Acceleration of the ATLAS-CERN Calorimeter Data Preparation with GPUs

Gabriela Paola Sereniski

Dissertation presented to the School of Technology and Management of Bragança to obtain the master's degree in Informatics within the scope of the double degree program with the Federal University of Technology – Paraná.

Supervisors:

Patricia Conde Muiño (IST)

José Carlos Rufino Amaro (IPB)

Rogério Aparecido Gonçalves (UTFPR)

Bragança

November 2025

Acceleration of the ATLAS-CERN Calorimeter Data Preparation with GPUs

Gabriela Paola Sereniski

Dissertation presented to the School of Technology and Management of Bragança to obtain the master's degree in Informatics within the scope of the double degree program with the Federal University of Technology – Paraná.

Supervisors:

Patricia Conde Muiño (IST)

José Carlos Rufino Amaro (IPB)

Rogério Aparecido Gonçalves (UTFPR)

Bragança

November 2025

Dedicated to my mother.

Acknowledgments

This work was supported by LIP - Laboratorio de Investigacão e Física Experimental de Partículas. The development of this work was only viable with the computing resources provided by CERN and with the extensive code documentation made available by the ATLAS Collaboration.

I would like to express my genuine gratitude for the continuous support, patience, understanding, and trust that my supervisors Patricia Conde Muiño, José Carlos Rufino Amaro, and Rogério Aparecido Gonçalves, have given me. I ought to also thank Professor André Martins Pereira, for the comprehensive review of this thesis and for the insightful feedback provided. Professors, thank you for accepting and inspiring me.

Although not an official supervisor, Professor Denis Oliveira Damazio was the person who guided me through the mazes of the Athena code. Thanks to Denis' will to help, availability, ATLAS expertise, and patience, I was able to truly understand something about what I was doing, instead of blindly developing code for a subject I am only now managing to grasp the surface of.

I must also acknowledge the help, friendly discussions, and advice offered by Nuno dos Santos Fernandes. Without his work to base mine (and without the first algorithm skeleton he so kindly prepared, so that I would have something to work with when I was the most lost), this work would certainly not have reached the state it is in today.

It would be unfair not to thank the friends that Bragança managed to gift me in such a short period of time. Without the support, experiences, laughter, and brain cells shared with Luana and Luisa Reck, my time in Portugal would have been much, much duller. Thank you for welcoming me and making me feel like I belong. I would also like to thank the tuxedo, sock-wearing cat that mewed at me from a random alley. Without following it, I would not have met Gabriel Fonseca Oliveira Roma, who, in so little time, managed to help me in ways that I cannot begin to describe, despite the chaos of everything.

E o mais importante, preciso agradecer à minha família. Sem o suporte dos meus pais, eu não estaria aqui. Obrigada por sempre me incentivarem e por não me deixarem desistir. Obrigada por serem tão compreensivos e gentis, de um jeito ou de outro. Obrigada, mãe, por todo o esforço que fez para que conseguisse me proporcionar o mundo. Você conseguiu.

Abstract

The High-Luminosity Large Hadron Collider (HL-LHC) will deliver substantially higher numbers of simultaneous proton-proton collisions (events), increasing the volume and frequency of detector readouts and placing growing demands on the ATLAS calorimeter (subdetectors that measure energy deposits from particle showers) data-preparation pipeline. Bytestream decoding is the first step of this pipeline, responsible for transforming the raw output of the Liquid Argon (LAr) and Tile (TileCal) calorimeters into physics-ready cells. Due to its sequential nature, it may become a significant bottleneck. At the same time, Graphics Processing Units (GPUs) have become increasingly relevant in high-throughput scientific computing, offering massive parallelism and high-memory bandwidth. This dissertation investigates the feasibility of offloading the calorimeter bytestream decoding to GPUs and evaluates the performance of a prototype implementation integrated within the AthenaMT framework. A complete GPU-based pipeline was developed, including bytestream staging, device transfers, parallel fragment parsing and decoding for both LAr and TileCal. The implementation follows a modular `CUDA C++` design tailored for parallel scalability. Validation against the CPU reference showed bit-identical agreement across all tested events. Performance measurements using Run-3 data demonstrated an order-of-magnitude improvement: while CPU decoding of a full calorimeter event takes on average 21.1 ms, the GPU prototype achieves 1.30 ms in synchronous and 1.08 ms in asynchronous mode on a constrained A100 MIG slice. These results confirm that massively parallel architectures can effectively accelerate calorimeter data preparation and motivate future accelerator-aware redesigns of the bytestream format for the HL-LHC.

Resumo

O *High-Luminosity Large Hadron Collider* (HL-LHC) suportará números substancialmente maiores de colisões protão-protão (eventos) simultâneas, o que aumentará o volume e a frequência das leituras do detetor e imporá maiores exigências à cadeia de preparação de dados dos calorímetros do ATLAS, os subdetetores que medem depósitos de energia provenientes de chuvas de partículas. A descodificação do *bytestream*, responsável por transformar os dados brutos dos calorímetros de Árgon Líquido (LAr) e Tile (TileCal) em células prontas para análise física, é o primeiro passo desta cadeia. Devido ao seu carácter sequencial, este passo poderá tornar-se um gargalo significativo. Em paralelo, as unidades de processamento gráfico (GPUs) têm ganho relevância na computação científica de alto desempenho, por oferecer paralelismo massivo e grande largura de banda de memória. Esta dissertação investiga a viabilidade de descarregar para GPUs a descodificação do *bytestream* e avalia o desempenho de um protótipo integrado no *framework* AthenaMT. Foi desenvolvido um pipeline totalmente baseado em GPU, com a preparação do dos dados, transferência para a GPU, *parsing* de fragmentos e descodificação paralela para ambos os calorímetros do ATLAS. A implementação segue um modelo modular em `CUDA C++` orientado para a escalabilidade paralela. A validação contra a referência em CPU mostrou concordância bit-a-bit em todos os eventos testados. As medições de desempenho com dados do *Run*-3 demonstraram uma melhoria de uma ordem de grandeza: enquanto a descodificação em CPU de um evento completo do calorímetro demora, em média, 21.1 ms, o protótipo em GPU atinge 1.30 ms em modo síncrono e 1.08 ms em modo assíncrono numa partição A100 MIG. Estes resultados confirmam que arquiteturas paralelas podem acelerar de forma eficaz a preparação de dados dos calorímetros e motivam futuras reformulações do formato de *bytestream* para o HL-LHC orientadas a aceleradores.

Contents

List of Figures

Acronyms

ACTS A Common Tracking Software

ADC analog-to-digital converter

ALICE A Large Ion Collider Experiment

ALU Arithmetic Logic Unit

AoS arrays of structures

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

AthenaMT Athena Multithreaded

ATLAS A Toroidal LHC Apparatus

BCID bunch-crossing identifier

CA `ComponentAccumulator`

CERN The European Organization for Nuclear Research

CMS Compact Muon Selonoid

CPU Central Processing Unit

CRC Cyclic Redundancy Check

`CUDA` Compute Unified Device Architecture

DAQ Data Acquisition

DCM Data Collection Manager

DMA Direct Memory Access

DMU Data Management Unit

DQ data quality

DRAM Dynamic Random-Access Memory

DSMEM Distributed Shared Memory

DSP Digital Signal Processors

EF Event Filter

FEB front-end board

FEE front-end electronics

FPGA Field-Programmable Gate Array

GPGPU General-Purpose Computing Graphical Processing Unit

GPU Graphical Processing Unit

HBM2 High Bandwidth Memory 2

HEP High-Energy Physics

`HIP` Heterogeneous-compute Interface for Portability

HL-LHC High Luminosity Large Hadron Collider

HLT High-Level Trigger

ID identifier

L0 Level-0

L1 Level-1

L2 Level-2

LAr Liquid Argon

LArCal LAr Calorimeter

LB luminosity block

LBA Long Barrel A-Side

LBC Long Barrel C-Side

LHC Large Hadron Collider

LHCb Large Hadron Collider beauty

LUT look-up table

MBTS Minimum Bias Trigger Scintillators

MIG multi-instance GPU

OC offset correction

`OpenCL` Open Computing Language

PCIe Peripheral Component Interconnect Express

PMT photomultiplier tube

pp proton-proton

RAM Random-Access Memory

ROB read-out buffer

ROCm Radeon Open Compute Platform

ROD read-out driver

RoIB region-of-interest builder

ROS read-out sub-system

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Thread

SM Streaming Multiprocessor

SoA structure of arrays

SRAM Static Random-Access Memory

TBB Threading Building Blocks

TBC Thread Block Cluster

TDAQ Trigger and Data Acquisition

TileCal Tile Calorimeter

TLB Translation Lookaside Buffer

TQ time and quality

ULP units-in-the-last-place

WLCG Worldwide LHC Computing Grid

Chapter 1

Introduction

The Large Hadron Collider (LHC) at The European Organization for Nuclear Research (CERN) delivers proton-proton (pp) collisions at unprecedented rates and energies, enabling the exploration of rare processes and precision measurements across a wide physics programme. Within the A Toroidal LHC Apparatus (ATLAS) experiment, the calorimeter system plays a central role in reconstructing jets, photons, electrons, and missing transverse energy, all of which require fast and accurate processing of detector signals.

The calorimeter system is the sub-detector that measures the energy deposited by particles generated by the collisions. Particles transverse a finely segmented layer of absorber and active material, producing electromagnetic or hadronic showers whose signals are sampled, digitised, and transmitted as a stream of encoded binary data – the bytestream. Subsequently, these samples are decoded and reconstructed into calorimeter cells, which provide the primary inputs for many physics objects used in ATLAS reconstruction.

As luminosity increases throughout Run-3 and toward the High Luminosity Large Hadron Collider (HL-LHC), the data throughput and event complexity will grow substantially, placing considerable performance demands on the software systems responsible for real-time and offline data preparation. Among these, the calorimeter bytestream decoding step represents the earliest point at which raw calorimeter readouts become accessible to the reconstruction chain. Its efficiency and scalability therefore have a direct impact on the latency and resource usage of the High-Level Trigger (HLT) and offline workflows.

The calorimeter decoding algorithms in ATLAS that are currently in use were designed more than two decades ago, when Central Processing Unit (CPU) architectures and event characteristics differed significantly from those of today and from those expected at the HL-LHC. Although well validated and operationally robust, these algorithms are inherently sequential and feature CPU-centric data layouts. This generates irregular memory access patterns that do not map naturally onto heterogeneous computing platforms.

As accelerator technologies, such as Graphical Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs), become increasingly prominent in High-Energy Physics (HEP) and in ATLAS software modernisation efforts, it is essential to understand whether the calorimeter bytestream decoding stage can benefit from massively parallel execution and how such a redesign could integrate with the existing framework. This dissertation addresses these questions by developing and evaluating a GPU-based prototype of the Liquid Argon (LAr) and Tile Calorimeter (TileCal) bytestream decoders, serving as a feasibility study for future heterogeneous architectures.

1.1 Motivation

The motivation for this work arises from the convergence of two trends: the increasing data volume delivered by the LHC and the growing adoption of heterogeneous computing in large-scale scientific experiments. The HL-LHC upgrade will raise the instantaneous luminosity of the collider by an order of magnitude, leading to a higher number of collisions happening at the same time (pile-up), larger bytestream payloads and, consequently, stricter real-time constraints on the HLT. On the software side, GPUs have become more prominent in contemporary high-performance computing due to their capacity for massive parallelism, high memory-bandwidth, and improved energy efficiency. Several reconstruction and simulation workflows in ATLAS and other LHC experiments have already demonstrated significant performance gains when ported to GPU-accelerated back-ends.

In the context of calorimeter data preparation, offloading the decoding stage to GPUs offers two major potential advantages. Firstly, the decoding logic for individual channels

(the digitised readout units that compose the calorimeter cells) is largely uniform and independent across and between fragments (the blocks of bytestream data produced by the calorimeter's readout modules), making it a naturally parallelizable process.

Secondly, performing decoding directly on the device allows the resulting calorimeter cell data to remain in GPU memory, avoiding repeated transfers between host and device and enabling downstream algorithms (such as clustering or topological selections) to operate without additional overhead. This could significantly reduce both latency and data movement in future trigger and offline pipelines. Exploring these possibilities requires a realistic prototype, integrated with the Athena framework, that can assess performance, correctness, and compatibility with current data formats.

1.2 Objectives

The general goal of this dissertation is to investigate the feasibility and performance of a GPU-based decoding of the ATLAS calorimeter bytestream. This involves designing, implementing and validating a prototype decoder capable of handling the LAr and TileCal calorimeter payloads in a manner consistent with the established CPU algorithms. To accomplish this, it is necessary to:

- Implement a complete GPU decoding pipeline within the Athena framework, including bytestream staging, device transfers, fragment parsing, channel reconstruction and cell finalisation;
- Ensure functional equivalence with the current CPU-based decoders, validated through detailed numerical comparison of all unpacked bytestream fields;
- Measure and characterise the performance of the GPU decoders under realistic conditions using Run-3 data, quantifying latency improvements and identifying bottlenecks;
- Assess the impact of execution models, memory allocation strategies and concurrency on overall performance;

- Explore the implications of GPU-based decoding for downstream reconstruction, particularly the potential for GPU-resident calorimeter data structures and future heterogeneous workflows;
- Provide insights that can inform the redesign of the calorimeter bytestream format and pipeline for the HL-LHC, where massively parallel architectures may become the dominant compute platform.

1.3 Dissertation Outline

The remainder of this dissertation is structured as follows:

- Chapter 2 introduces the scientific context of HEP, the ATLAS experiment and its trigger and data acquisition systems.
- Chapter 3 presents the technical background on the current bytestream format and its decoding algorithms. It also covers heterogeneous computing concepts, such as the use of GPUs as accelerators, the fundamentals of Compute Unified Device Architecture (`CUDA`), and the state-of-the-art use of accelerators in HEP.
- Chapter 4 details the design and implementation of the GPU-based decoders for the LAr and TileCal calorimeters, including their data structures, GPU functions and integration with Athena Multithreaded (AthenaMT).
- Chapter 5 reports the experimental setup, test dataset characteristics, benchmarking methodology and performance evaluation of the prototypes under various configurations.
- Finally, Chapter 6 discusses the results, outlines the main conclusions of the work and highlights directions for future developments of the GPU-based data preparation in the HL-LHC era.

Chapter 2

Scientific Context

This chapter establishes the physics and experimental context for the work presented in this dissertation. It begins with an overview of HEP and the Standard Model, motivating open questions that drive searches for physics beyond it. It then introduces CERN and the LHC, followed by a focused description of the ATLAS experiment, with emphasis on the calorimeter systems that produce the bytestream data decoded in this work. The Trigger and Data Acquisition (TDAQ) system is summarized to explain how raw fragments flow from the detector to the software chain, and the upcoming HL-LHC upgrade is discussed. Details on the Athena software framework are also presented.

2.1 High-Energy Physics

HEP, also known as Particle Physics, is a field of science dedicated to understanding the fundamental constituents of matter and their interactions. The theoretical framework in which the field is based is known as the Standard Model of particle physics. This model provides precise descriptions of the elementary particles and three of the four fundamental forces of nature.

2.1.1 The Standard Model

According to the Standard Model, all ordinary matter is composed of a small number of elementary particles called fermions, which have half-integer spin. They are grouped into two families — quarks and leptons — each consisting of six types or "flavors". Every fermion has a corresponding antiparticle with the same mass and opposite charge [1].

These particles are further arranged into three generations of increasing mass: the first generation (up and down quarks, electron, and electron neutrino) forms all stable matter of the universe, while the heavier generations are unstable and rapidly decay into lighter particles [2]. The full set of known fermions is summarized in Figure 2.1.

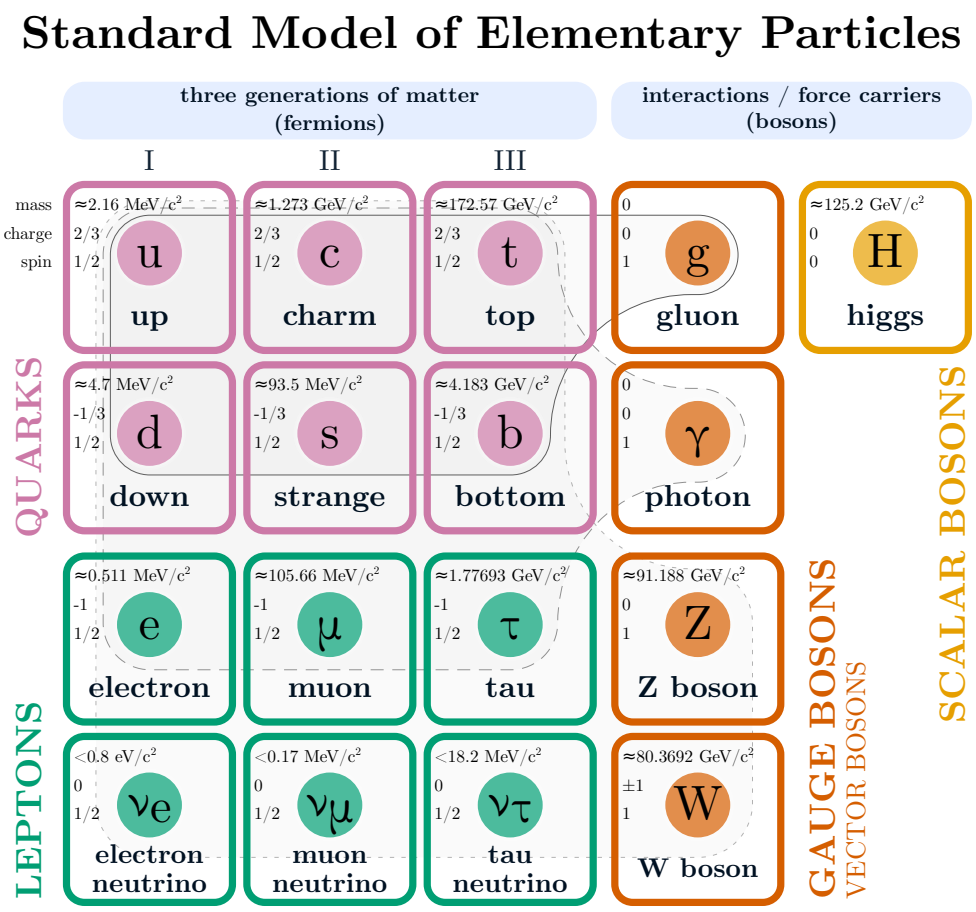

Figure 2.1: Standard model of elementary particles: the 12 fundamental fermions and 5 fundamental bosons. Grey loops indicate which bosons couple to which fermions. Source: Cush [3] (Adapted).

Interactions between matter particles are mediated by another class of fundamental particles, the gauge bosons, which carry the forces of nature. Bosons are also summarized in Figure 2.1. The electromagnetic force is mediated by the photon (γ), a massless particle that couples to electric charge and governs phenomena ranging from atomic binding to visible light [1].

The strong nuclear force is mediated by gluons (g). It is what binds quarks into hadrons and holds protons and neutrons together inside the atomic nuclei. Unlike the electromagnetic force, the strong force becomes stronger as quarks are pulled apart, a behaviour that leads to "confinement": quarks are never observed in isolation, but always bound together inside composite particles [4].

