long long int ret = __float_as_int(SNR);
    ret = (ret << 32) | index;
    return ret;
}

```

A.3.3 Pair Creation

Restating for clarity's sake, this step is only needed if adopting the selective pair creation strategy.

```

//grid_size = int_ceil_div(NCaloCells, block_size);
__global__
void cell_pairs_kernel( Helpers::CUDA_kernel_object<PairsArr> neighbour_pairs,
                        const Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                        const Helpers::CUDA_kernel_object<GeometryArr> geometry)
{

```

```

const int index = blockIdx.x * blockDim.x + threadIdx.x;
if (index < NCaloCells)
{
    if (state_arr->cellSNR[index] > GrowThreshold)
    {
        const int num_neighs = geometry->nNeighbours[index];
        int neigh_list[NMaxNeighbours];
        int num_bad_neighs = 0;
        //We will exclude invalid cells.
        for (int i = 0; i < nNeighb; ++i)
        {
            const int neigh_id = geometry->neighbours[index][i];
            if (state_arr->cellSNR[neigh_id] > CellThreshold)
            {
                neigh_list[i - num_bad_neighs] = neigh_id;
            }
            else
            {
                ++num_bad_neighs;
            }
        }
        const int real_neighbours = num_neighs - num_bad_neighs;
        const int n = atomicAdd(&(neighbour_pairs->pairsNumber), real_neighbours);
        for (int i = 0; i < real_neighbours; ++i)
        {
            neighbour_pairs->cellID[n + i] = index;
            neighbour_pairs->neighbourID[n + i] = neigh_list[i];
        }
    }
}
}

```

A.3.4 Cluster Growing

A.3.4.1 Synchronization Kernel

For shortness of notation, let `/*(M)*/` mark portions of the code that are used when a growing strategy with merge has been adopted, let `/*(I)*/` mark portions of the code that are used with implicit pairs and `/*(E)*/` mark code that is used with explicit (that is, selective or fixed) pairs. Unmarked code will be common to all strategies.

```

//block_size = 1;
//grid_size = 1;
__global__
void cluster_growing_kernel(Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                            Helpers::CUDA_kernel_object<int> * flag,
                            /*(E)*/ const Helpers::CUDA_kernel_object<PairsArr> neighbour_pairs, /*(E)*/
                            /*(I)*/ const Helpers::CUDA_kernel_object<GeometryArr> geometry, /*(I)*/
                            /*(M)*/ Helpers::CUDA_kernel_object<ClusterMergeTable> table /*(M)*/)

```

```

{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index == 0)
    {
        const int pairs_number = neighbour_pairs->pairsNumber;
        //Grid calculations , if necessary...
        *flag = 1;
        while(*flag > 0)
        {
            *flag = 0;
            propagate_neighbours<<<<dimGrid1, dimBlock1>>>>(state_arr, flag,
                                                            /*(M)*/ table, /*(M)*/
                                                            /*(I)*/ geometry, /*(I)*/
                                                            /*(E)*/ pairs_number, neighbour_pairs /*(E)*/);

/*(M)*/ merge_clusters<<<<dimGrid2, dimBlock2>>>>(state_arr, table); /*(M)*/

            if(*flag == 0)
            {
                cudaDeviceSynchronize();
                // If flag is already non-zero,
                //we can go ahead and launch another kernel
                //without needing to wait for the current to finish.
            }
        }

        propagate_terminal<<<<dimGrid3, dimBlock3>>>>(state_arr,
                                                        /*(I)*/ geometry, /*(I)*/
                                                        /*(E)*/ pairs_number, neighbour_pairs /*(E)*/);
    }
}

```

A.3.4.2 Strategies With Selective or Fixed Pairs

For shortness of notation, let /*(M)*/ mark portions of the code that are used when we use a merge step, /*(F)*/ mark portions that are used with fixed pairs and /*(S)*/ mark portions that are used with selective pairs. Unmarked code will be common to all strategies.