The weak nuclear force is carried by three massive bosons: W^+, W^-, and Z^0. It is what allows quarks to transform into one another and governs processes such as beta decay. Because the W and Z bosons are heavy, the weak interaction is very short-ranged, which explains why it appears so feeble compared to electromagnetism [2].

At high energies, electromagnetism and the weak interaction merge into a single force described by the electroweak theory [5]. However, this framework predicted massless force carriers, contradicting the observed heavy W and Z bosons. The Brout-Englert-Higgs mechanism resolves this by introducing a scalar field that fills all of space [6, 7]. When this field — the Higgs field — acquires a non-zero vacuum expectation value, it breaks the electroweak symmetry and gives mass to the W and Z bosons while leaving the photon massless.

The excitation of the Higgs field corresponds to the Higgs boson. Its discovery in 2012, by the ATLAS and Compact Muon Selonoid (CMS) experiments at the LHC [8, 9], confirmed the final missing piece of the Standard Model and provided direct evidence for the mechanism through which elementary particles acquire mass.

2.1.2 Physics beyond the Standard Model

Despite its success, the Standard Model does not answer all of particle physics conundrums. The most prominent open question is the hierarchy problem: quantum corrections should drive the Higgs mass to the Planck scale (10^{19} GeV), where gravity becomes strong. However, its observed mass was of 125 GeV. The fact that it is so light requires a seemingly unnatural fine-tuning of the model's parameters [10]. Theories such as supersymmetry, extra dimensions, or the composite Higgs model attempt to explain this issue [11–13].

Moreover, cosmological observations show that Standard Model particles account for only about 5% of the universe's energy density, leaving dark matter, dark energy, and the matter–antimatter asymmetry unexplained [14, 15]. These gaps strongly suggest the existence of new physics.

To probe such phenomena, experiments must explore energy scales where these effects can manifest. Theoretical models predict new, heavy particles that would have existed in the early universe and can only be produced through high-energy collisions of known particles, such as protons [4]. This motivates the creation of high-energy accelerators like the LHC, to recreate these extreme conditions and uncover new fundamental physics [16].

2.2 CERN and the LHC

CERN (previously called *Conseil Européen pour la Recherche Nucléaire*), located near Geneva on the Franco-Swiss border, is the world's leading laboratory in particle physics research [17]. Founded in 1954, CERN has been responsible for significant advancements in accelerator physics and particle detection technologies. As illustrated in Figure 2.2, CERN's facilities host an accelerator complex composed of a series of machines. These machines successively accelerate particles before finally injecting them into the LHC.

The LHC is the final and most powerful accelerator in this chain. It is a superconducting synchrotron with a circumference of 27 km, installed about 100 m underground. Two counter-rotating proton beams are accelerated to nearly the speed of light and brought into collision at a centre-of-mass energy of 14 TeV and with a frequency of 40 MHz.

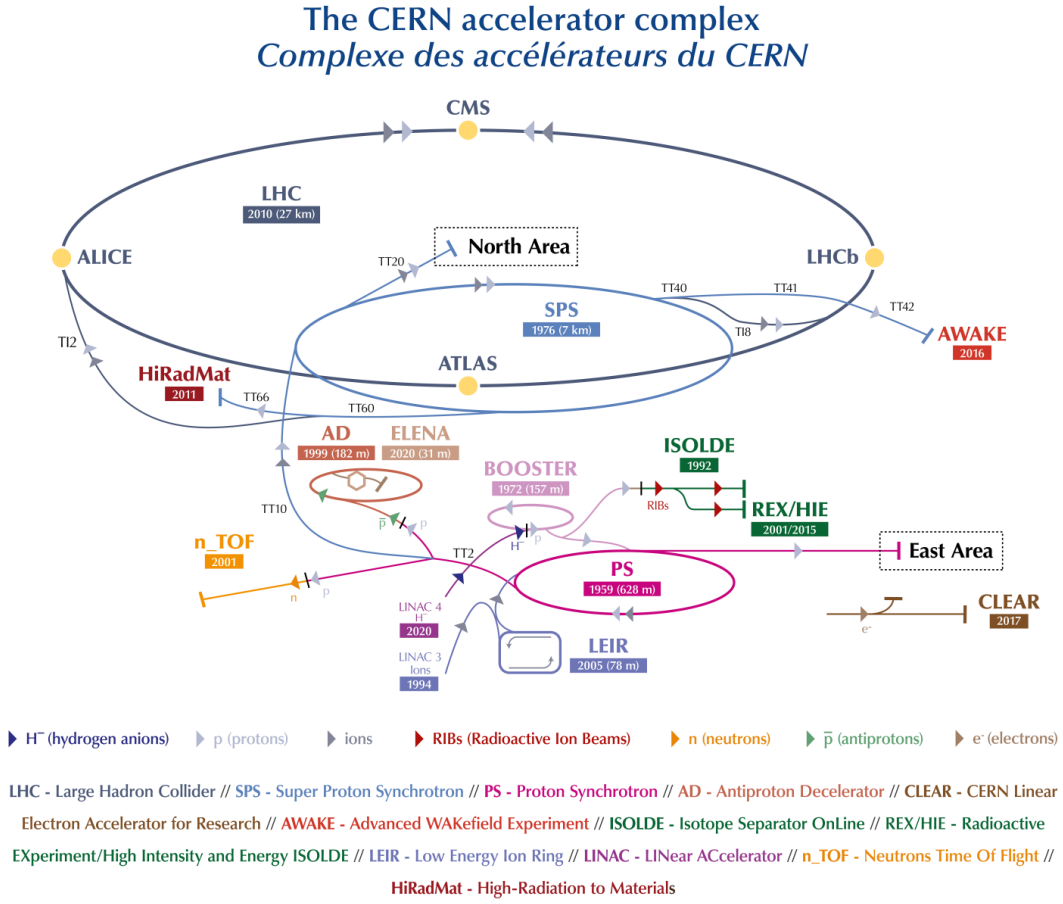

Figure 2.2: Diagram of the CERN accelerator complex and its experiments. The coloured arrows indicate the flow of the different particle species within the accelerators. The particles begin by being accelerated in a linear accelerator (LINAC3 or LINAC4), then pass to the first circular accelerators (LEIR or BOOSTER), which brings them up to a sufficient velocity for the Proton Synchrotron (PS). The particles are then sent to the Super Proton Synchrotron and finally to the LHC. Source: Mobs [18].

A key performance parameter of any collider is its luminosity, which measures the interaction rate per unit area. The synchrotron was designed for a peak luminosity of $\mathcal{L} \simeq 10^{34}\,\mathrm{cm}^{-2}\,\mathrm{s}^{-1}$, an occurrence of up to 1.2 billion interactions per second [19].

Each beam is composed of "bunches" containing approximately 10^{11} protons. When two bunches are made to collide at an interaction point, a "bunch crossing" occurs, and the detector response to that crossing forms a single *event*.

At current luminosities, the average number of pp interactions per bunch crossing is

$\langle\mu\rangle \approx 60$ [20]. The resulting superposition of signals from multiple simultaneous interactions, known as *pile-up*, increases detector occupancy and, consequently, the amount of data generated by the collisions. This poses significant challenges for event reconstruction and data processing.

2.2.1 LHC Coordinate system

In the LHC, a right-handed coordinate system is used, with origin at the nominal interaction point. The axes are defined as follows [21]: the z-axis is aligned along the beam pipe (the direction of the colliding proton beams); the x-axis points from the interaction point toward the centre of the LHC ring, and the y-axis points upward, perpendicular to the plane of the ring, as depicted in Figure 2.3.

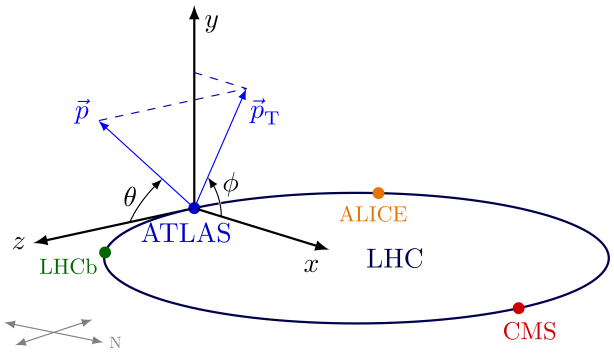

Figure 2.3: Schematic of the LHC coordinate system: beam axis (z), transverse plane ($x-y$), angle ϕ, and pseudorapidity η. Source: Chatham Strong [22].

The plane transverse to the beam (the $x-y$ plane) is used to define the cylindrical coordinates. The azimuthal angle ϕ is measured around the beam axis with respect to the x-axis. The polar angle θ is defined from the beam axis (the z-axis), but in collider physics it is common to use the pseudorapidity $\eta = -\ln[\tan\left(\frac{\theta}{2}\right)]$, which depends only on the direction of the particle momentum and not on its energy. When expressing detector segmentation, particle trajectories, or angular separations, the pair (η, ϕ) is used, because particle production tends to be more uniform in η and differences in $\eta - \phi$ space are invariant under boosts along the beam axis [21].

2.3 The ATLAS Experiment

At the interaction points of the LHC ring, there are four large-scale particle detectors that record the products of the collisions: i) A Large Ion Collider Experiment (ALICE), ii) ATLAS, iii) CMS, and iv) Large Hadron Collider beauty (LHCb). These experiments collect enormous volumes of data at extremely high rates. The analysis of this data is what allowed for the discovery of the Higgs Boson in 2012, as described in Section 2.1.

Among the main experiments, ATLAS is one of the general-purpose detectors constructed to observe a broad range of particles and physics phenomena, such as the Higgs Boson, extra dimensions and dark matter [21, 23]. ATLAS has a cylindrical geometry surrounding one of the LHC interaction points, with a length of 46 m, a diameter of 25 m, and a mass of about 7,000 tons, making it the largest particle detector ever constructed.

The detector is made up of various subsystems, each optimized for specific measurements. As presented in Figure 2.4, closest to the beam line, the inner tracking detectors serve to reconstruct the trajectories of charged particles. Surrounding them, there are calorimeter systems that measure the energy of electrons, photons and hadrons. Muons are identified by the outer part of the structure, in the muon spectrometer. A superconducting magnet system provides the magnetic fields required for momentum measurements. Together, they allow ATLAS to perform complete reconstruction of the collisions.

2.3.1 Calorimeters

The ATLAS calorimeters are of particular relevance to this dissertation, as they constitute the source of the bytestream data under study. Their role, among others, is to absorb and measure the energy and position of particles produced in pp or heavy ion collisions [24]. This enables the conversion of physical interactions into electronic signals that are later digitised, transmitted as raw data and can be unpacked for event reconstruction.

Calorimeters are typically sampling detectors, consisting of alternating layers of dense absorber material and active medium. The absorbers induce particle showers, multiplying the number of secondary particles, while the active layers record the deposited energy.

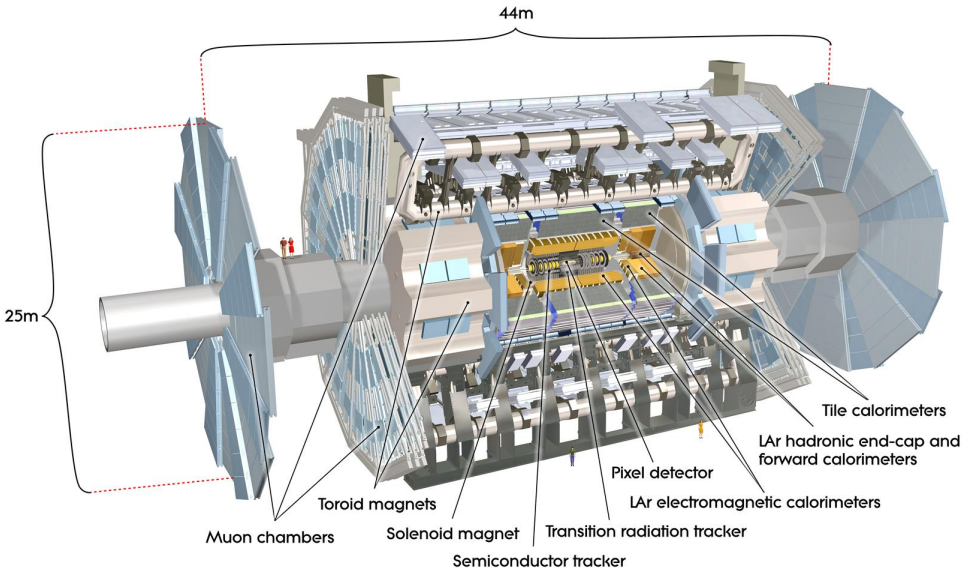

Figure 2.4: Cut-away view of the ATLAS detector. Human figures were placed next to it for scale. Source: Aad et al.[21].

In ATLAS, calorimeters are arranged in nearly hermetic coverage around the interaction point, ensuring that most of the particles produced in collisions are detected. The system combines electromagnetic calorimeters, optimized for electrons and photons, and hadronic calorimeters, optimized for strongly interacting particles such as pions and protons [21].

The electromagnetic calorimeter of ATLAS is based on LAr as the active medium. Lead plates act as absorbers, forcing incoming electrons and photons to initiate electromagnetic showers. The shower particles ionize the liquid argon, creating free charges that drift under an applied electric field and generate a measurable current. This current is collected by electrodes and transferred through cryogenic feedthroughs to front-end boards (FEBs). Since argon is liquid only at cryogenic temperatures, the calorimeter is enclosed in cryostats operating at approximately 88 K ($-185°$C) [25].

The front-end electronics amplify and shape the signals before transmitting them to the readout drivers, which digitise the information and format it into bytestream fragments. Surrounding the LAr is the hadronic calorimeter, known as the TileCal due to its construction from plastic scintillating tiles. Steel absorber plates initiate hadronic showers

when struck by protons, neutrons, or charged pions.

Secondary particles traversing the scintillator tiles produce light, which is collected by wavelength-shifting fibers and directed to photomultiplier tubes (PMTs). Each calorimeter channel is read out by two PMTs to improve precision and provide redundancy. The PMTs signals are proportional to the energy deposited in the scintillators, which is in turn related to the energy of the incoming hadron [26].

Both calorimeter systems are finely segmented in (η, ϕ), producing a three-dimensional map of the energy flow from the collision. The fine segmentation improves particle identification and event reconstruction, but it also means that the calorimeter readout consists of tens of thousands of individual electronic channels (cells), each contributing to the raw data volume. The resulting electronic signals are digitised and assembled into the bytestream format, a detector-independent structure that serves as the interface between hardware readout and software reconstruction [27]. The bytestream format is detailed in Section 3.1.

2.3.2 Trigger and Data Acquisition System

Given the bunch crossing frequency of 40 MHz and considering an average of 3 MB of data per event, the ATLAS experiment generates around 120 TB of data each second, which is impossible (and not useful) to store in its entirety. To filter out relevant information, ATLAS employs the TDAQ system.

When collisions happen, the front-end electronics digitise and store the signals in local buffers. Meanwhile, the Level-1 (L1) hardware trigger uses reduced granularity data from the detectors to evaluate the events: identifies high energy deposits, applies topological selections (tests how multiple trigger objects relate to each other in space and kinematics, such as requiring two objects to be close together, far apart, or consistent with a particular invariant mass [28]) and a configurable trigger menu makes a decision [29]. In Run-3, the L1 trigger accepts events at a maximum rate of 100 kHz, with a fixed latency of 2.5 μs. Once an event is accepted, the digitised data is read out to the read-out drivers (RODs).

The RODs sends event fragments to the read-out sub-system (ROS), which in turn buffers them in read-out buffers (ROBs). These fragments are delivered to the HLT on demand [30]. After the L1 accepts the event, a small portion of these fragments may be requested to reconstruct a part of an event by the region-of-interest builder (RoIB). If the event is deemed relevant, the whole event data will be requested by the HLT for reconstruction. The first step of the event reconstruction algorithms is the conversion of the raw data bytestream into appropriate `C++` structures, as later discussed in Section 3.1.

The HLT is a software-based trigger running on a large computing farm. In 2022, the HLT farm was comprised of around 56,000 heterogeneous CPU cores, with each node hosting a Data Collection Manager (DCM) (a single-threaded `C++` application that handles all I/O for the HLT tasks [30]) and one or more processing units. The HLT trigger selection involves multiple filter and hypothesis algorithms. An event is accepted if any chain passes all its hypothesis steps; otherwise, processing stops and the data is cleared from the ROS buffers. The current HLT acceptance rate is of approximately 3 kHz ($\approx$ 6 GB/s). The accepted events are sent to a dedicated cluster of servers for packing, compression, and transfer to offline storage [29]. An overview of the complete data flow can be visualised in Figure 2.5.

2.3.3 High Luminosity Upgrade

To extend its physics reach, the LHC is undergoing an upgrade known as the HL-LHC. Planned for completion at the end of the 2020s decade, it aims to increase the integrated luminosity to $\mathcal{L} \simeq 7.5 \times 10^{34} \text{cm}^{-2}\text{s}^{-1}$ and achieve a number of collisions per bunch crossing of $\langle \mu \rangle \simeq 200$. This will dramatically increase the demand on the TDAQ system.

The new architecture will adopt a single hardware Level-0 (L0) trigger that operates at 40 MHz and must reduce the readout data to 1 MHz in a span of 10 μs. The new software Event Filter (EF) (the previous HLT) is expected to perform offline event reconstruction to reduce the event rate to a manageable 10 kHz [20]. To cope with this, the use of more efficient computing architectures and software adaptations is being studied, as is further

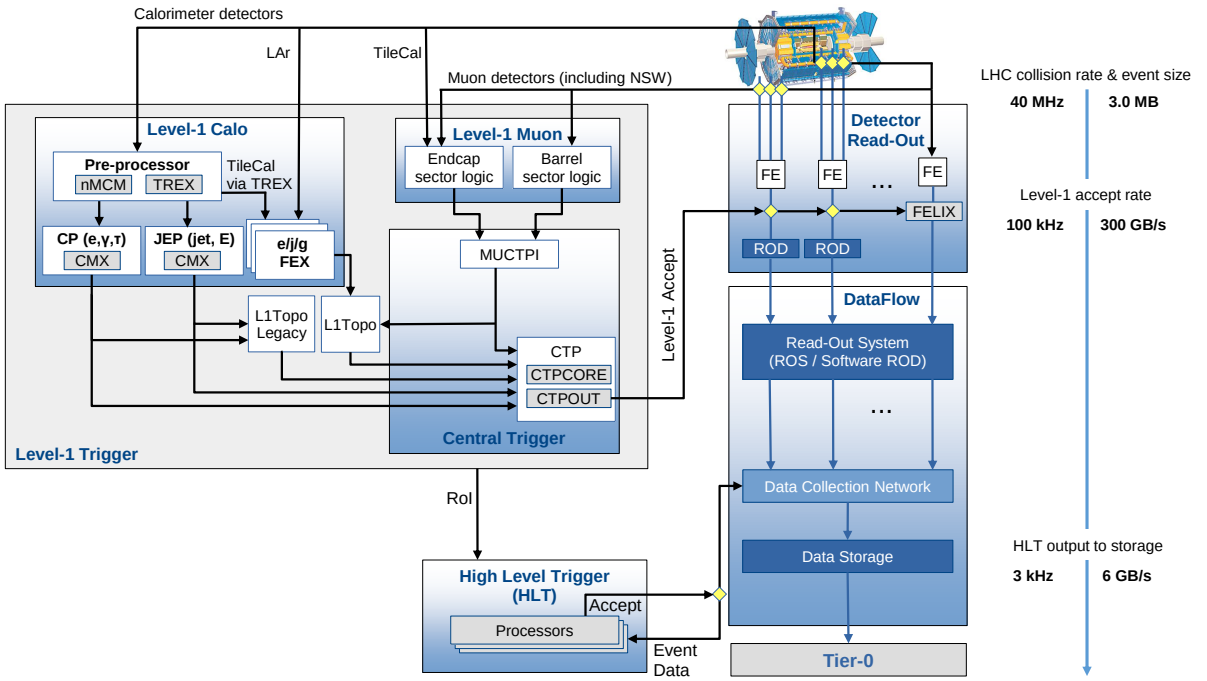

Figure 2.5: Schematic overview of the ATLAS Trigger and DAQ system at the beginning of Run-3. Source: G. et al.[31]

discussed in Chapter 3.

2.3.4 Athena Software

Athena [32, 33] is ATLAS's official software for processing event data. It is used for event generation, detector simulation, event reconstruction, and analysis. This encompasses what is known as the "offline" software. The Athena framework is also used for event reconstruction and selection at trigger level, which is called the "online" software [34].

Athena is built on top of the Gaudi framework [35], whose core architectural principle is a strict separation between the data being processed and the components that perform the processing. In practice, this means that data objects (representations of detector quantities or intermediate results) are stored independently in a central data store, and algorithm components access and manipulate this data exclusively through abstract interfaces rather than embedding data directly within processing code. This decoupling allows algorithms and data representations to evolve independently, simplifies testing and

maintenance, and makes it easier to replace or optimise individual components without affecting others [36].

The framework is organised in a hierarchy. At the lowest level, there are the `C++` components (algorithms, tools, or services) that perform the data processing. These components are grouped into logical packages that can be compiled independently. They can be configured in a chain to interact with data, characterising a job. The components developed for this dissertation are part of the `CaloRecGPU` package, which is a work in progress to port the logic behind the Calorimeter reconstruction algorithms to GPU.

Running modes Athena supports different running modes, the most used being serial and multi-threaded (AthenaMT). The serial mode runs one process on a single core, without concurrency, and is generally used only for local development and debugging. In contrast, AthenaMT supports multi-threaded scheduling by integrating Intel's Threading Building Blocks (TBB) programming model [34].

In AthenaMT, each algorithm explicitly declares its data dependencies, so the scheduler can create a graph to dynamically determine the execution plan of a job. An algorithm becomes eligible to run as soon as all of its required inputs are available from upstream algorithms, and a processing thread is free. This architecture enables inter- and intra-event parallelism, so different threads can process independent events at the same time, or, alternatively, different parts of the same event concurrently.

To function correctly in this environment, the components must be designed with thread-safe mechanisms. To manage this, algorithms are classified as i) non-cloneable (only a single instance of the algorithm exists for the entire job), ii) cloneable (the framework creates a pool of instances of the algorithm, so multiple threads work on different events), or iii) re-entrant. Re-entrant algorithms are explicitly written to be thread-safe and allow inter-event parallelism. The developer bears the responsibility of ensuring that there are no race conditions in the code.

Components and configuration Algorithms are the primary processing units of the framework. They process input data and produce an output, with both stored in the transient store, which is detailed later in this Section. They are implemented as `C++` classes that extend the `AthAlgorithm` or `AthReentrantAlgorithm` class. Algorithms must implement the `initialize()`, `finalize()` and `execute()` methods. While the latter is executed once for every event, the former methods are only executed once, at the start and end of a job. This work was implemented as an extension of the `AthReentrantAlgorithm` class [34].

Tools are reusable, configurable sub-components that are owned and used by algorithms or other tools. While algorithms are executed only once per event, tools can be invoked as many times as necessary. Finally, services are singleton components that provide common, job-wide functionalities to any component that requests them. They are persistent throughout the job, so they are only instantiated once and can be viewed by all other components. The most used service is the `StoreGateSvc`, which is responsible for handling access to permanent storage. Services are also needed to grant access to specific databases, such as the ROB data provider service, as later discussed in Section 3.1.

Although the event-processing components are implemented in `C++`, ATLAS jobs are configured in Python. The Python layer builds a `ComponentAccumulator` (CA) object that holds all configuration of the job: which algorithms, tools, and services should run; input and output data and their respective access keys; and the values of the components' configurable parameters, which are exposed in `C++` as `Gaudi::Property` member variables. In addition, the CA carries the job-level `ConfigFlags` (e.g. input files, detector geometry, conditions tags, and concurrency). At runtime, the framework instantiates the `C++` components and populates their `Gaudi::Property` members from the `CA`, allowing one to change algorithm parameters without recompiling the `C++` code.

Data storage and retrieval The Athena framework manages two distinct forms of data for every event: transient and persistent. Transient data refers to the `C++` objects that algorithms create, manipulate, and exchange in memory during the execution of an Athena job. Their existence is ephemeral, lasting only for the processing of a single event

before being deallocated. Persistent data, on the other hand, is the on-disk representation of the data, serialized into a format suitable for long-term storage and retrieval.

Of relevance to this dissertation is the transient data store. It is what allows the exchange of data between algorithms during execution. In Athena, this is achieved by the store gate service (`StoreGateSvc`), which can be conceptualized as a dictionary that maps unique string keys to data objects in memory [37].

An algorithm retrieves its necessary input data objects from `StoreGate` by requesting them via their type and key. After performing its computations, it creates new output objects and records them back into `StoreGate` with a new key. Subsequent algorithms in the processing chain can then access these objects. This is what allows the decoupling between the data-producing and data-consuming algorithms.

Because AthenaMT processes many events concurrently, the `EventContext` was introduced to uniquely identify each event. It encapsulates information such as run number, event number, and timestamp, and is passed to the `execute()` method of every algorithm.

To access event data, Athena uses a system of read and write handles. Instead of being permanent members of a component, algorithms must declare `ReadHandleKey` or `WriteHandleKey` objects to configure the handle by storing the identifier string of the data object in `StoreGate`. The keys are must be configured in the job initialization step and, during execution, the handle is constructed using both its key and the `EventContext`, ensuring that the correct payload for the event is retrieved. In addition, these handles serve as explicit declarations of data dependencies, which, as mentioned earlier, the scheduler uses to build the job's execution graph. All `HandleKey` objects must be initialized on the algorithm's `initialize()` method.

Apart from the `StoreGateSvc`, Athena contains two other relevant event stores, the Detector and Conditions stores. The Detector store holds detector specific configuration and constant geometry information. Algorithms can access it via the `DetectorStoreSvc` service. Time-dependent detector calibration and alignment data is held in the Conditions store. It's data may change for every event, and a different kind of handle is necessary to interface with it (`ReadCondHandle` and `WriteCondHandle`).

Chapter 3

Technical Background

This chapter provides the technical background necessary to understand the development and implementation of the GPU-based bytestream decoder. The structure of the bytestreams from the ATLAS LAr and Tile calorimeters is described, as well as the CPU-based decoding logic. It also covers the basic concepts of heterogeneous computing, describes the `CUDA` architecture, and focuses on the use of accelerators in high-energy physics.

3.1 Calorimeter Bytestream

The calorimeter bytestream represents the most granular digital record of the signals measured by the ATLAS calorimeters. It contains the raw output of the front-end electronics after minimal on-detector processing and is the first format in which calorimeter data becomes accessible to the software reconstruction chain.

The ATLAS raw event format (the bytestream) is structured hierarchically in a series of fragment headers. The full event is composed of multiple sub-detector fragments, each containing one or more ROB fragments – see Figure 3.1.

The ROB fragments contain a header, the inner data payload (typically a ROD fragment) and an optional trailer checksum. The ROB fragment header has a common generic

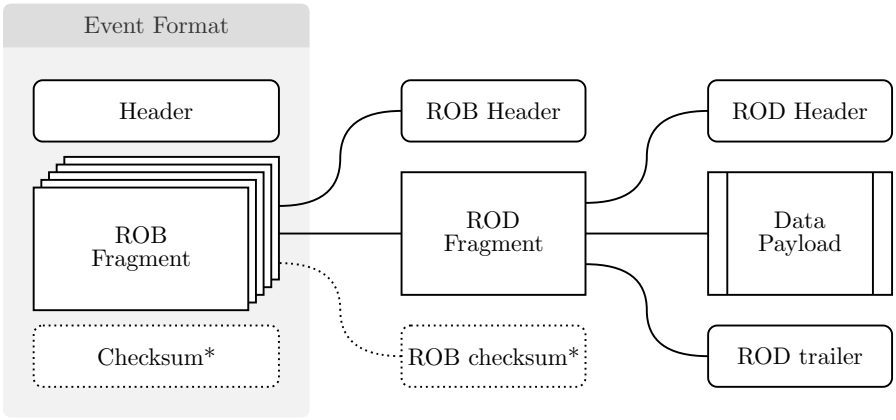

Figure 3.1: Organization of ATLAS raw event format. Dotted elements are optional.

header followed by a sub-detector specific header. The generic header contains information such as fragment size, header size, version number, and hardware source identifier [27].

In this dissertation, the data being processed are part of the LAr and TileCal ROD payloads. In a ROD specific header, one can extract the run number, bunch-crossing identifier (BCID), and event type, among other fields. The fragment also contains a series of status words, with the first of them indicating the global status of the fragment (a non-zero value means that the data payload might be corrupted, with missing data or bit errors) [27].

Finally, the data payload has a detector specific format, as the hardware of the Digital Signal Processorss (DSPs) that generates it is not the same. Therefore, the LAr and TileCal payloads must be decoded differently.

3.1.1 LAr Bytestream

The amplification, shaping, and digitisation of the signals generated by the collisions are read-out by a hardware component known as a FEB. In the LAr architecture, each FEB services up to 128 readout channels (calorimeter cells). In the off-detector back-end, ROD boards collect data from multiple FEBs via GLINK optical inputs. When on "physics mode" operation, the onboard DSPs compute calibrated quantities (energy, time,

quality) from the raw analog-to-digital converter (ADC) samples, and then the outputs of two FEBs are merged by an output controller and sent over a single SLINK optical link into a ROB. The ROB stores the fragment until it is requested by higher-level components of the Data Acquisition (DAQ) system [38].

When unpacking the bytestream, the data reflects this data path, but in the opposite direction. One must first identify the ROB that received the ROD payload. Its packed ROD fragment groups the read-out channels of two FEBs, totalling 256 raw channels per ROD. The organisation of the bytestream payload of a ROB is depicted in Figure 3.2.

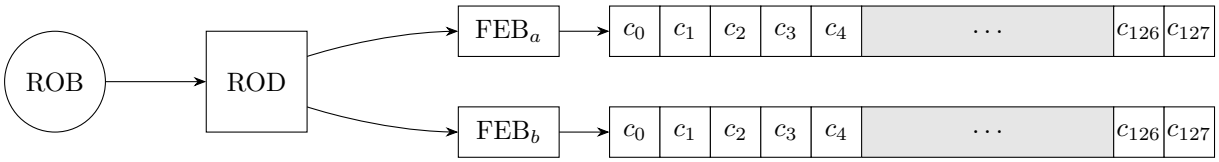

Figure 3.2: LAr bytestream structure. The ROB payload encompasses the ROD metadata, that contains package produced by two FEBs and their read-out channels. The output channels represent the encoded energy, gain, time and quality information.

In physics-mode operation, the data recorded in the bytestream reflect the result of real-time signal reconstruction performed by the on-board DSPs in the RODs. Upon trigger acceptance, a small, fixed number of ADC samples (typically five) per pulse are forwarded from the front-end electronics to the ROD. The DSP applies an optimal-filtering algorithm — a linear estimator that computes a noise-suppressed weighted sum of the digitised samples, to extract the true pulse amplitude and timing, based on a known reference pulse shape and noise characteristics — in order to minimise noise contributions and maximise the accuracy of the signal amplitude and timing. From this weighted sum, the DSP extracts the pulse amplitude (and optionally time and a quality metric) in ADC counts [39]. Finally, the DSP converts the amplitude from ADC counts into physical energy units (MeV) using calibration constants, retrieved from the Conditions database, and writes this reconstructed energy (and associated metadata) into the bytestream.

The energy is therefore already reconstructed and approximately calibrated, though

not yet corrected to the final electromagnetic energy scale used in the HLT and offline reconstructions. Each channel operates in one of three possible electronic gains (high, medium, or low), and the DSP uses gain-dependent coefficients and conversion factors to ensure that the output amplitudes are consistent across gains. The reconstructed energy is stored as a 16-bit signed integer in the bytestream payload, accompanied by gain, time, and quality information [38].

The packed quality factor quantifies the agreement between the measured pulse samples and the reference pulse shape used in the digital optimal filtering algorithm. The optimal filtering coefficients themselves are derived offline from dedicated calibration runs that measure reference pulse shapes and noise behaviour; one set is chosen to maximise the signal-to-noise ratio for amplitude estimation, while a second set is chosen to extract the pulse timing with sub-nanosecond precision, with the amplitude coefficients selected to minimise the variance of the reconstructed energy estimate and the timing coefficients chosen to emphasise the waveform slope around its peak for precise temporal measurement [38].

The reconstructed time value is encoded in the bytestream as a fixed-point integer; for typical pulses in the LAr electromagnetic calorimeter, the time resolution has been measured to be on the order of 70 ps (with the electronics contribution alone at $\approx$20 ps) [40]. This sub-nanosecond precision means that deviations of a few ns due to out-of-time pile-up or timing miscalibrations can be reliably detected on a channel-by-channel basis, enabling the identification and mitigation of delayed or early energy deposits.

Both the reconstruction time and quality are only meaningful if the signal amplitude is sufficiently above the electronic noise level. For low-energy signals, the digitised samples are dominated by random noise fluctuations, and the resulting time and quality values no longer correspond to physical pulse properties but to statistical noise variations. To avoid transmitting meaningless information, time and quality are included in the payload data only for channels where the signal-to-noise ratio exceeds a configurable threshold.

Although this work focuses only on the energy reconstruction, the bytestream also accounts for a Digits and Raw data block which may contain a set of ADC samples per

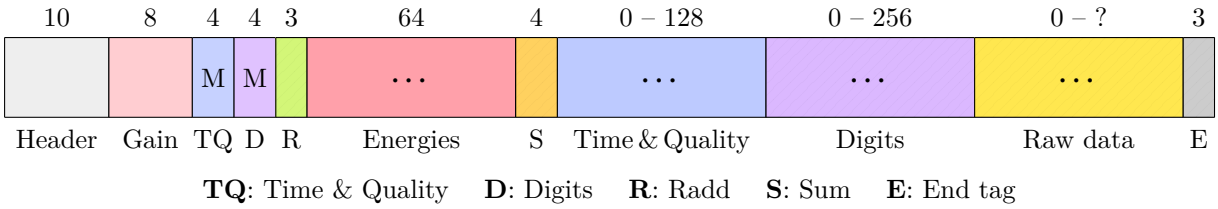

Figure 3.3: LAr FEB bytestream layout. The numbers above each block represent the number of 32-bit words the field is composed of. The small blocks marked with an "M" are mask fields used to verify the presence of the optional per cell time and quality and digits quantities. The dashed fields are not used in the energy reconstruction.

channel instead of the reconstructed quantities [38]. These blocks are more commonly used for calibration and debugging, so they will not be discussed here. It is important to note, though, that they cannot be removed from the payload without prior parsing.

The general layout of the bytestream of a FEB is represented in Figure 3.3. A complete ROD payload is made up of two such streams, separated by a 7 32-bit word spacer header. The inner details of each decoded field, as well as the mapping between raw channels and the final cells, are further discussed in Chapter 4.

3.1.2 Tile Bytestream

The TileCal bytestream is the serialised digital representation of the signals recorded by the calorimeter's PMTs and formatted by its specific front-end electronics (FEE). Similar to the LAr structure, each ROD payload contains the data acquired from one or more drawers – a long electronic module servicing up to 48 read-out channels in the barrel or extended barrel partitions [41]. Figure 3.4 shows the structure of the TileCal bytestream.

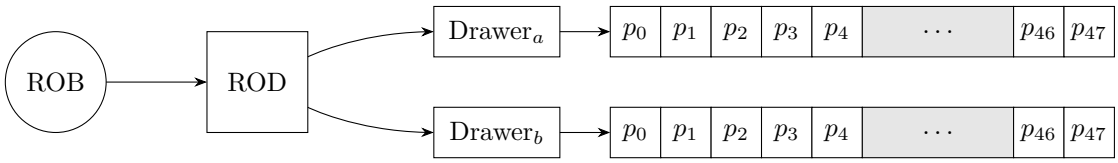

Figure 3.4: TileCal bytestream structure. The ROB payload includes the ROD metadata, that contains package produced by two drawers and the read-out of up to 48 PMTs. The output channels represent the encoded energy, gain, time and quality information.

The data relevant to this dissertation were taken with the RODs operating in the reconstruction mode. The raw digitised samples from each PMT channel are processed by the on-board DSP to produce three quantities: the reconstructed amplitude (energy), the pulse time, and the quality factor.

Each PMT converts the scintillation light collected from a calorimeter channel unit into an electrical pulse, which is amplified and shaped by the FEE before being digitised at 40 MHz sampling rate. The analogue signal is split into two gain channels: a high-gain and a low-gain path, with a ratio of 64:1. Both gains are continuously digitised by 10 bit ADCs, and, upon receiving a L1 trigger accept signal, a window of seven consecutive samples is extracted from the pipeline memory and transferred to the RODs for reconstruction [41].

Inside the ROD, the DSPs apply a fast implementation of the optimal filtering algorithm to these samples, with coefficients specific to each channel and gain, just as in the LAr calorimeter. The resulting amplitude is converted from ADC counts to MeV-equivalent units using calibration constants derived from dedicated calibration runs. Similar to the LAr bytestream, the time provides the phase of the signal relative to the LHC clock, and the quality factor quantifies the residual between the measured and expected pulse shapes.

After this primary reconstruction, the DSPs pack the computed values into the data payload. The values are stored as fixed-point integers within a 32-bit word for each channel, together with a gain flag indicating whether the high- or low-gain chain was used. The individual drawer payloads are concatenated into a sub-fragment of the raw-channel type, which is transmitted through the S-link optical interface to the ROD for unpacking.

A portion of the drawers may also contain a distinct subset of cells, the Minimum Bias Trigger Scintillators (MBTS) cells. These are special scintillators that live between the barrel and the end-caps [42], and their output signals must be handled differently by the decoder algorithms. The way their signals are encoded is equal to the PMTs, but their values are stored in a separate container.

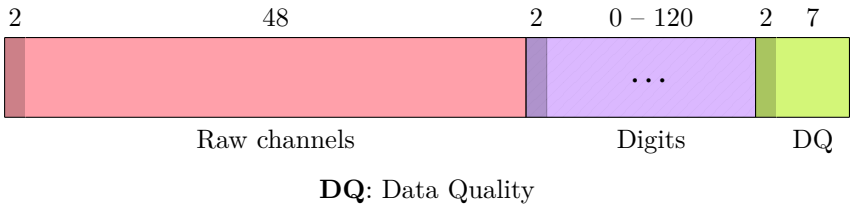

Figure 3.5: Tile ROD fragment layout. The numbers above each block represent the number of 32-bit words the field is composed of. The darker, unlabelled blocks are sub-fragment headers that should all contain the same identifier. The dashed fields are not used in the energy reconstruction.