```

//grid_size = int_ceil_div(pairs_number, block_size);
__global__
void propagate_neighbours(Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                          Helpers::CUDA_kernel_object<int> flag,
                          /*(M)*/ Helpers::CUDA_kernel_object<ClusterMergeTable> table, /*(M)*/
                          const int pair_number,
                          const Helpers::CUDA_kernel_object<PairsArr> neighbour_pairs)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < pair_number)
    {

```

```

    const int this_ID = neighbour_pairs->cellID[index];
    const int neigh_ID = neighbour_pairs->neighbourID[index];
    const tag_type neigh_tag = state_arr->clusterTag[neigh_ID];

/*(S)*/ const tag_type old_tag = atomicMax(&(state_arr->clusterTag[this_ID]), neigh_tag); /*(S)*/

/*(F)*/ const tag_type old_tag = state_arr->clusterTag[this_ID]; /*(F)*/
/*(F)*/ if (old_tag >= GrowTag && neigh_tag > old_tag) /*(F)*/
/*(F)*/ { /*(F)*/
/*(F)*/     atomicMax(&(state_arr->clusterTag[this_ID]), neigh_tag); /*(F)*/

        if (neigh_tag > old_tag)
        {
            atomicMax(flag, 1);

/*(M)*/         if (old_tag > GrowTag) /*(M)*/
/*(M)*/         // If it was part of a cluster already /*(M)*/
/*(M)*/         { /*(M)*/
/*(M)*/             const unsigned int address = old_tag; /*(M)*/
/*(M)*/             //This truncates the tag to the lowest 32 bits, /*(M)*/
/*(M)*/             //which correspond to the index into the table that we want. /*(M)*/
/*(M)*/             atomicMax(&(table->new_tag[address]), neigh_tag); /*(M)*/
/*(M)*/         } /*(M)*/

        }

/*(F)*/ } /*(F)*/

}

//grid_size = int.ceil_div(pairs_number, block_size);
__global__
void propagate_terminal(Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                       const int pairs_number,
                       const Helpers::CUDA_kernel_object<PairsArr> neighbour_pairs)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < pairs_number)
    {
        const int this_ID = neighbour_pairs->cellID[index];
        const int neigh_ID = neighbour_pairs->neighbourID[index];

/*(F)*/ const float neigh_SNR = state_arr->cellSNR[neigh_ID]; /*(F)*/
/*(F)*/ if (neigh_SNR > GrowThreshold) /*(F)*/
/*(F)*/ { /*(F)*/
/*(F)*/     const float this_SNR = state_arr->cellSNR[this_ID]; /*(F)*/
/*(F)*/     if (this_SNR > CellThreshold) /*(F)*/
/*(F)*/     { /*(F)*/

```

```

        atomicMax(&(state_arr->clusterTag[neigh_ID]), state_arr->clusterTag[this_ID]);

/*(F)*/    }                                /*(F)*/
/*(F)*/    }                                /*(F)*/

    }
}

```

A.3.4.3 Strategies With Implicit Pairs

For shortness of notation, let */*(M)*/* mark portions of the code that are used when we use a merge step, */*(I)*/* mark portions that use implicit shuffles to perform warp-based reductions and */*(E)*/* mark portions that are needed to perform the warp-based reduction explicitly. Unmarked code will be common to all strategies.

```

//grid_size = int_ceil_div(NCaloCells, block_size / WarpSize);
__global__
void propagate_neighbours(Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                          Helpers::CUDA_kernel_object<int> changed_flag,
                          /*(M)*/ Helpers::CUDA_kernel_object<ClusterMergeTable> table, /*(M)*/
                          const Helpers::CUDA_kernel_object<GeometryArr> geometry)
{
    /*(E)*/ __shared__ tag_type temps[block_size/WarpSize][WarpSize]; /*(E)*/

    const int warp_index = threadIdx.x / WarpSize;
    const int thread_index = threadIdx.x % WarpSize;
    const int accessed_neighbour = thread_index;
    const int accessed_cell = warp_index + blockIdx.x * blockDim.x / WarpSize;

    if (accessed_cell < NCaloCells)
    {
        const tag_type this_tag = state_arr->clusterTag[accessed_cell];
        if (this_tag >= GrowTag)
            //We are interested in propagating through growing cells.
            {
                const int num_neighs = geometry->nNeighbours[accessed_cell];
                tag_type new_tag = this_tag;
                if (num_neighs >= 1)
                {
                    if (accessed_neighbour < num_neighs)
                    {
                        const int neigh_index = geometry->neighbours[accessed_cell][accessed_neighbour];
                        new_tag = max(this_tag, state_arr->clusterTag[neigh_index]);
                    }
                    if (num_neighs > 1)
                    {
                        warp_based_reduce_max(new_tag, accessed_neighbour, num_neighs, /*(E)*/ temps[warp_index] /*(E)*/);
                    }
                    if (thread_index == 0)
                    {

```

```

        if (new_tag > this_tag)
        {
            state_arr->clusterTag[accessed_cell] = new_tag;
            atomicMax(changed_flag, 1);
        }

/*(M)*/         if (this_tag > GrowTag)                                     /*(M)*/
/*(M)*/         {                                                         /*(M)*/
/*(M)*/             const int address = ((unsigned int) this_tag);         /*(M)*/
/*(M)*/             atomicMax(&(table->new_tag[address]), new_tag);         /*(M)*/
/*(M)*/         }                                                         /*(M)*/