There can also be a data quality (DQ) sub-fragment related to the drawer in the payload. This is a per-event integrity summary produced by the ROD DSP that carries a global Cyclic Redundancy Check (CRC), the DSP's reconstructed BCID, BCID consistency bits, and a set of flags that mark encountered data problems. If the identified errors indicate widespread issues, the DQ mask can be used in the downstream reconstruction to skip channels likely to be corrupted.

A Digits sub-fragment might also be present, but as for the LAr, it will not be discussed. Each of these sub-fragments contains a small two 32-bit word header with the sub-fragment payload size, drawer identifier, and a set of flags specific to the sub-fragment type. The overall layout of the data payload retrieved for a single drawer is represented in Figure 3.5. The complete payload of a ROD consists of two of them in sequence. Again, the inner details of each field and the mapping procedure between PMTs and TileCal cells are discussed in Chapter 4.

3.2 CPU Decoder algorithms

The bytestream decoding of the ATLAS calorimeters is implemented within the `TrigCaloDataAccessSvc` class, a service of the HLT framework responsible for transforming raw calorimeter fragments into physics-ready cell objects. Within the HLT software chain, this service forms the interface between the trigger dataflow system – where raw fragments

are received from the detector readout – and the calorimeter reconstruction tools that require structured per-cell information such as energy, time, and quality.

When a region of interest or full event request is issued, the `TrigCaloDataAccessSvc` queries the `ROBDataProviderSvc` for the relevant ROB fragments belonging to the LAr and TileCal. The service selects the detector-specific decoding tool to interpret the ROD payloads and convert them into calorimeter channel data. During this process, information from the Conditions Store (e.g., cabling and identifier mappings and bad-channel masks) is used to match the unpacked data to the corresponding calorimeter cell identifiers.

For the Liquid Argon calorimeter, once the FEB payloads can vary in size (depending on the number of channels that contain time and quality (TQ) data), the `LArRodDecoder` must first parse the FEB header to locate the start of each data field (raw channels, TQ, etc). Only then can it iterate through the words to unpack the correct channel data.

The TileCal, in contrast, benefits from the predictable data format. Since all subfragments start with a small header, and the channel and DQ fragments have a fixed size, the `TileROD_Decoder()` can jump directly to the data region and process all channels sequentially without extensive checking. Since each cell of the TileCal is read out by two distinct PMTs, their signals must be combined after the unpacking is done.

Despite these structural and organizational differences, both decoders share a common conceptual flow. Each fragment is traversed linearly, control words are interpreted, and fixed per-channel operations are applied to produce calibrated quantities. The cell decoding logic (the numerical conversion of raw outputs into reconstructed quantities) is similar across all channels of the given detector, with a few corner cases to be dealt with in the TileCal unpacking, due to the merging of PMTs signals.

Once all channels are processed, the resulting calorimeter cells are inserted into thread-local caches (`LArCellCont` and `TileCellCont`), which are merged into a unified `CaloCellContainer` by the `TrigCaloDataAccessSvc`. This container represents a view of the calorimeter data, indexed by cell identifiers, and is the entry point for the clustering and energy-summing algorithms used in the HLT and offline reconstruction.

3.2.1 Current Limitations

The current CPU-based decoding system is reliable, compatible with diverse front-end configurations, and has a stable integration within the Athena framework. However, its algorithmic structure and data layout were conceived more than two decades ago, and although the decoders are efficient enough in the current data taking conditions, there are several intrinsic limitations that become apparent as data rates increase.

The first limitation is the inherently sequential nature of the decoding workflow. The absence of intra-fragment or even inter-ROD parallelism means that the overall throughput cannot easily grow with the number of processing cores or threads.

A second limitation arises from memory layout and access patterns. The cell containers used by the calorimeter reconstruction are implemented as arrays of structures (AoS), where each element holds all attributes of a given cell. This organisation is convenient for CPU traversal and object-oriented programming, but it results in scattered memory accesses and limited cache efficiency when the decoder loops over large numbers of channels. Additionally, most decoding functions use pointer-based navigation across complex nested structures, an idiom common in legacy `C++` code but suboptimal for modern architectures. These patterns lead to irregular memory access, poor data locality, and reduced cache reuse, particularly when decoding many small fragments interleaved in memory.

A third limitation relates to data placement and interoperability with hardware accelerators. The decoding stage produces fully populated cell containers in CPU memory. However, many calorimeter reconstruction and clustering algorithms are being developed or prototyped on GPUs or FPGA. To operate on these devices, data must first be retrieved from the CPU containers, transformed into a structure of arrays (SoA) layout suitable for coalesced memory access, and then transferred to device memory. This additional retrieve–transform–transfer sequence introduces significant overhead, both in latency and host–device bandwidth [43]. If the containers could be constructed directly in a device-resident, GPU-friendly layout during decoding, these steps could be avoided entirely.

Another factor limiting scalability is framework coupling and data dependencies. The

decoders are tightly integrated into the Athena infrastructure, relying on numerous services such as identifier mapping, conditions access, and monitoring tools. These dependencies, while necessary for correctness and calibration, constrain the ability to offload portions of the decoding logic to external accelerators.

Finally, although branching and control flow (the points in code where execution can follow different paths based on conditional logic, such as if statements or loops that depend on data values) do not represent a major performance issue on CPUs, thanks to branch prediction and speculative execution[1], they do contribute to a complex and irregular instruction pattern that reduces the benefits of instruction-level parallelism. The variability of fragment sizes and conditional inclusion of fields also complicates attempts to reorganise the decoding loop for vectorised execution. While these effects are modest on traditional processors, they become more relevant when considering accelerators with stricter execution coherence requirements, where many threads must follow the same instruction sequence to achieve high throughput.

Together, these characteristics define the current boundaries of CPU-based calorimeter decoding performance. The decoding step is not entirely inefficient by design, but it was never intended to scale to the data volumes foreseen for the HL-LHC. These limitations motivate the investigation of alternative computing strategies. In particular, exploiting the uniformity of the per-channel decoding logic and the independence of individual fragments opens the possibility of offloading this workload to massively parallel architectures. Chapter 4 presents a proof-of-concept implementation of a GPU-based decoder designed to evaluate the feasibility and potential performance benefits of this approach.

3.3 Heterogeneous Computing

For decades, computing performance was reliably delivered by increasing the transistor density of CPUs, as predicted by Moore's Law [44, 45], and by raising clock frequencies,

[1] Branch prediction and speculative execution are mechanisms by which modern CPUs guess the likely path of a branch and start executing instructions ahead of time to keep pipelines full; mispredictions force the processor to discard work and recover.

a principle known as Dennard scaling [46, 47]. However, in the mid 2000s, increasing clock speeds any further led to prohibitive power consumption and heat generation, a phenomenon termed the "power wall" [48]. With that, the industry shifted its focus to parallelism. First, by adding more identical cores to a single CPU (homogeneous multi-core computing) and, later, by integrating specialised co-processors or accelerators. This paradigm shift towards heterogeneous computing redefined the architecture of high-performance systems.

Heterogeneous computing refers to systems that utilise more than one kind of processor or core to execute a computational task, by combining a general-purpose CPU with one or more specialised co-processors. This architectural diversity can be leveraged by identifying the sub-tasks of an application that can benefit from different hardware architectures. A CPU, which is optimised for low-latency execution of complex control flow and serial tasks, manages the overall application. In turn, computationally intensive, highly parallel portions of code can be offloaded to an accelerator specifically designed for high-throughput computation [49].

Specialised accelerators can achieve performances that are orders of magnitude higher for specific workload types, while consuming significantly less power than a CPU would for the same task [50]. This, however, increases the development complexity at all levels of the computing stack. The primary burden of performance optimisation is transferred from hardware architects to software developers, who must adapt system designs and refactor legacy algorithms so they can better utilise the accelerator's capabilities [48].

While a diverse range of accelerators is currently available, their selection should be dictated by the specific needs of the application, such as the nature of its parallelism, latency constraints, and memory access patterns. Among the most commonly used hardware accelerators, there are GPUs, FPGAs and Application-Specific Integrated Circuits (ASICs).

GPUs were originally constructed for efficient graphics rendering but became increasingly more programmable, allowing their use in general-purpose computing. A far greater portion of its silicon is used in its data processing units if compared to a CPU [51]. This

makes GPUs well-suited for data-parallel problems, where the same operation is applied to a large number of data elements simultaneously. The architecture is capable of high-throughput, and together with its high memory bandwidth, GPUs became a cornerstone of modern high-performance computing [49].

FPGAs are integrated circuits that contain an array of programmable logic blocks and a hierarchy of reconfigurable interconnects. This allows the blocks to be rearranged by a programmer after manufacturing, which in turn enables the creation of custom hardware circuits tailored to specific algorithms. Applications that can be expressed as a deep pipeline or fixed-function logic can benefit considerably from the use of FPGAs, which, in this case, can offer superior performance and power efficiency when compared to GPUs [52]. Due to its ability to achieve low and deterministic latency, FPGAs are ideal for real-time applications such as network packet processing, high frequency trading and, in HEP, are especially useful for first-level data processing in detector trigger systems [53, 54].

While the discussed architectures offer considerable levels of flexibility, the ASIC architecture is a chip fabricated to perform a single, specific function. These circuits deliver the highest possible performance and power efficiency, but once manufactured, they cannot be reprogrammed, making their inflexibility their primary drawback. The cost of designing, verifying, and fabricating an ASIC is also very high [55]. ASICs are widely used in HEP detectors as part of their trigger hardware [29, 56] and front-end readout electronics [57].

3.4 GPUs as Accelerators

Given that this work employs GPUs as the primary computing device, this section provides a more detailed characterisation of their architecture and execution model. The principles that make GPUs particularly suited for massively parallel, data-intensive workloads, such as calorimeter data unpacking, are hereafter highlighted.

GPUs were designed to maximise the total number of operations completed per unit

of time (throughput) across thousands of concurrent threads. They deliberately sacrifice single-thread performance to achieve massive aggregate computational power. This is carried out by a large array of simple, in-order Arithmetic Logic Units (ALUs) (`CUDA` cores) and high-bandwidth memory interfaces (historically, the bandwidth of a GPU has been up to an order of magnitude greater than that of a CPU [58]).

The control logic of a GPU is also much simpler if compared to a CPU's. Instead of employing large caches to hide memory latency, it exploits massive hardware multi-threading. By having thousands of threads available for execution, the hardware can be kept busy with other work if a portion of the threads get stalled by long-latency memory operations. Because the GPU performance is not dependent on the speed of a single processing unit, it was not affected by the cited "power wall," and continued to scale by integrating more simple cores with each generation of semiconductor technology [58].

3.4.1 Programming models

Despite being a powerful accelerator, not all computing tasks fit the GPU architecture. GPUs thrive when processing problems with a high level of data parallelism, which occurs when computations performed on different parts of a dataset can be done independently. Often, writing data parallel code entails reorganising the computation around the data so that the independent tasks are executed in parallel, completing the overall job faster [58].

Consequently, GPU programmers often need to make explicit choices that are generally abstracted in traditional CPU programming. To effectively use a GPU, developers must identify appropriate tasks that can benefit from parallel execution and, depending on the programming model, they are responsible for the direct management of memory allocation, data transfer orchestration, and the launch of device side computation [48].

The most prominent GPU programming models are currently the following:

- **`CUDA`:** Introduced by NVIDIA in 2006, `CUDA` is a proprietary parallel computing platform and programming model specifically designed for NVIDIA GPUs. It is an extension of `C/C++` that gives developers direct access to the GPU's virtual

instruction set and parallel computational elements. This low-level control enables fine-grained optimisation. NVIDIA also provides a large set of libraries, tools and developer resources to facilitate development [51].

- **Open Computing Language (`OpenCL`):** Developed by Apple and maintained by the Khronos Group, `OpenCL` is the open source, royalty-free answer to `CUDA`, designed to be vendor-agnostic. `OpenCL` allows developers to write code that can target many devices, including CPUs, GPUs, FPGAs and DSPs from various manufacturers [59].

- **Radeon Open Compute Platform (ROCm):** ROCm is AMD's open-source software stack for GPU computing on its Radeon and Instinct accelerators [60]. ROCm has a `C++` runtime Application Programming Interface (API) and a kernel language called Heterogeneous-compute Interface for Portability (`HIP`), which was intentionally designed with a syntax similar to `CUDA`. AMD also provides tools to automatically translate `CUDA` code into `HIP` with minimal intervention. This was aimed at lowering the barrier for developers to migrate existing applications from the dominant NVIDIA ecosystem to AMD hardware.

- **`SYCL`:** Also maintained by the Khronos Group, `SYCL` is a higher-level programming model that enables single-source `C++` programming for heterogeneous systems [61]. With `SYCL`, host and device code can be written in the same source file using standard `C++` features. `SYCL` is the foundation of Intel's oneAPI initiative [62] to unify programming models across the different available Intel hardware architectures.

Despite the appeal of open, vendor-neutral platforms like `OpenCL` and `SYCL`, `CUDA` is the *de facto* industry standard for General-Purpose Computing Graphical Processing Unit (GPGPU) computing, particularly in scientific research and artificial intelligence. This is not necessarily related to technical superiority, as research shows that `SYCL`, `OpenCL` and ROCm have evolved to produce programmes with performance that is very similar to (or even better than) `CUDA` [63–66]. The dominance of `CUDA` comes from a long-term investment in building a comprehensive and stable developer ecosystem.

Since the platform was launched in 2006, CUDA established a multi-year lead before relevant competitors emerged, and its ecosystem was designed to eliminate as much friction for developers as possible. This includes the availability of highly optimised libraries for use in fields such as linear algebra, Fourier transforms and deep neural networks [67], which provided high-performance implementations of common primitives and encouraged scientific use. NVIDIA also provided robust profiling and debugging tools [68], extensive documentation [69] and active community support forums [70]. All of this served to lower the entry barrier and accelerate development and adoption.

In contrast, open standards have struggled to achieve the same level of cohesion. OpenCL's evolution was slow, and vendor implementations often varied in quality, performance, and feature support, undermining its promise of "write once, run anywhere" and frustrating developers [71, 72]. SYCL, while more modern, is still working to reach the same level of ecosystem maturity [64, 73]. The strategy of AMD's ROCm/HIP, which focuses on the portability of CUDA code, just highlights CUDA as an established and dominant API for GPU programming, even with migration getting easier day by day [74–77].

3.5 Fundamental CUDA concepts

This section explores the fundamental concepts of the CUDA programming model in greater depth. Understanding the CUDA execution and memory models is necessary to explain how the proposed algorithms are organised, optimised, and parallelised on the GPU architecture. The following subsections describe the thread hierarchy, execution model, and memory spaces that underpin the implementation choices presented in Chapter 4.

3.5.1 CUDA execution model

A CUDA program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer. The GPU code includes functions (kernels) to be executed in a data-parallel manner. The execution starts with the host code, and when a kernel function is called, a large number of threads are launched on the GPU to execute it. The launching

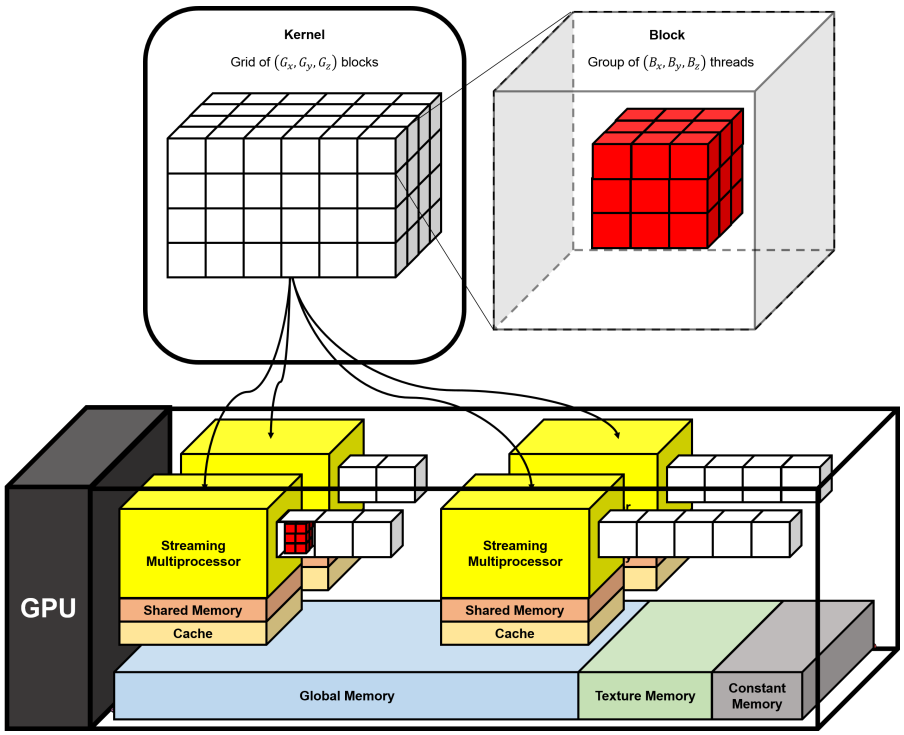

Figure 3.6: Layout of the execution model and architecture of a CUDA-compatible GPU. Each kernel is comprised of multiple blocks of threads, each block is executed on a streaming multiprocessor. The streaming multiprocessors offer shared memory that can be accessed by the threads within a block, as well as a local cache for global memory accesses. The GPU also contains texture memory and constant memory that can be accessed by the streaming multiprocessors. Source: Fernandes [43]

of threads can be assumed to take very few clock cycles, contrasting with traditional CPU threads that can be thousands of times slower. Kernels are designated by the `__global__` function type qualifier and are compiled with the GPU's instruction set [58].

The threads launched by a kernel are classically organised into a three-level hierarchy composed of threads, blocks, and grids. Threads are the most fundamental unit of execution in the CUDA model. Each thread executes an instance of the kernel function and is uniquely identified by an ID. Threads possess their own program counter, a private set of registers, and follow a single, sequential path through the kernel code [58]. A schematic view of the CUDA execution model and memory hierarchy is shown in Figure 3.6.

Thread blocks are groups of threads that are executed together. All of the threads of a block are guaranteed to execute on the same Streaming Multiprocessor (SM). An SM

can be described as an array of streaming processors (the ALUs, or `CUDA` cores). Each SM generally operates independently from the others and, apart from the cores, is composed of an on-chip memory and a warp scheduler [58], which will be discussed shortly.

Once all threads of a block are executed in the same SM, they can communicate and share data through the on-chip shared memory space. The execution of threads within a block can also be coordinated using the `__syncthreads()` intrinsic function. Once a thread reaches a barrier, it pauses and waits until all threads of the block have reached it, ensuring the current stage of the computation is completed before continuing [58].

Lastly, all thread blocks launched by a kernel invocation form a grid. Blocks within a grid function independently, so they cannot share data through shared memory or synchronise with a single barrier. This structure is what provides transparent scalability. A programmer typically defines the size of the grid based on the size of the data being processed, not the number of physical processors in the GPU. The `CUDA` runtime and GPU scheduler are then responsible for block distribution across SMs, which can execute them in any order, serially or in parallel.

Consequently, a GPU with a small number of SMs will execute the blocks over a longer period of time, while a GPU with many SMs can execute more blocks concurrently, finishing work faster. The application code itself remains unchanged across a wide range of hardware, allowing software to scale once the hardware is updated [58].

To map parallel computation onto structured data, `CUDA` allows threads and blocks to be organised into one, two, or three dimensional structures. Within a kernel, a thread can determine its unique position in this hierarchy using a set of built-in, read-only variables represented by a tuple (x, y, z). These include `threadIdx` and `blockIdx`, which specify the thread index within its block and the block index within a grid, respectively. Additionally, the variables `blockDim` and `gridDim` define the dimensions of the block (number of threads in each dimension) and grid (number of blocks in each dimension) [58].

3.5.2 SIMT Architecture

While `CUDA` promotes an abstraction of thousands of independent threads, once a thread block is assigned to an SM, the hardware partitions it into groups of 32 threads, known as warps. They are grouped according to their linearised thread index. At any given clock cycle, an SM issues a single instruction to be executed by all threads in a warp, which amortises the cost of fetching and decoding one instruction across 32 separate ALUs [58].

This is what defines the Single Instruction, Multiple Thread (SIMT) architecture. Conceptually similar to the Single Instruction, Multiple Data (SIMD) model of vector processors, it offers a more flexible programming abstraction. While the hardware executes in lockstep, each of the 32 threads has its own independent state, thanks to their individual program counter and register file. This gives the programmer the illusion of writing code for fully independent, scalar threads, as if each were executing in its own core [58].

However, the SIMT model full efficiency is achieved only when all 32 threads in a warp follow the same execution path. When threads encounter a data-dependent conditional branch (e.g., an `if-else` statement) and take different paths, a control divergence occurs. The hardware handles divergence by serialising the execution of the different paths [58].

For an `if-else` construct, where some threads in a warp take the `if` and the others take the `else` path, the SM will first execute the `if` block for the corresponding threads, while the others are disabled. Once complete, the former threads are disabled and the `else` block gets executed. This means that the total execution time of a divergent branch is the sum of the execution time of both paths. This leads to an underutilisation of `CUDA` cores, as some are idle during each pass. Minimising thread divergence can significantly increase an application's performance if the instruction set of each branch is large [58].

An application's performance is also affected by the latency of global memory access, which can take hundreds of clock cycles. However, GPUs are designed to camouflage this through the massive hardware multithreading, managed by the SM's warp scheduler [58].

An SM is assigned many more active warps than it can execute at any single moment, generating a state of high occupancy. When a warp executes an instruction that results

in a long-latency stall, the warp scheduler does not wait. It instantly switches to another resident warp that is ready to execute. This context switch has no overhead because the full execution state (program counters, registers) for all resident warps is maintained on-chip in the SM's register file. By continuously finding warps that are ready to execute, the scheduler "hides" the memory latency of the stalled warps, keeping the computational units of the SM highly utilised and maximising overall throughput [58].

3.5.3 Memory Hierarchy

Avoiding the bottlenecks of high-latency global memory and achieving high performance with `CUDA` code depend on the explicit management of data across a hierarchy of distinct memory spaces. Each memory type offers a different trade-off between speed, size, scope, and persistence. An NVIDIA GPU has different types of on-chip and off-chip memories.

On-chip memories On-chip memories reside physically on the SM chip, offering extremely high bandwidth and low latency. They include registers and shared memory. Registers are the fastest memory available to a kernel, with access latencies of just a few clock cycles. Each register is private to a single thread, and its lifetime is the duration of the kernel's execution [58].

The `CUDA` compiler automatically places scalar variables declared in a kernel in registers. While fast, the set of registers per SM is finite. A kernel that uses a large number of registers per thread may limit the number of active blocks and warps that can reside on an SM, reducing occupancy and hindering the hardware's ability to hide latency [58].

Shared memory is a high-speed, on-chip Static Random-Access Memory (SRAM) that acts as a programmer-managed scratch-pad or cache. It is allocated per thread block and is accessible by all threads within that block. Its lifetime is also the same as the duration of the kernel's execution. Shared memory enables cooperation and data sharing among threads in a block. Its primary performance role is to cache frequently accessed data from global memory, allowing for data reuse within a block and reducing traffic to the slow, off-chip Dynamic Random-Access Memory (DRAM) [58].

Off-chip memories Off-chip memories are physically located in off-chip DRAM, offering large capacity at the cost of high access latency. There are 3 different levels of off-chip memory: global, local, and constant memory. The global memory is the primary GPU memory space, analogous to the main system's Random-Access Memory (RAM). It is accessible by all threads in all blocks of a grid and can also be read from and written to by the host CPU [58].

Data in global memory persists for the entire lifetime of the application, across multiple kernel calls. It is used for the initial input and final output data between host and device, and for large datasets that cannot fit in the on-chip memories. Accesses are slow, with latencies of hundreds of clock cycles, making global memory the primary performance bottleneck for most memory-bound kernels [58].

Local memory is a space that is logically private to each thread but is physically located in the off-chip DRAM. The compiler uses local memory for two main purposes: for automatic arrays declared within a kernel that are too large for registers and for "register spilling", which occurs when a thread requires more registers than are available. Accesses to local memory are as slow as to global memory and should be avoided through careful code structuring [58].

Constant memory is a read-only memory space that is physically located in off-chip DRAM but is aggressively cached on-chip. It is accessible by all threads in the grid, and the data persist for the entire lifetime of the application. Constant memory is optimised for the access pattern where all threads in a warp read from the same memory address. In this case, the read is serviced by the on-chip cache, and the single value is broadcast to all 32 threads. This provides very high bandwidth for uniform read access patterns [58].

With thousands of threads running concurrently, a traditional transparent, hardware-managed cache system (like a CPU's) would be prohibitively complex and power consuming. Instead, the `CUDA` model provides simpler, more explicit memory primitives and shifts the responsibility of managing data locality and movement from the hardware to the programmer. This makes the hardware more efficient but requires developers to use techniques like tiling or data reorganisation to ensure coalesced accesses if they require

high-performance algorithms [58].

Memory management and optimization The performance of most `CUDA` kernels is limited not by the speed of the ALUs, but by the rate at which data can be fetched from global memory, a problem often called the "memory wall". The primary goal of performance optimisation is to improve the compute-to-global-memory-access ratio, which is the number of floating-point operations performed for each byte transferred from global memory. Two techniques are commonly employed to combat this bottleneck: memory coalescing and tiling [58].

Memory Coalescing Modern DRAM systems achieve high bandwidth by transferring data in large, contiguous blocks or "bursts". The `CUDA` hardware is designed to leverage this by coalescing memory requests from threads within a warp. When all 32 threads in a warp execute a load or store instruction and their memory accesses are to consecutive locations in global memory, the hardware can combine these 32 individual requests into a single, large memory transaction that aligns perfectly with the DRAM's burst capabilities. This is the most efficient way to utilise the available global memory bandwidth [58].

Conversely, an uncoalesced, or "strided" access pattern, where consecutive threads access memory locations a fixed distance apart (e.g., accessing elements down a column in a row-major matrix), is highly inefficient: it forces the hardware to issue multiple, smaller memory transactions, reducing the effective bandwidth and stalling the warp [58].

Tiling While coalescing optimises the efficiency of individual global memory transactions, tiling reduces the total number of transactions by improving data locality through the use of on-chip shared memory. In a tiled kernel, data are partitioned into small blocks (tiles) that fit into a thread block's shared memory, allowing multiple threads to reuse the same data without repeatedly accessing high-latency global memory. This can significantly increase the compute-to-memory-access ratio.

Tiling and coalescing are not independent optimisations; they are actually synergistic. An effective tiling strategy is one where the initial data movement from global to shared

memory is performed with perfectly coalesced accesses [58].

It is worth noting that there are other memory levels and optimisations (e.g., Thread Block Clusters (TBCs) and Distributed Shared Memory (DSMEM)) introduced by NVIDIA in recent years. However, this dissertation did not utilise these features, as the GPUs available for development and testing did not support them. Therefore, they are not discussed here.

3.6 Accelerators in High-Energy Physics

Under the high event-rate conditions foreseen for the HL-LHC upgrade, the ATLAS computing infrastructure must process vastly larger data volumes within strict latency and power constraints. Large-scale data processing in HEP has traditionally relied on clusters of x86_64 CPUs distributed through the Worldwide LHC Computing Grid (WLCG), but the anticipated growth in event complexity threatens to make this model unsustainable. Studies comparing custom hardware and commodity accelerator cards indicate that mixed CPU-GPU-FPGA solutions offer the best performance-per-watt ratio [78]. These ongoing R&D efforts motivate the investigation carried out in this work.

3.6.1 GPU adoption across LHC experiments

The ALICE experiment pioneered large-scale GPU usage through its upgraded O^2 Event Processing Nodes farm, which performs both online reconstruction and data compression entirely on GPUs. During Run 3, 350 servers equipped with 2800 GPUs process Pb–Pb collisions at a 50 kHz interaction rate, achieving real-time throughput with data rates of 1–2 PB/day. This configuration replaced a purely CPU-based farm, reducing hardware and power requirements by nearly an order of magnitude. Without GPUs, roughly eight times as many CPU servers would be required to achieve equivalent throughput [79]. The ALICE deployment remains one of the most successful demonstrations of large-scale GPU integration in an operational HEP experiment.

Similarly, the LHCb collaboration developed Allen, the first fully GPU-based high-level

trigger in a collider experiment. Allen executes real-time event reconstruction, including tracking, vertexing, and particle identification, at the full 30 MHz LHC bunch-crossing rate. Processing a 40 Tbit/s input data stream with roughly 500 GPUs, it reduces the event rate by a factor of 30 to 60 before transferring data to offline storage. The system demonstrated that GPUs can meet the latency requirements of online triggers while maintaining physics accuracy comparable to traditional CPU-based solutions [80, 81].

The CMS collaboration has adopted a heterogeneous strategy combining CPUs and GPUs in both the HLT and offline reconstruction. GPU-accelerated algorithms were first deployed for electromagnetic calorimeter local reconstruction in Run-3 and were implemented in `CUDA` and `OpenMP`. The results demonstrated a substantial reduction in per-event processing time, maintaining trigger latency under HL-LHC conditions with high pile-up [82]. In preparation for the HL-LHC, the CMS software framework is being extended to transparently execute GPU-enabled workflows across heterogeneous computing sites, including Tier-1 and Tier-2 WLCG resources [83]. Resource projections suggest that offloading 25% of reconstruction tasks to GPUs could increase the effective computing capacity by 20–26%, while halving the associated power consumption [84].

In the ATLAS experiment, GPUs are being evaluated in a broader heterogeneous computing program aimed at modernising the reconstruction and simulation software stack, with AthenaMT having been successfully executed on both ARM and GPU architectures via the PanDA (Production and Distributed Analysis System) workflow management system, enabling distributed heterogeneous jobs on the WLCG and commercial clouds [85].

3.6.2 GPU acceleration within ATLAS

A notable example of GPU acceleration in ATLAS is the `FastCaloSim` package, a parametrised calorimeter simulation originally developed to reduce the computational burden of full `Geant4` simulations. The package was ported to `CUDA` and later adapted to performance-portable frameworks such as `Kokkos`, `SYCL`, and `Alpaka`. Benchmarks reported speed-ups of up to 60× relative to single-threaded CPU execution while preserving physical accuracy,

proving that calorimeter simulation can efficiently exploit GPUs [86].

On the reconstruction side, ATLAS has implemented the Topo-Automaton Clustering algorithm, a GPU-parallelizable version of the standard topological clustering used to identify calorimeter energy deposits. Integrated within AthenaMT, this implementation achieved a 3.5–5.5× performance improvement over the CPU version while maintaining physics equivalence [43, 87]. Profiling revealed that most of the remaining runtime is spent on host-to-device data transfers rather than computation, motivating the development of new data models optimised for heterogeneous memory environments.

To address these limitations, the Marionette framework was introduced as a zero-cost abstraction layer between CPU and GPU memory layouts. Written in `C++`, it decouples data structure definitions from memory placement, allowing the same algorithmic interface to operate efficiently across devices. Its initial integration into ATLAS calorimeter reconstruction showed negligible performance overhead versus hand-optimised `CUDA` [88].

Programming models and portability frameworks The growing diversity of accelerator hardware has driven the development of performance portability frameworks that allow a single codebase to target multiple back-ends. Approaches such as `Kokkos`, `SYCL`, `Alpaka`, and `std::par` have been evaluated using representative HEP workloads like ATLAS `FastCaloSim` and Fermilab's tracking test codes. While these frameworks often achieve near-native GPU performance, results depend heavily on memory access patterns, compiler maturity, and data layout choices [89].

The A Common Tracking Software (ACTS) project exemplifies this evolution toward portable, experiment-independent reconstruction software. Its `traccc` R&D line demonstrated that large portions of track reconstruction can be parallelised across CPU and GPU back-ends using `CUDA` and `SYCL`, achieving multi-fold speed-ups while maintaining physics consistency [90].

Emerging trends and challenges Across all experiments, GPUs have proven advantageous in throughput, latency, and energy efficiency. However, several challenges remain:

1. Data transfer overheads between CPU and GPU memory continue to dominate the runtime of reconstruction workflows;

2. Software portability remains an issue due to the coexistence of `CUDA`-specific and multi-back-end implementations;

3. Resource scheduling across heterogeneous WLCG sites must evolve to efficiently utilise mixed architectures;

4. Algorithmic redesign is often required to expose sufficient parallelism, particularly for irregular data structures such as calorimeter cells or tracking hits.

Ongoing software modernisation efforts address these challenges through asynchronous execution models and unified data abstractions. The integration of Marionette within AthenaMT and the heterogeneous execution capabilities of the CMS software framework exemplify this transition toward fully accelerator-aware HEP frameworks.

In this context, this work extends the ATLAS heterogeneous computing program by investigating the feasibility of offloading the calorimeter bytestream decoding step to GPUs. Performing data decoding directly on the device enables calorimeter cell information to remain resident in GPU memory, thereby reducing host communication overhead and paving the way for fully GPU-based clustering and reconstruction workflows.

Chapter 4

Development of the GPU decoders

As previously discussed, under the high event-rate conditions expected at the HL-LHC, the bytestream decoding step of the HLT and offline calorimeter reconstruction may become a non-negligible bottleneck. The GPU decoder developed in this dissertation aims to demonstrate that this can be offloaded to massively parallel architectures while preserving functional equivalence with the established CPU algorithms. The work hereby presented attempts to explore whether the current LAr and TileCal data streams can be efficiently unpacked in parallel. The main goals of the implementation are, therefore:

- **Functional compatibility:** produce identical or nearly identical decoded outputs as the reference CPU algorithms at the bit level.
- **Integration with downstream GPU processing:** operate as a standard algorithm within Athena, compatible with the trigger data flow, but adapt the existing `CaloCellContainer` data model to a GPU-friendly layout.
- **Parallel scalability:** exploit thread-level parallelism to decode multiple cells simultaneously.
- **Maintainability:** implement the GPU logic using modern `CUDA C++` and a modular design to facilitate future code adaptations to the new bytestream layout.

This Chapter presents the design and implementation of this GPU decoder, describing its architecture, data structures, `CUDA` kernels, and validation methodology. Performance results and benchmarking of the prototype are discussed in Chapter 5.

4.1 Algorithm architecture

Implemented in the `CaloGPUbytestreamConversion` class, the decoders developed follow the architecture of the trigger data access services but replace the fragment decoding loops with GPU kernels. The algorithm handles the full decoding pipeline: data retrieval from the read-out buffers, transfer of ROD payloads to device memory, parallel parsing and unpacking on the GPU, and transfer of structured calorimeter channel data back to the host, if required. Conceptually, the pipeline comprises six stages:

1. **ROD selection and bytestream access (host):** determine which calorimeter RODs to unpack and retrieve the corresponding ROD fragments from the bytestream provider.

2. **Payload staging (host):** linearise and pack the selected ROD payloads into contiguous, 32-bit–aligned host buffers, together with the necessary metadata.

3. **Device transfer and workspace management (host → device):** move payload and metadata to device memory. In the latest developed version, slot-specific device workspaces are pre-allocated to avoid per-event `cudaMalloc`/`cudaFree`.

4. **Parallel parsing (device):** a lightweight parser kernel operates on each RODs fragment, scanning the concatenated payload and emitting validated FEB boundaries/offsets for the decoder.

5. **Parallel decoding (device):** a decoder kernel operates on each channel to unpack data into GPU-native structures (energy, time, gain, quality/provenance), mirroring the CPU bit logic.

6. **Results verification (device → host):** if configured to, decoded cells are copied back to the host and compared with the CPU decoded `CaloCellContainer`.

4.1.1 Algorithm setup

The decoder was implemented as a standard `AthReentrantAlgorithm`. Its role is analogous to the `TrigCaloDataAccessSvc` used in the HLT: it forms the interface between the calorimeter readout and the reconstruction tools. But now, the decoding of the bytestream fragments is executed on a `CUDA` device rather than the CPU. The algorithm was designed as a modular pipeline that separates detector-specific logic from the GPU execution and operates transparently within the multi-threaded event-processing model of Athena.

As an `AthReentrantAlgorithm` extension, its `execute()` method will process all inputted events in a loop. However, tools and services must be configured during job setup in the `initialize()` method. Certain information remains constant throughout all events, so they can be retrieved only once. Since the developed class also inherits the `CaloGPUCUDAInitialization` class (which provides the basic infrastructure for appropriate `CUDA` initialisation), the common `initialize()` method is overridden by the `initialize_non_CUDA()` and `initialize_CUDA()` methods.

In `initialize_non_CUDA()`, all service and tool handles are configured through their `retrieve()` calls. This includes `IROBDataProviderSvc`, which supplies the ROD payloads given a list of ROB identifiers (IDs), and `IRegSelTool`, responsible for constructing that list using its region-selector lookup tables. The `GenericMonitoringTool` may also be retrieved at this stage to enable runtime monitoring of the decoded values.

A series of detector conditional data must be read from the `StoreGate`. Those will be introduced throughout this section, but their access keys (`ReadCondHandleKey` and `ReadHandleKey`) must be initialised at setup so the Athena scheduler can construct its data dependency graph.

In turn, the `initialize_CUDA()` method is used to preallocate data objects used during decoding when the property `PrealocateNStructures` is configured with a value

greater than 0. The requested number of structures will be allocated in a thread safe manner, a feature enabled by the `separate_thread_holder` template class.

4.1.2 The event loop

The `execute()` method is used to orchestrate the event specific processing. It allocates, or retrieves from the preallocated buffers, the necessary data structures for the decoding, and launches the LAr and TileCal decoder driver methods. However, some constant, event-independent structures containing metadata are populated and shipped to the GPU exclusively during the first loop iteration. To do so, `std::call_once()` is used to call the `initialize_on_first_event()` method, before the regular event loop starts.

The `initialize_on_first_event()` method is responsible for the retrieval of the ROB IDs that will be used to request the per-event ROD payloads. By accessing the region selector lookup tables, the ROB IDs related to each of the LAr Calorimeter (LArCal) regions are concatenated in an `std::vector`. The same is done for the TileCal regions.

Various look-up tables (LUTs) containing constant metadata about the LAr and TileCal are constructed and transferred to the device at this stage. This avoids repeated host to device transfers and improves the overall algorithm performance. How these LUTs are constructed, and what they are used for, is discussed in Sections 4.2.2.2 and 4.2.2.3.