    }
}
}
}
}

//grid_size = int_ceil_div(NCaloCells, block_size / WarpSize);
__global__
void propagate_terminal(Helpers::CUDA_kernel_object<CellStateArr> state_arr ,
                       const Helpers::CUDA_kernel_object<GeometryArr> geometry)
{
    /*(E)*/ __shared__ tag_type temps[block_size/WarpSize][WarpSize]; /*(E)*/

    const int warp_index = threadIdx.x / WarpSize;
    const int thread_index = threadIdx.x % WarpSize;
    const int accessed_neighbour = thread_index;
    const int accessed_cell = warp_index + blockIdx.x * blockDim.x / WarpSize;
    if (accessed_cell < NCaloCells)
    {
        const tag_type this_tag = state_arr->clusterTag[accessed_cell];
        if (this_tag == TerminalTag)
        {
            const int num_neighs = geometry->nNeighbours[accessed_cell];
            tag_type new_tag = this_tag;
            if (num_neighs >= 1)
            {
                if (accessed_neighbour < num_neighs)
                {
                    const int neigh_index = geometry->neighbours[accessed_cell][accessed_neighbour];
                    if (state_arr->cellSNR[neigh_index] > GrowThreshold)
                    {
                        new_tag = max(state_arr->clusterTag[neigh_index], this_tag);
                    }
                }
            }
            if (num_neighs > 1)
            {
                warp_based_reduce_max(new_tag, accessed_neighbour, num_neighs, /*(E)*/ temps[warp_index] /*(E)*/ );
            }
        }
    }
}

```



```

void warp_based_reduce_max_explicit(tag_type & new_tag,
                                   const int thread_id,
                                   const int max_id,
                                   volatile tag_type *array)
//array will be a pointer to a tag_type[WarpSize] array held in the shared memory
{
    array[thread_id] = new_tag;
    const int pow2_max = 1 << (32 - __clz (max_id - 1));
    //The same as before.

    __syncwarp();

    for (int i = pow2_max/2; i > 0; i /= 2)
    {
        const tag_type old_low = array[thread_id];
        const tag_type old_high = (thread_id + i < WarpSize ? array[thread_id + i] : 0);

        __syncwarp();

        array[thread_id] = max(old_low, old_high);

        __syncwarp();
    }
    new_tag = array[thread_id];
}

```

For completeness' sake, the fully intrinsic warp-based reduction that `__reduce_max_sync` would make possible could be written as:

```

__device__ inline
void warp_based_reduce_max(tag_type & new_tag,
                           const int thread_id,
                           const int max_id)
{
    unsigned int mask = __ballot_sync(0xFFFFFFFF, thread_id < max_id);
    //This is to leave out all the threads that do not contain useful values.
    new_tag = __reduce_max_sync(mask, new_tag);
}

```

A.3.4.4 Cluster Merging

For the strategies that include an explicit cluster merging step (and thus use the second tagging strategy), that step will be:

```

//grid_size = int.ceil_div(NCaloCells, block_size);
__global__
void merge_clusters(Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                   const Helpers::CUDA_kernel_object<ClusterMergeTable> table)
{

```

```

const int index = blockIdx.x * blockDim.x + threadIdx.x;
if (index < NCaloCells)
{
    const tag_type old_tag = state_arr->clusterTag[index];
    if (old_tag > GrowTag)
    {
        const int address = ((unsigned int) old_tag);
        state_arr->clusterTag[index] = table->new_tag[address];
    }
}
}

```

A.3.5 Final Cluster Properties Calculation

A.3.5.1 First Tagging Strategy Cluster Finding

As the name implies, this is only necessary for the first tagging strategy, as the protocluster array has not been created so far.

```

//grid_size = int_ceil_div(NCaloCells, block_size);
__global__
void find_clusters_kernel(Helpers::CUDA_kernel_object<ClusterInfoArr> clusters_arr,
                        const Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                        const Helpers::CUDA_kernel_object<GeometryArr> geometry)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < NCaloCells)
    {
        const float cell_SNR = state_arr->cellSNR[index];
        if (cell_SNR > SeedThreshold)
        {
            const tag_type cellTag = calculateTag(index, cell_SNR);
            // If the tags match, then this is the seed cell of the protocluster.