4.1.3 GPU Cell Container

The "GPU-friendly" structure used to hold the decoded cells on the device is the `CellInfoArr`. It is a minimal container used in the downstream GPU reconstruction algorithms that holds the energy, gain, time and data-quality/provenance of all valid cells in the calorimeter (a total of `NCaloCells`). It uses a SoA layout, as defined in Code 4.1.

This layout stores each field in a contiguous array (one array per attribute) instead of an array of per-cell structures. On the GPU, threads in a warp typically access the same attribute for consecutive cells; SoA makes those accesses naturally coalesced and cache-friendly, reducing global memory transactions and improving throughput.

```
struct CellInfoArr
{
  float          energy[NCaloCells];
  unsigned char  gain[NCaloCells];
  float          time[NCaloCells];
  unsigned int   qualityProvenance[NCaloCells];
  ...
};
```

Code 4.1: GPU cell container structure.

Note that this container is shared among both LAr and TileCal decoders, as they operate in different portions of the container. This is done to avoid an extra merging step after the unpacking is done. After the kernels execute, the complete cell container can be copied back to the CPU for results verification, as is explained in Section 4.5.

4.2 Liquid Argon Bytestream Decoder

As previously stated, the LArCal channels come from N_L different RODs, with $N_L \in [0, 762]$, and each ROD_i, with $i \in [0, N_L)$, is connected to two FEBs (Fa_i and Fb_i). The readout is performed in chunks, with up to 128 cells being handled by one FEB. In the host driver method, given the retrieved list of ROB IDs, one can invoke the `getROBData` method from the provider to populate a `std::vector` of `ROBFragment` objects. The ROD payloads (retrieved with the `rod_data()` method) are copied to a single contiguous array that is then sent to the GPU for processing.

Because of how the LAr bytestream is organised, the header of each FEB must be parsed before the channels can be decoded. That is why the LAr decoding is divided into two steps: one to parse the headers and another for the actual channel unpacking.

Since the bytestream data of different RODs are stored in a single array, and the payloads have varying sizes, information on where they start is necessary for efficient work distribution on the GPU. To achieve this, during the construction of the bytestream array itself, a "rod-offsets" array is filled using a prefix-sum logic. This means that, for ROD fragment i, the rod-offset array at position $i + 1$ will contain the sum of the value

```
tb.rod_offsets[0] = 0;                // initializes first offset
uint32_t *cur_rod = tb.bytestream; /* sets pointer at the start of
                                      the bytestream array */
int i = 0;                            // indexer for the offsets array
for (ROBFragment ptr : rob_frags)  // loops over fragments
{
  uint32_t rod_size = rob_frags[i]->rod_ndata();
  tb.rod_offsets[i + 1] = tb.rod_offsets[i] + rod_size;
  std::memcpy(  // copies payload to the bytestream array
    cur_rod,
    ptr->rod_data(),
    ptr->rod_ndata() * sizeof(uint32_t)
  );
  cur_rod += rod_size;  // updates pointer for next copy
}
```

Code 4.2: Construction of the bytestream and rod-offsets array.

stored at position i and the size of fragment i, as demonstrated in Code 4.2.

As such, the parser kernel workload can be distributed among N_L threads, and the threads can consult where the data they need to access are, as exemplified in Figure 4.1.

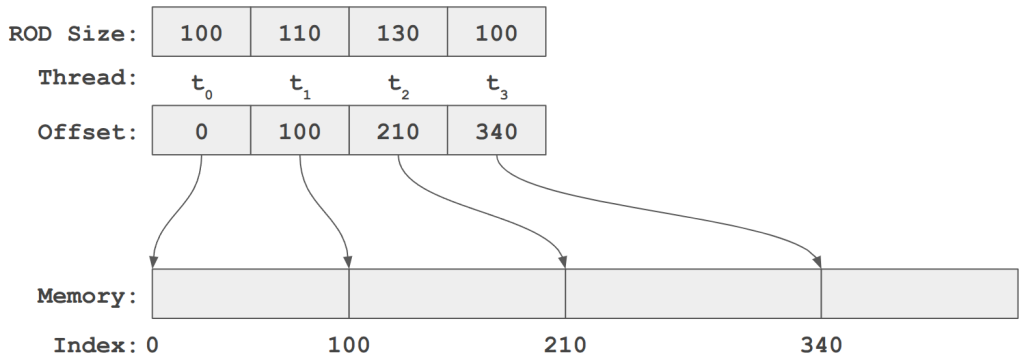

Figure 4.1: Example of the mapping between parser threads and bytestream indexes.

4.2.1 LAr parser kernel

Each FEB within a ROD fragment starts with a 10 32-bit word header (with some entries represented by 16-bit words) that encodes the structure of its payload. The contents of the header are described in Table 4.1. By reading the header entries, the parser kernel

word	Header Field	enum	Description
0	NWTot	0	Number of words in the fragment (32-bits).
1	FEBID	2	FEB compact identifier (32-bits).
2	FEB_SN	4	FEB serial number (32-bits).
3	ResultsOff	6	Offset to the results block.
	ResultsDim	7	Size of the results block.
4	DigitsOff	8	Offset to the Digits block.
	DigitsDim	9	Size of the Digits block.
5	RawDataOff	10	Offset to the raw data block.
	RawDataDim	11	Size of the raw data block.
6	EventStatus	12	Event status (32-bits).
7	NGains	14	Number of gains.
	NSamples	15	Number of samples.
8	FirstSampleIndex	16	Offset to samples.
	FebConfig	17	FEB configuration.
9	InFPGAFormat	18	FPGA format (32-bits).

Table 4.1: Description of the header entries of a FEB. Coloured rows are used for the verification of valid block and payload sizes.

can determine the absolute positions of all inner FEB fields (as in Figure 3.3).

The kernel uses these entries to populate an AoS structure, `LArFebBlocks`, which stores, for each FEB, its identifier, total size, and the absolute offsets of all fields relative to the start of the bytestream array. This structure is defined in Code 4.3. To accommodate all FEBs, the arrays of this structure must be of at least $M_L = 2N_L$ ($M_L \in [0, 1524]$).

A FEB payload can be divided into 3 data blocks: Results, Digits, and Raw data. If

```
struct LArFebBlocks {
  int size[ML],         id[ML],              base_offset[ML];
  int gain_offset[ML], tq_mask_offset[ML], digits_mask_offset[ML];
  int radd_offset[ML], energy_offset[ML],  sum_offset[ML];
  int tq_offset[ML],    digits_offset[ML],  raw_data_offset[ML];
};
```

Code 4.3: LArFebBlocks structure definition.

one knows a field size, one can assume its information is in the payload if the size of the block it is contained in is equal to or greater than the accumulation of previous field sizes and the size of the field itself. For example: the TQ mask is always 4 words long, and it comes right after the gain field, which has 8 words in total; this means that, for the TQ mask to be contained in the payload, the total size of the Results block must be of at least 12 words, so it can accommodate both fields (8+4 words). Figure 4.2 represents this logic.

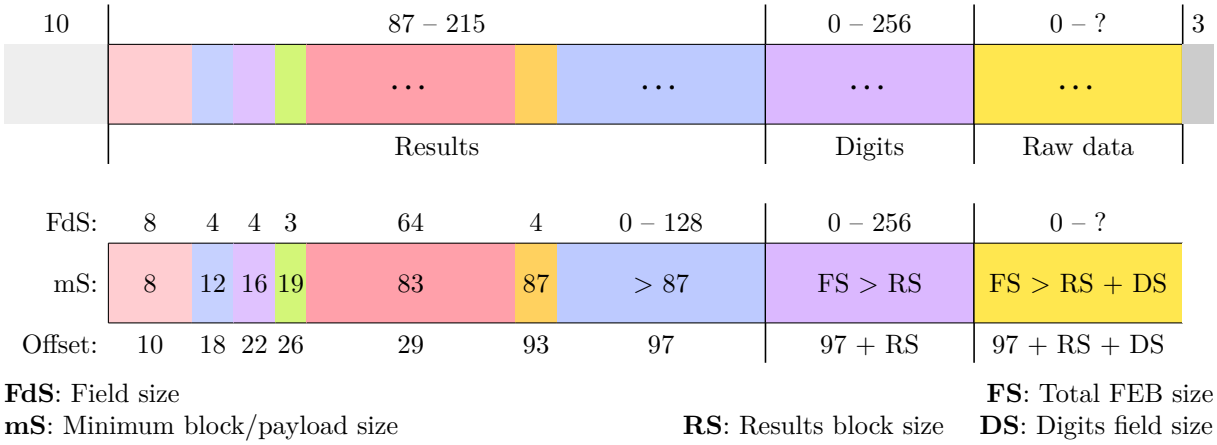

Figure 4.2: FEB blocks and field offsets. The top diagram displays the division of the FEB payload into the data blocks. The bottom diagram shows the inner field sizes, their offset relative to the start of the FEB payload, and the minimum block/payload size required for each field to be present. Block size is used to verify the presence of the Results fields. The total FEB size is used to verify the presence of the Digits and Raw data blocks.

Iterating through multiple FEBs After the first FEB Fa is parsed, the kernel must determine whether there is enough space left in the ROD fragment to contain Fb. This verification is handled by the function `next_feb()`, which operates as follows:

1. The size of Fa (`feb->size[fi]`) is summed to a constant `s_feb_middle_header_size`, which corresponds to the 7-word header separating consecutive FEBs.

2. This value is subtracted from the remaining size of the current ROD to obtain the number of words left in the fragment.

3. If there is not enough space to contain a new header, the function returns `false`, and the iteration stops.

4. Otherwise, the first word of the next header is read (the `NWTot` field) to check if the declared FEB size fits in the remaining space. If it does, Fb is present in the payload: the index `fi` is incremented, default offsets are initialised, and the Fb identifier and size are stored.

This process ensures that both FEBs within the fragment are discovered and indexed. As each thread t_i parses one ROD, the final FEB indexes in the `LArFebBlocks` structure will be $Fa_i = 2\,t_i$ and $Fb_i = 2\,t_i + 1$. Code 4.4 shows this procedure. The function `set_feb_offsets()` parses the header and verifies the payload sizes, as previously summarised.

By pre-computing the offsets of all FEB fields on the device, the decoder kernel can access each calorimeter channel directly through these indices, avoiding repeated header parsing and enabling fine-grained parallel decoding at the channel level.

```
int fi = tidx * 2;
do {
    set_feb_offsets(bytestream, fi, feb);
} while (next_feb(bytestream, rod_offset, rod_size, fi, feb));
```

Code 4.4: Simplified logic of FEB iteration within a ROD. `fi` is incremented inside the `next_feb()` function if a second FEB is identified.

4.2.2 LAr decoder kernel

The decoder kernel operates strictly in the Results data block. It is launched with one thread t_j for each raw LArCal channel contained in the bytestream. As there are $C = 128$ channels per FEB, and 1562 FEBs in the whole detector, the total number of threads should be $LC = 195,072$ and $j \in [0, LC)$. The index of the FEB to which a channel belongs can be retrieved by calculating $f_i = t_j \div C$, and the cell index can be calculated as

$c_k = t_j \bmod C$. With this, thread t_j operates over the offsets stored in the `LArFebBlocks` structure at index f_i and the per-channel field offsets are calculated based on c_k.

4.2.2.1 Field decoding

Energy decoding The energy E of channel c_k is encoded in a 16-bit word, divided into 3 fields (Figure 4.3). The lowest 13 bits store the base energy value b, bit 13 (zero-indexed) encodes the sign of the energy, and the two most significant bits define the scaling range r. These fields represent the measured energy in a compact logarithmic format.

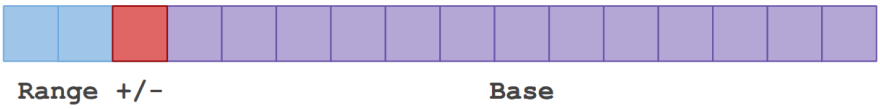

Figure 4.3: Bit-level representation of the LAr calorimeter encoded energy format.

When retrieving the 16-bit energy value of a given channel, the decoder must account for the endianness of the calorimeter bytestream. Each 32-bit word contains the energies of two consecutive channels, packed in little-endian order: the lower 16 bits correspond to the first (even) channel, and the upper 16 bits correspond to the next (odd) channel. Because the decoder accesses data as 32-bit words, the correct 16-bit half must be selected depending on whether the channel index is even or odd.

During decoding, the base value b is first extracted from the lower 13 bits. The range field r determines the exponent applied to the base, such that the final magnitude is obtained by multiplying b by 2^{3r}. If $b = 0$ but $r \neq 0$, the base is replaced by 0x2000 (8192) before scaling. The sign bit in position 13 is then evaluated to determine whether the result should be negative.

The final energy is stored as a signed floating-point value in the `CellInfoArr` container. Additional BCID correction may be applied later, as discussed in Section 4.2.2.3.

Gain decoding The gain field occupies a total of 256 bits with each calorimeter channel encoded using two bits. Since $256 \div 2 = 128$, this field contains the gain information for

```
uint32_t gain_word = get_bytestream_value<uint32_t>(
  bytestream,
  feb->gain_offset[fi],
  (ck >> 4)
);
gain = (uint32_t)((gain_word >> ((ck & 0xf) << 1)) & 0x3);
gain = raw_to_offline_gain(gain);
       /* 0 -> 0, 1 -> 2, 2 -> 1, 3 -> 0, default -> 0xFFFFFFFF */
```

Code 4.5: LAr gain decoding.

128 channels per FEB, arranged in groups of 16 channels per 32-bit word. In other words, each 32-bit block stores the gain values for 16 consecutive channels.

To obtain the gain associated with a specific channel c_k, one first determines which 16-channel block it belongs to. This is done by computing $\left\lfloor \frac{c_k}{16} \right\rfloor$, and using this index together with the base offset stored in `feb->gain_offset[fi]` to get the corresponding 32-bit word via the function `get_bytestream_value<uint32_t>()`.

Within this 32-bit block, each channel is represented by two bits, so the position of channel c_k inside the block is given by c_k mod 16. Since each position occupies exactly two bits, the gain for channel c_k corresponds to bits $2 \times (c_k \bmod 16)$ and $2 \times (c_k \bmod 16) + 1$ of the 32-bit word.

This value represents a hardware-specific encoding of the gain, which is then translated into the offline gain convention through the function `raw_to_offline_gain()`. This conversion applies the mapping, with invalid codes defaulting to `0xFFFFFFFF`. Code 4.5 shows the function snippet that executes this process.

Time and quality decoding Each FEB contains up to 128 calorimeter channels, and for each channel, the bytestream may optionally include time (τ) and quality (q) information. The presence of these values is indicated by the dedicated TQ mask, which is a 128-bit field where each bit corresponds to one channel. If the c_k-th bit of the mask is set, then channel c_k has time and quality data stored in the TQ field.

The TQ field is organised as a list that stores the (τ, q) pairs of all channels whose bits are set in the mask. Consequently, the size of this field varies between events, and

the position of a given channel's data cannot be computed from a fixed offset. Instead, to access the correct position for channel c_k, it is necessary to count how many channels before c_k also have their bits set in the TQ mask. The resulting count corresponds to the index of that channel within the TQ field list.

This step is illustrated in Figure 4.4. The darkest blocks represent the channel c_k being processed. Medium dark blocks in the TQ mask highlight the set bits before c_k. Blocks with set bits to the right of the wide marker should be counted towards the final TQ index. In the leftmost scheme, there is a single set bit in the mask before the c_2 bit, so the TQ information of c_2 sits at position 1 of the TQ list. On the right, there are 3 set bits before c_6, so its position on the TQ list must be 3.

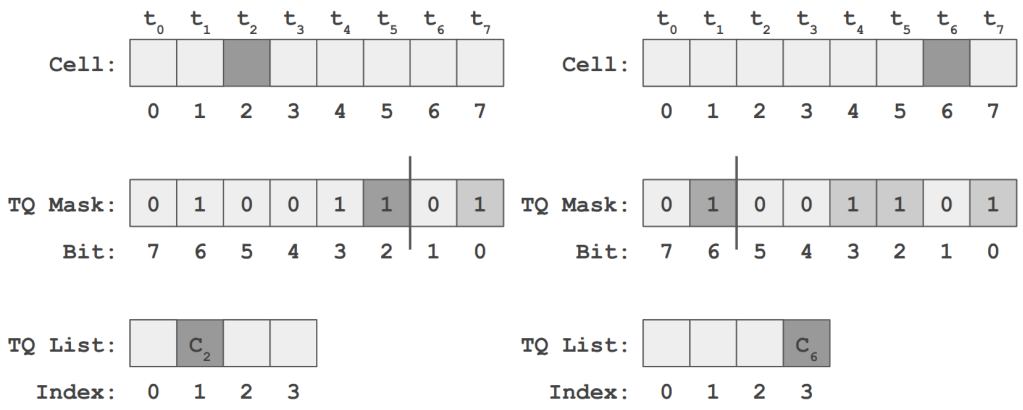

Figure 4.4: Simplified example of how the TQ decoder function uses the mask to determine the bytestream index of a channel's TQ.

In the GPU decoder, this operation is implemented by the `get_TQ_index()` function. It first sums the number of set bits in all 32-bit words of the mask that precedes the one containing the bit of interest, using the intrinsic `__popc()` to count bits efficiently. Then, it counts the active bits that appear before c_k within its own 32-bit word. The sum of both results gives the position of the channel in the TQ list (multiplied by two, since each entry contains both time and quality values).

Each TQ list entry contains the (τ, q) pair packed into 32-bit words, each with 16-bits. The extracted time value is stored as a floating point, as its signed integer value

```
enum LArProvenance : uint16_t {
  PEAKAVG        = 0x1,    PEAKPARABOLA   = 0x3,  PEAKCUBIC = 0x4,
  PEAKOFC        = 0x5,    PEAKTILEINFO   = 0x6,  PEAKNN    = 0x7,
  PATCHED        = 0x8,    RAMPCONST      = 0x10, RAMPDB    = 0x20,
  PEDSAMPLEZERO  = 0x40,   SCPASSBCIDMAX  = 0x40, PEDDB     = 0x80,

  DEFAULTRECO    = PEAKOFC | RAMPDB | PEDDB,

  ITERCONVERGED  = 0x0100, SCTIMEPASS     = 0x0200,
  SATURATED      = 0x0400, MASKED         = 0x0800,
  DSPCALC        = 0x1000, QTPRESENT      = 0x2000,
};
```

Code 4.6: LAr Provenance bitfield definition.

is converted to nanoseconds by dividing it by 100. The quality is used to prepare two values that are stored in the `CellInfoArr` container: the quality value (`iquality`) and the provenance bit-field (`iprovenance`).

The `iquality` variable stores the 16-bit raw quality value returned by the `decode_time_quality()` function. The `iprovenance` variable, on the other hand, encodes metadata about how the channel energy, time, and quality were reconstructed, based on the definitions of the `LArProvenance` enumeration structure (see Code 4.6) [91].

In the GPU decoder, `iprovenance` is initialised with the value of `DSPCALC`, marking that the energy originates from the online DSP calculation. If the channel also contains valid time and quality information (that is, if the decoded quality is non-negative), bit `QTPRESENT` is added to the provenance word. The `iquality` variable then receives the decoded quality value, while channels lacking TQ data retain a zero quality.

Finally, both fields are combined into the `QualityProvenance` structure (a 32-bit word in which the lower 16 bits contain the quality and the remaining bits contain the provenance) and stored in the output container.

4.2.2.2 LAr cell indexing

The way the cells are ordered in the bytestream is different from the one used by subsequent algorithms. Calorimeter cells are identified by a unique number. In the ATLAS detector, there are online and offline IDs. The ID packed in the ROD payloads is the

online (also called hardware) ID, which serves to identify the electronics of the FEB and the channel of a read-out. However, for the HLT and offline systems, using the offline ID makes more sense, as it is related to the (η, ϕ) coordinate of the cell.

The hardware ID can be retrieved by consulting the `LArOnlineID` database from the Detector Store. To do so, both the channel's FEB ID and raw channel index are used as a key. This database also enables the conversion from channel hardware ID to hardware identifier hash. Now, from the Conditions Store, with the `LArOnOffMapping` database, one can retrieve the offline hash using the aforementioned hardware hash. The offline hash is what orders the final `CaloCellContainer`. This database maps the invalid cells to an invalid container position, and "brings" the good cells to the front.

However, some LArCal channels may be faulty. To identify problematic channels, the Condition Store `LArBadChannelCont` database is used. Given the offline ID of the cell, the database returns a `LArBadChannel` object that allows the verification of a series of possible issues. If a cell is flagged as unstable, has a high noise level, or an unidentified problem, the cell's channel is considered "bad", and its readout should be zeroed.

Since porting all of these databases and their functionalities to GPU is currently not a priority, a solution to make this information available for device kernels is to create a LUT that maps the FEB ID and channel pair to the final container index.

The FEB IDs are usually 64-bit integers in offline software, but in the bytestream, their compact, 32-bit form is present. An issue is that these IDs, even in compact form, are very large numbers (higher than 900 million), so using them to create a LUT directly is not a viable option.

However, after inspection of the FEB IDs, one can notice that, from their 32 bits, only 11 of them change. If these bits are extracted, one can get a number $h \in [0, 1933)$. This is then used to create a table (`cell_map`) with 1933 rows and 128 columns (one for each channel). The final LUT, therefore, is constructed as a one-dimensional array with 1933×128 slots. Only 1524 of these rows are valid, so some storage is still wasted.

An option to avoid this would be to first map h to the FEB in a small table, and then use the calculated table index to access a map that, in this case, would occupy 1524×128

slots. However, this would create another level of indirection, being less efficient than the direct access given by the first option.

FEB Id	FEB Id$_{(bin)}$	$h_{(bin)}$	$h_{(dec)}$
0x3b988000	00111011100110001000000000000000	11100110001	1841
0x3b990000	00111011100110010000000000000000	11100110010	1842
0x3b998000	00111011100110011000000000000000	11100110011	1843
0x3b9a0000	00111011100110100000000000000000	11100110100	1844
0x3b9a8000	00111011100110101000000000000000	11100110101	1845

Table 4.2: Examples of mapping between FEB ID, its binary representation, and the extracted bits h in binary and decimal.

Therefore, when the decoder kernel processes thread t_j, the LUT position of channel c_k from FEB F_i is stored at index $C_j = \texttt{cell\_map}[h_i \times 128 + c_k]$. All `CellInfoArr` container values are zeroed once the decoder kernel starts, and the mapping already points invalid or bad channels to an invalid container position. Before the field specific decoding starts, threads that point to an invalid position are interrupted. Thus, no extra verifications are necessary: valid channels are saved accordingly, and the remaining ones are ignored.

4.2.2.3 BCID Offset corrections

The BCID mechanism associates each calorimeter pulse sample with the 25 ns LHC bunch clock. Because the pulse shaping time of the LAr readout electronics extends over several consecutive bunch crossings, the reconstructed energy depends on the precise time alignment between the true pulse maximum and the digitised samples.

Figure 4.5 shows this effect using a simplified simulation of a shaped pulse, sampled at 25 ns intervals. Each blue point corresponds to a digitised sample, and each group of samples belongs to a distinct bunch crossing, identified by its BCID. The reconstructed transverse energy (E_T) varies with the time offset between the injected signal and the sampling clock. To ensure all pulses are reconstructed consistently, per-channel correction factors must be applied to compensate for these timing-dependent energy biases [92]. These correction factors are stored in the `CaloBCIDAverage` Conditions database.

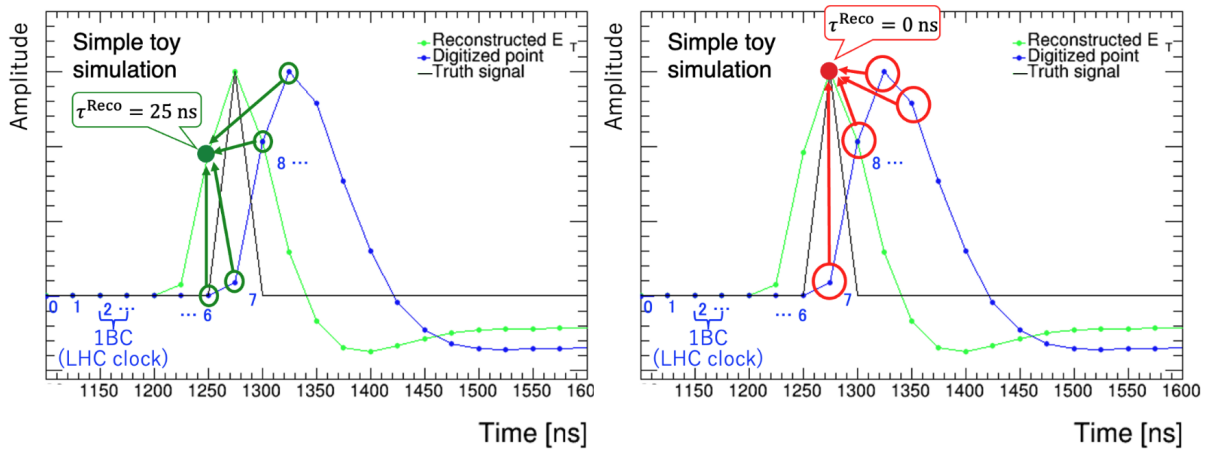

Figure 4.5: A toy simulation of a pulse. An injected signal is shown in black, digitised data points are shown in blue, E_T and τ reconstructed from blue points using optimal filtering are shown in green. The blue numbers represent BCIDs. Source: Furukawa [92].

In the GPU decoder, these per-channel offset corrections (OCs) values are incorporated through a small LUT. The goal is to enable the application of BCID-dependent corrections directly within the bytestream decoding kernels, while keeping memory usage and data transfer overhead minimal. An indexing LUT is constructed once at initialisation and remains static throughout the run, whereas the numerical correction values are updated for every event.

LUT construction As explained in Section 4.2.2.2, each digitised sample in the LAr bytestream is associated with a unique hardware identifier, which can be translated into an offline calorimeter cell index C_j through the cabling map. However, groups of hardware channels share identical calibration constants due to geometric and electronic symmetries.

The `LArMCSym` Conditions database defines these equivalences by providing, for each channel, a corresponding "symmetric" identifier (`symID`). Because the number of symmetric identifiers (1.835) is much smaller than the total number of LAr cells (182.468), the correction table can be indexed by `symID` rather than by channel, which significantly reduces the size of the structure transferred to the GPU.

During initialisation, the decoder iterates through all channels known to the cabling service and constructs two host-side structures:

- Symmetric ID index map (`m_lar_sym_map_set`): In the first iteration, each unique symmetric identifier is assigned a dense integer index s in the range $[0, S)$, where $S = 1835$ is the total number of distinct `symID`s. This map set provides the correspondence between the hardware identifiers used in the bytestream and the compact indices that will be used to access the corrections array.
- Symmetry index LUT (`sym_map`): A second pass through all valid channels maps the cell C_j to s by consulting `m_lar_sym_map_set`. Channels without a valid symmetry entry are assigned a default index equal to S, which corresponds to a zero correction.

The `sym_map` is transferred to device memory once and reused in subsequent events.

Updating the BCID Corrections At every event, the decoder retrieves the `CaloBCIDAverage` and `LArOnOffIdMapping` Condition objects to update the array of correction values, `bcid_corr`. For each symmetric identifier in `m_lar_sym_map_set`, the corresponding offline identifier is obtained through the cabling service and its average correction is fetched via `avg->average(id)`. The `bcid_corr` array is transferred to the device just as the bytestream payload.

Access on the GPU In the `decode_bytestream()` kernel, using the `m_LAr_cell_idx_map` structure, C_j is restored. If offset correction is enabled, the thread then retrieves the BCID correction value through a two-step lookup: $corr = \texttt{bcid\_corr}[\texttt{sym\_map}[C_j]]$. The correction is subtracted from the decoded energy before writing to the output structure. Channels mapped to the default symmetry index simply read the zero entry, ensuring consistent behaviour for masked cells. Figure 4.6 exemplifies the procedure.

4.3 Tile Bytestream Decoder

The decoding of TileCal calorimeter data on the GPU is organised as a pipeline that mirrors the logical structure of the bytestream. A host-side phase prepares compact, run-dependent lookup information so that device kernels can operate without consulting the

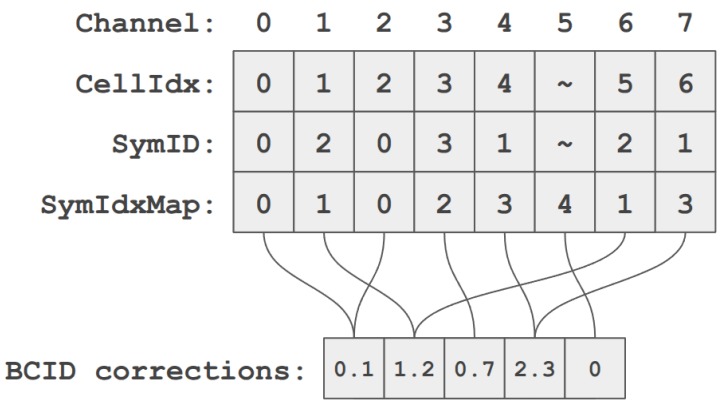

Figure 4.6: Simplified example of the mapping between symmetric identifier and BCID correction.

full Athena geometry and cabling services at runtime. A parser kernel then scans each ROD payload to locate the sub-fragments and extract drawer-level data quality, and a raw-channel kernel unpacks the per-channel words into amplitude, time, gain, and quality. The final merging step, which combines the two PMT signals into reconstructed cells, is under active development and is thus not included in the performance results shown here.

The input to this pipeline is the set of ROB fragments produced by the TileCal readout system. On the host, the bytestream is retrieved with `getROBData` into a `std::vector` of `ROBFragment`s and its payloads are packed into a single contiguous array for transfer to the device. Alongside the payload, two auxiliary structures are copied: an array of ROD offsets that delimit the fragment boundaries within the flat buffer, and a ROD-indexed table of BCIDs used during drawer-level data quality verification.

The remainder of this section details, in order, how the kernels map the bytestream drawers and channels, the construction of the lookup tables that capture the online–offline mapping and drawer metadata, the operation of the parser kernel that discovers sub-fragment locations and computes drawer quality masks, and the raw-channel decoder that applies the bit-level extraction and scaling rules to produce physical quantities.

4.3.1 Indexing and detector data model

The TileCal bytestream elements are mapped to the parallel execution model of the GPU as follows. At the top level, there are N_T RODs, with $N_T \in [0, 128)$. These RODs transmit the data of two drawers, denoted Da_u and Db_u, with $u \in [0, N_T)$. Each drawer, in turn, contains up to $P = 48$ raw calorimeter channels p_w, with $w \in [0, P)$, and one DQs block that summarises its integrity status.

Fragment identifiers Each TileCal fragment is labelled by a 16-bit identifier F_{id} that combines the read-out system number (ROS) in the high byte and the drawer number in the low byte. ROSs are numbered $R \in [1, 4]$, and drawers $D \in [0, 64)$, resulting in 256 unique fragment identifiers in the whole TileCal. Because of this byte-wise encoding, the identifiers are not contiguous in memory: for example, the range 0x0100 – 0x013F (256 – 319) corresponds to the first read-out system, while 0x0200 – 0x023F (512 – 575) corresponds to the second.

To avoid sparse GPU memory access patterns, a dense compact index $d \in [0, 255]$ is derived from the hardware identifier F_{id}, with the forward and inverse mappings given by:

$$d = 2^6\left(\left\lfloor \frac{F_{id}}{2^8} \right\rfloor - 1\right) + \left(F_{id} \bmod 2^6\right), \qquad F_{id} = 2^8\left(\left\lfloor \frac{d}{2^6} \right\rfloor + 1\right) + \left(d \bmod 2^6\right).$$

This mapping preserves a one-to-one correspondence between the hardware numbering and the compact device-side index and allows all drawer fragments to be stored in contiguous arrays.

Thread mapping Two kernels operate on these indices. The parser kernel is launched with one thread per ROD. If u denotes the parser thread index, the corresponding drawers are addressed by $d_u^a = 2u$ and $d_u^b = 2u + 1$ so that both drawers belonging to a ROD are handled by the same parser thread.

The raw-channel decoder kernel is launched with one thread per hardware channel. With M_T=$2N_T$ total drawers and P=48 channels per drawer, the total number of raw

channels is $TC{=}M_T \times P{=}12{,}288$. For a decoder thread index $v \in [0, TC)$, the `TileDrawers` index and local channel number are computed as $d_v{=}v \div P$ and $w_v{=}v \bmod P$.

4.3.2 Lookup tables for Tile decoding

Before the bytestream fragments can be decoded on the GPU, a set of static lookup tables is constructed on the host. These tables provide the device with compact, run-dependent metadata that links online hardware identifiers to the corresponding offline calorimeter cells and records the per-drawer electronics configuration. The tables are built once, during the processing of the first event, and remain valid for the duration of the run.

Services and conditions access The function `initialize_tile()` performs this construction step. It retrieves from the Detector Store the identifier helpers (`TileID` and `TileHWID`), the detector description manager (`TileDetDescrManager`), the cabling service (`TileCablingService`), and the run-aware mapping object `TileHid2RESrcID`. Together, these services provide the translation between the online electronics numbering and the offline identifiers used by the reconstruction software.

Fragment mapping Each TileCal fragment is uniquely identified by the code F_{id} defined previously. The fragments can, however, have different online (F_{on}) and offline (F_{of}) identifiers. By iterating over all offline IDs, the conditions object `TileHid2RESrcID->getDrawerInfo(`F_{of}`)` returns a small vector that includes F_{on} and a drawer-type tag that indicates whether the drawer follows the legacy or demonstrator electronics layout. F_{on} is converted to the dense index d_{on}, and F_{of} to d_{of}.

Using this information, two main lookup tables are produced. The first, `frag_map[`d_{on}`]`, performs the online-to-offline translation by storing the canonical fragment identifier F_{of} corresponding to each online code. This allows the GPU parser to recover, in constant time, the correct fragment identifier when reading the low 16 bits of a section header. The second, `frag_drawer_type[`d_{of}`]`, records the drawer type for each fragment and is later used by the decoder kernel to apply the appropriate channel ordering.

Channel and cell mapping For each drawer, the procedure iterates over the 48 possible hardware channels p_w and uses the cabling service to convert the triplet (R, D, p_w) into the final offline cell hash C_v used to index cells in the `CaloCellCont`. These hashes are stored in `m_tile_cell_map[`C_v`]`, enabling the later merging step to associate decoded PMT signals with their offline calorimeter cells.

Demonstrator permutations Because the demonstrator drawers employ a different channel wiring scheme, two additional 48-element permutation tables are constructed on the host: `LB_demo_channel` for the long-barrel and `EB_demo_channel` for the extended-barrel modules. These tables remap the local channel numbering into the demonstrator ordering. All lookup tables are then copied to global device memory and remain available to the kernels for the duration of the job.

4.3.3 Tile parser kernel

The parser kernel discovers, for each ROD payload, the positions and sizes of the TileCal sub-fragments (as in Figure 3.5) and unpacks the drawer-level DQ words into a single quality mask. The results are written to a device-side metadata container (`TileDrawers`) organised as a structure of arrays to enable coalesced access by subsequent kernels. The container has one slot per drawer, as presented in Code 4.7.

The parser kernel is launched with one thread u per ROD. As stated, the two drawers read out by a ROD are addressed at $d_u^a = 2u$ and $d_u^b = 2u{+}1$ in the fragment table. At first, the thread initialises both drawer slots. Sizes are set to zero, the base offsets are marked with an invalid sentinel, the quality masks are cleared, the per-drawer `channel_unit` is set to zero, the `raw_channel_unit` defaults to ADC counts, and the flag is initialised to enforce conservative behaviour in the absence of valid blocks.

The thread then reads the headers contained in the contiguous bytestream interval, delimited by the precomputed ROD offsets. A leading sentinel word `0xff1234ff`, when present, is skipped. The implementation enforces a per-section size guard by requiring each section to contain at least three words.

```
struct TileFlagSlot {
  uint32_t bits;
  enum : uint32_t { //         Flag bits layout
    Found         = 1u << 0,  MissingDQ        = 1u << 1,
    OF2Correction = 1u << 2,  CorrectAmplitude = 1u << 3,
  };
  ... // host/device methods to set and verify the flags.
};

struct TileDrawers {
  int size[MT], id[MT], base_offset[MT];
  uint16_t dquality[MT];
  int channel_unit[MT];
  TileRawChannelUnit raw_channel_unit[MT];
  TileFlags flags[MT];
};
```

Code 4.7: Definition of the `TileDrawers` metadata structure.

Each sub-fragment comprises a size word followed by a header word. The low 16 bits of the header carry the electronics' online fragment identifier F_{on}; the parser converts this identifier to a dense online index $d_{on} = \texttt{index\_from\_id}(F_{on})$ and then translates it to the offline dense index d_{of} via the lookup table $\texttt{frag\_map}[d_{on}]$.

The implementation pins the first distinct offline fragment seen in the ROD to d_u^a and the second to d_u^b, and selects the destination slot accordingly for all subsequent sections belonging to the same drawer. The upper 16 bits of the header encode the block type and a set of unit and data-type flags used to interpret the payload.

Three block types are relevant: digits (`0x1`), raw channel (`0x4`), and DQ (`0xa`). Digits sections are ignored by the current algorithm.

Raw-channel block When a raw-channel section is encountered and the destination drawer has not yet been marked as found, the parser records the absolute base offset of the section within the flat bytestream buffer and its size, sets the `Found` flag, and decodes the unit and correction flags from the header.

In particular, the header bits select the online unit and the raw-channel unit to be used by the decoder, and two flags indicate whether an optimal filtering correction is expected and whether the on-board DSP has already corrected the amplitude. The implementation

distinguishes two classes of data types: if the encoded type is smaller than three, it means the decoder is dealing with real detector data, so the correction flags and units are set directly from the header; otherwise, the data comes from simulation datasets, so the code clears the "missing DQ" flag and resets the raw-channel unit to the per-drawer `channel_unit`.