            if (state_arr->clusterTag[index] == cellTag)
            {
                const int n = atomicAdd(&(clusters_arr->clusterNumber), 1);
                clusters_arr->seedCellID[n] = index;
                clusters_arr->seedCellSNR[n] = cell_SNR;
                clusters_arr->seedCellEta[n] = geometry->eta[index];
                clusters_arr->seedCellPhi[n] = geometry->phi[index];
                clusters_arr->clusterTag[n] = cellTag;
                clusters_arr->clusterEnergy[n] = 0.f;
                clusters_arr->clusterEt[n] = 0.f;
                clusters_arr->clusterEta[n] = 0.f;
                clusters_arr->clusterPhi[n] = 0.f;
            }
        }
    }
}

```

A.3.5.2 Cell Contributions Calculation

For shortness of notation, the parts of code between `/*(1)*/` are only present when the first tagging strategy is adopted and those between `/*(2)*/` are only present when the second tagging strategy is adopted, with the rest of the code being the same regardless of the tagging strategy.

```
//grid_size = int.ceil_div(NCaloCells, block_size);
__global__
void calculate_cell_contrib(Helpers::CUDA_kernel_object<ClusterInfoArr> clusters_arr,
                           const Helpers::CUDA_kernel_object<CellStateArr> state_arr,
                           const Helpers::CUDA_kernel_object<CellEnergyArr> energy_arr,
                           const Helpers::CUDA_kernel_object<GeometryArr> geometry)
{
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < NCaloCells)
    {
        const tag_type cell_tag = state_arr->clusterTag[index];
        if (cell_tag > GrowTag)
            //This means the cell belongs to a valid cluster.
            {

/*(1)*/ int cluster_index = -1;                                /*(1)*/
/*(1)*/ const int cluster_number = clusters_arr->clusterNumber; /*(1)*/
/*(1)*/ for (int i = 0; i < cluster_number; ++i)                /*(1)*/
/*(1)*/ {                                                        /*(1)*/
/*(1)*/     if (clusters_arr->clusterTag[i] == cell_tag)         /*(1)*/
/*(1)*/     {                                                  /*(1)*/
/*(1)*/         cluster_index = i;                               /*(1)*/
/*(1)*/         break;                                           /*(1)*/
/*(1)*/     }                                                  /*(1)*/
/*(1)*/ }                                                        /*(1)*/
/*(1)*/ if (cluster_index >= 0)                                  /*(1)*/
/*(1)*/ {                                                        /*(1)*/

/*(2)*/ const unsigned int cluster_index = cell_tag;            /*(2)*/
/*(2)*/ //This truncates the tag to the lowest 32 bits,         /*(2)*/
/*(2)*/ //which correspond to the index into the array.         /*(2)*/

        const float energy = energy_arr->energy[index];
        const float abs_energy = fabsf(energy);
        const float phi_raw = geometry->phi[index];
        atomicAdd(&(clusters_arr->clusterEnergy[cluster_index]), energy);

        atomicAdd(&(clusters_arr->clusterEt[cluster_index]), abs.energy);
        //This is just temporary, will be updated with the correct value in the final step.

        atomicAdd(&(clusters_arr->clusterEta[cluster_index]), abs.energy * geometry->eta[index]);
        const float phi_0 = clusters_arr->seedCellPhi[cluster_index];
        const float phi_real = phi_raw +
            ( phi_raw > phi_0 + pi<float> ? -2 :
```

```

        phi_raw < phi_0 - pi<float> ? 2 : 0 ) * pi<float>;
        atomicAdd(&(clusters_arr->clusterPhi[cluster_index]), phi_real * abs_energy);

/* (1) */ }                                     /* (1) */

    }
}
}

```

A.3.5.3 Cluster Properties Finalization

```

//grid_size = int_ceil_div(clusters_arr->clusterNumber, block_size)
__global__
void finalize_cluster_info_kernel(Helpers::CUDA_kernel_object<ClusterInfoArr> clusters_arr)
{
    const int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < clusters_arr->clusterNumber)
    {
        const float abs_energy = clusters_arr->clusterEt[i];
        //This was the temporary sum of all abs(energies)
        if(abs_energy > 0)
        {
            const float temp_eta = clusters_arr->clusterEta[i] / abs_energy;
            clusters_arr->clusterEta[i] = temp_eta;
            clusters_arr->clusterPhi[i] /= abs_energy;

            clusters_arr->clusterEt[i] = abs_energy / coshf(temp_eta);
            //This sets it to the transverse energy, as desired.

        }
        else
        {
            clusters_arr->seedCellSNR[i] = CellThreshold;
            //This is just a way to signal that this is an invalid cluster.
            //It's only truly necessary in the second tagging strategy,
            //since, for the first, only valid clusters will be part of the list.
        }
    }
}

```