Data quality block When a DQ section is found, the thread invokes a routine that validates the section and aggregates the 12 16-bit quality words into a per-drawer mask. It first checks that the declared section size is at least nine words; otherwise, a neutral mask is returned. It then extracts the DQ header's fragment code and normalises the BCIDs format by forcing the most significant bit of the ROB BCID. The 12 words defined in Code 4.8 are read in sequence and tested to update the mask.

```
enum TileDQWords {
    GlobalCRC    = 0, DspBCID      =  1, BCIDChecks   =  2,
    MemParity    = 3, SingleStrobe =  4, DoubleStrobe =  5,
    HeadFormat   = 6, HeadParity   =  7, SampleFormat =  8,
    SampleParity = 9, FEChipMask   = 10, RODChipMask  = 11,
};
```

Code 4.8: Tile data quality words.

A set of early-return conditions immediately saturates the mask (`0xffff`) when critical errors are detected, namely: `GlobalCRC` indicates a corrupted fragment checksum; `DspBCID` means the word differs from the normalised ROB BCID; `BCIDChecks` means the word reports BCID mismatches in all four motherboards.

If these checks pass, the routine proceeds to combine parity and format errors from the `HeadFormat`, `HeadParity`, `SampleFormat`, and `SampleParity` words into the mask.

When none of these errors are present, the front-end chip mask (`FEChipMask`) is expanded to a per-Data Management Unit (DMU) representation. For extended-barrel drawers, a small remapping and one-bit shift are applied to account for unpopulated positions. The inverted result is merged into the mask to flag disabled or missing chips. Finally, a non-zero `RODChipMask` value contributes a unit flag, completing the aggregation.

The resulting 16-bit mask encodes the drawer's overall integrity state and is stored in the drawer table. It is later used by the reconstruction kernels to gate amplitude and timing corrections, or to veto unreliable channels entirely.

4.3.4 Raw-channel decoder kernel

Once all fragment metadata have been resolved by the parser, a second kernel decodes the individual raw calorimeter channels. This kernel is launched with one thread per hardware channel. For a thread index v, the corresponding drawer d_v and channel w_v are determined as explained earlier. Here, d_v is used to index the `TileDrawer` metadata container, and w_v identifies the local channel within that drawer.

Each thread first checks whether its assigned drawer has been marked as found by the parser. If not, the thread returns immediately. The absolute position of the fragment section within the global bytestream buffer is obtained from `frag->base_offset[`d_v`]` (pointing to its header h), and the corresponding 16-bit online fragment identifier is read from the header. This identifier is converted to the canonical offline fragment identifier through the same dense mapping used by the parser.

The kernel then computes the word position of the current channel within the fragment block as $c_v = h + 2 + w_v$, where the two leading words correspond to the section header. For demonstrator drawers, the channel index is remapped through the appropriate 48-element permutation table (`LB_demo_channel` for long-barrel and `EB_demo_channel` for extended-barrel modules) according to the `drawer_type` lookup.

Each channel word encodes four quantities within a single 32-bit field: gain, amplitude, time, and quality. These are extracted through simple bitwise operations:

$$g = (w \gg 31)\ \&\ 0\text{x}1, \qquad Q = ((w \gg 0)\ \&\ 0\text{x}1F)$$
$$A = \frac{((w \gg 16)\ \&\ 0\text{x}7FFF) - \mathrm{amp}_{\mathrm{offset}}[g]}{\mathrm{amp}_{\mathrm{factor}}[\mathrm{unit}]}, \qquad t = \frac{((w \gg 5)\ \&\ 0\text{x}7FF) - 1024.0}{16.0}.$$

The amplitude decoding relies on two constant lookup tables that are transferred to the device constant memory at initialisation. The first table, denoted by amp$_{\mathrm{o}}$ffset, contains

the per-gain offset values that must be subtracted from the raw amplitude code before applying the scaling. Each entry corresponds to one of the two gain modes used in the calorimeter readout: low gain and high gain. The second table, amp$_{\text{f}}$actor, defines the conversion factors that translate the corrected ADC code into physical energy units. Four distinct scaling factors are provided to account for the different operational configurations of the readout system. Their definition is as follows:

$$\text{amp}_{\text{offset}} = \{512,\, 2048\}, \qquad \text{amp}_{\text{factor}} = \{16,\, 32,\, 32,\, 1/32\}.$$

If the channel is encoded in non-ADC units and has high gain ($g = 1$), the amplitude is further normalised by a factor 1/64. The time encoding uses an offset of 1024 and a scale factor of 16, yielding a granularity of 1/16 of a time tick and a range centred around zero. The quality field occupies five bits, with a threshold value of quality_threshold = 15.999, which distinguishes between the magnitude bits and an additional flag bit used by the reconstruction algorithms.

Once the four quantities have been extracted, the decoder performs a sequence of validity checks and corrections to ensure that the resulting values are physically meaningful and consistent with the CPU TileCal reconstruction workflow.

Data-quality and masking logic The TileCal drawer DQ word marks groups of three consecutive channels associated with individual DMUs. For the current channel index w_v, the kernel inspects the bit corresponding to the group $\left\lfloor \frac{w_v}{3} \right\rfloor$ to determine whether this DMU is flagged as problematic. In parallel, the locally decoded quality value DQ is compared against the predefined threshold quality_threshold. A channel is considered valid only if $Q <$ quality_threshold and DMU group is not flagged as bad. If either condition fails, the channel is masked, and both its reconstructed energy and time are set to zero.

Calibration-unit handling The raw amplitude value is initially expressed in one of several online hardware units. The field `raw_channel_unit[`d_v`]` indicates whether the

encoded channel is already in MeV or requires additional calibration. If the value is not provided in `OnlineMegaElectronVolts` or `MegaElectronVolts`, the decoder marks the channel for recalibration. While the current GPU prototype does not implement the full recalibration path, the kernel issues a warning whenever such a case occurs.

Time-based amplitude correction For channels that pass the masking criteria and belong to fragments configured to apply Optimal Filtering (OF) corrections, the amplitude may require a time-dependent adjustment. The kernel first verifies that the reconstructed time lies within the allowed range $t_{\min} \leq t \leq t_{\max}$. Channels outside this interval are discarded by zeroing both energy and time.

For in-time channels, and provided the reconstructed energy exceeds a configurable minimum, the amplitude is corrected using the function that implements two types of correction curves depending on the drawer configuration:

- a piecewise quadratic correction used for "OF2-mode" drawers; or
- a fourth-degree polynomial correction used for "OF1-mode" drawers.

These corrections compensate for the intrinsic time-dependent bias of the digital filters and ensure consistency with the offline TileCal calibration chain.

Quality-bit assignment The floating-point quality value is compressed into a single byte quantity before being stored. The kernel takes the absolute value of the decoded quality, truncates it to an integer, and saturates it at 255: $\texttt{qual} = \min(255,\ |\texttt{quality}|)$.

In addition, the kernel computes a compact quality flag, `qbit`, using the helper function `qbitfun()`. This flag encodes whether the reconstructed pulse time lies within the configured acceptance window. Channels with $\texttt{time} = 0$ are assigned a null quality code, whilst non-zero pulses receive either a combined amplitude-and-time bit mask when the time falls inside the interval $t_{\min} < t < t_{\max}$, or an amplitude-only mask otherwise.

PMT-to-cell mapping Finally, each PMT channel is mapped to a calorimeter cell index through the function `pmt_slot()`. Valid channels return a slot index within the

TileCal cell container, whereas unused or invalid channels return a sentinel value. For each valid PMT, the kernel writes the decoded and processed quantities into the GPU-side intermediate structure `TilePMTs`.

This structure stores, for each TileCal cell index in the final container range $[0, N_{TF})$, the contributions from the two associated PMTs (0 and 1), together with their quality flags and status bits. Here $N_{TF} = 5184$ corresponds to the total number of valid TileCal cells after PMT merging. The layout of this intermediate buffer is defined in Code 4.9, and it serves as the input for the subsequent PMT-merging kernel, which will combine the two PMT signals into a single calibrated cell measurement.

```
struct TilePMTFlags {
    uint32_t bits;
    enum : uint32_t { Mask = 1u << 0, D0 = 1u << 1 };
    ...
};

struct TilePMTs {
    uint32_t hash_idx[NTF];
    uint32_t gain0[NTF], gain1[NTF];
    float    ener0[NTF], ener1[NTF];
    float    time0[NTF], time1[NTF];
    uint8_t  qual0[NTF], qual1[NTF];
    uint8_t  qbit0[NTF], qbit1[NTF];
    TilePMTFlags flags0[NTF], flags1[NTF];
    ...
}
```

Code 4.9: `TilePMTs` structure definition.

4.3.5 Cell merging and finalisation

The next kernel merges the two PMT contributions belonging to each calorimeter cell and writes the combined results to the appropriate `CellInfoArr` position. The kernel is launched with one thread per calorimeter cell, indexed by the cell hash C. Each thread gathers the two raw entries corresponding to C and reads their tuples (e_p, t_p, q_p, g_p), where

$p \in \{0, 1\}$ denotes the PMT index. It reads the drawer-level data-quality mask dq, used to determine whether each PMT should contribute to the reconstructed cell.

A PMT is considered valid if its quality value is below the threshold used during decoding and if its corresponding DMU bit in the drawer's DQ mask is not set. Invalid PMTs ware masked, and how the kernel handles masked PMTs depends on its cell type.

For regular TileCal cells, both PMT contributions are preserved: if they are both valid, the `CellInfoArr` will contain the computed aggregate quantities following the CPU `TileCell` convention: $E{=}e_0{+}e_1$ and $T{=}0.5\times(t_0{+}t_1)$. If only one PMT is valid, the other contributes zero energy and is marked as masked, while the valid PMT is left unchanged.

The TileCal decoder also deals differently with $D0$ cells. These cells are shared between the negative Long Barrel C-Side (LBC) and positive Long Barrel A-Side (LBA) sides of the calorimeter. This means that each PMT of a $D0$ cell is shared by a different ROS, and are thus packed in separate fragments. The CPU algorithm merges the $D0$ cells after everything else has been processed, as it needs to discover all $D0$ PMTs sequentially.

Since all channels are decoded at once on the GPU, their merging can happen alongside the ordinary cells. These follow a special duplication rule: if exactly one PMT is valid, its (e_p, t_p, q_p, g_p) values are copied to the other side. If both PMTs are invalid, the cell energy and time are neutralised, and the quality values are saturated to indicate a masked cell. Otherwise, they both contribute to the final cell value, with their quantities aggregated like any other common cell.

Finally, the kernel packs the per-PMT quality and provenance into the 32-bit `QualityProvenance` word of the `CellInfoArr`. The lower bytes (`tile_qual1` and `tile_qual2`) hold the channel quality values, while the upper bytes (`tile_qbit1` and `tile_qbit2`) store bit flags describing the reconstruction algorithm and channel status. Bits [0–2] identify the reconstruction type, bit 3 marks bad channels, bit 4 indicates ADC overflow, bit 5 flags channels with measurable time, bit 6 marks compact container storage, and bit 7 indicates non-zero reconstructed time.

4.4 Implementation variants

Several implementation variants were developed to investigate how different memory-management strategies, execution models, and CUDA features influence the performance and scalability of the bytestream conversion. These variants do not alter the decoding logic itself; rather, they explore alternative ways of organising memory, managing per-thread resources, scheduling kernel execution, and structuring data transfers. The following subsections summarise the most relevant design alternatives evaluated during this work, which form the basis for the performance studies presented in Chapter 5.

4.4.1 Structure allocation

The initial implementation allocated all GPU-related data structures inside the `execute()` method, which led to substantial allocation overhead. A second version moved the structures to the algorithm class as persistent members, but this resulted in a single shared instance of each buffer and was therefore not thread-safe under the multi-threaded execution model. To address this, the current implementation employs the previously mentioned `separate_thread_holder` template, which preallocates one instance of each structure per thread slot during algorithm initialisation. This provides thread-local storage that is safe under AthenaMT concurrency and eliminates all per-event allocations.

4.4.2 Memory management

To simplify memory management across different execution contexts, the `CaloRecGPU` package provides a set of helpers that define lightweight tags for CPU, CUDA devices, and CUDA pinned host memory, and uses them in a set of generic allocation and copy helpers.

On top of these helpers, a `SimpleContainer` template provides dynamically sized arrays in a given memory context. The container, which originally reallocated arrays upon resizing, was upgraded to follow a grow-only policy: if a resize request fits within the existing capacity, only the logical size is updated and no reallocation occurs; a new allocation and copy are performed only when the requested size exceeds the current capacity.

All bytestream-related structures are initialised with a capacity that covers the maximum expected event size, which avoids repeated allocations inside the event loop and ensures that buffers can be reused efficiently across events and threads.

4.4.3 Pageable and pinned memory

Host memory used in data transfers to the GPU can be allocated as either pageable (unpinned) or page-locked (pinned). Pageable memory is managed by the operating system and may be moved in and out of physical RAM. When such memory is passed to a `CUDA` transfer, the driver must first copy it into an internal pinned buffer before initiating the Direct Memory Access (DMA) transfer to the device, introducing additional overhead and preventing true asynchronous execution.

In contrast, pinned memory is guaranteed to remain resident in physical RAM and is directly accessible to the GPU's DMA engine. Transfers from pinned buffers therefore have lower latency and can be issued asynchronously with respect to the host, enabling overlap between data movement and kernel execution. For this reason, the optimised decoding variants allocate the bytestream staging buffers in pinned host memory.

4.4.4 Use of `CUDA` streams

`CUDA` streams are independent sequences of operations (kernel launches, memory copies, or synchronisation commands) that the GPU executes in issue order. Operations issued to the same stream execute sequentially, while operations issued to different streams may execute concurrently, provided the hardware supports it. This enables asynchronous execution: memory transfers and kernel launches return control to the host immediately, allowing the CPU to perform other work or enqueue additional operations.

When multiple streams are used, the GPU can overlap independent kernels, computation with memory transfers, or multiple memory transfers in different directions. The use of multiple streams can hide latency, improve device utilisation, and allow different stages of a pipeline to progress concurrently without explicit blocking on the host.

For benchmarking purposes, the baseline version of the developed algorithms utilises synchronous host-to-device transfers in the default `CUDA` stream, and all kernel launches are followed by a `cudaStreamSynchronize` call, also in the default stream. This was done to enable the wall time evaluation of each algorithm step, since no device operations can overlap. This version will be hereby called *GPU-msync* or *GPU-baseline*.

However, to further improve the throughput of the decoder and to better understand the role of synchronisation and pipelining, two additional `CUDA`-based variants were implemented. The first variant, denoted *GPU-ssync*, replaces the blocking host–device transfers and per–kernel synchronisations with asynchronous copies and a single `cudaStreamSynchronize` call after the last kernel launch, while still using only the default `CUDA` stream. This design isolates the overhead associated with frequent synchronisation points and blocking transfers and allows the quantification of how much of the total latency is attributable to these host–side stalls as opposed to the device computation itself.

The second variant, denoted *GPU-streams*, assigns separate `CUDA` streams to the LAr and TileCal decoding chains within the event loop, enabling their kernels and data transfers to execute concurrently when resources permit. The device buffers and data structures are initialised using asynchronous transfers in the default stream, after which each event dispatches its LAr and TileCal workloads to their respective streams. This configuration evaluates whether a pipelined, multi-stream execution model can provide throughput gains by overlapping independent decoding tasks, even though the individual kernels are comparatively small.

4.4.5 Shared memory

A further implementation variant of the LAr decoding kernel was developed to investigate the use of `CUDA` shared memory. In this version, the per-channel fields required during the bit-extraction and amplitude/time reconstruction steps are first staged into shared memory before the decoding logic is applied. This approach was considered because shared memory provides substantially lower access latency than global memory in workloads

with significant data reuse. The variant preserves the semantics of the baseline decoder but introduces an explicit data-staging phase, allowing the impact of shared-memory utilisation to be evaluated independently of other kernel optimisations.

4.5 GPU Results verification

To verify the correctness of the GPU decoding procedure, the calorimeter cell data reconstructed on the device are systematically compared with reference values obtained from the standard CPU-based algorithms. This validation ensures that the GPU implementation reproduces the expected numerical results. Because GPU reductions are non-deterministic and floating-point representations may differ slightly between architectures, even small deviations must be measured at this stage, as later algorithms that are more intensive in floating-point operations can amplify such differences.

The verification process is implemented through a dedicated consistency check that evaluates the GPU-decoded calorimeter cells against their CPU counterparts on a cell-by-cell basis. To do so, it retrieves the host `CaloCellContainer` decoded in the HLT, identified by the "`CellsName`" key in the `SeedLessFS` Store. The procedure measures both absolute and relative deviations in floating-point quantities, such as energy and time, while also checking for bit-wise equality through units-in-the-last-place (ULP) comparison. Discrete attributes, including gain, quality, and provenance, are validated through exact integer comparisons. All results are collected through the `Athena Monitoring` framework, allowing for a statistical characterisation of any discrepancies.

Units-in-the-last-place distance The ULP distance quantifies how many representable floating-point numbers lie between two given values. It measures the difference in terms of machine precision rather than real-number magnitude. In other words, it answers how many distinct floating-point values the hardware would need to increment to go from a to b. To compute it, both GPU and CPU decoded floating-point numbers are reinterpreted as unsigned integers using `std::bit_cast`, preserving their IEEE-754 bit pattern. Since

the IEEE encoding is not lexicographically ordered (negative numbers have their sign bit set), a transformation is applied to map the bit patterns to a monotonic integer space:

- For positive numbers, the sign bit is flipped (`xor 0x80000000u`), pushing them after all negatives.
- For negative numbers, all bits are inverted (`xor 0xFFFFFFFFu`), reversing their order so that more negative values correspond to smaller integers.

After this mapping, the integer ordering matches the numerical ordering of the real values, and the ULP distance can be expressed as:

$$\mathrm{ULP}(a, b) = \Big| \,\mathrm{to\_ordered}(\mathrm{bits}(a)) - \mathrm{to\_ordered}(\mathrm{bits}(b)) \,\Big|.$$

If either input is not finite (`NaN` or `Inf`), the function returns the maximum possible integer to signal an invalid comparison.

The code iterates over all calorimeter cells in the host container and computes the container index `Ci` using `m_calo_id->calo_cell_hash(cell->ID())`, which should be equivalent to the indexing used in the AoS `CellInfoArr`. Iteration stops once the index reaches the first Tile cell (`Ci` $\geq$ `CaloRecGPU::TileCellStart`), as currently, the comparison targets only the LArCal. All cells are evaluated once the TileCal implementation is finished.

For each cell, the function retrieves the host values `energy_h = cell->energy()` and `time_h = cell->time()`, and the corresponding device-decoded values `energy_d = cell_info->energy[Ci]` and `time_d = cell_info->time[Ci]`. It then computes absolute and relative errors,

$$E_{\mathrm{abs}} = |\mathrm{energy\_d} - \mathrm{energy\_h}|, \qquad \tau_{\mathrm{abs}} = |\mathrm{time\_d} - \mathrm{time\_h}|,$$
$$E_{\mathrm{rel}} = \frac{E_{\mathrm{abs}}}{\max(|\mathrm{energy\_h}|, 10^{-30})}, \qquad \tau_{\mathrm{rel}} = \frac{\tau_{\mathrm{abs}}}{\max(|\mathrm{time\_h}|, 10^{-30})},$$

where the small constant 10^{-30} prevents division by zero and stabilises the monitoring of very small magnitudes. The energy and time ULP distances are then calculated as

previously described, yielding zero only when the CPU and GPU values are bit-wise identical.

Discrete fields are also validated for exact equality. The gain value on the CPU `gain_h` is converted from standard to offline convention and compared with the device counterpart `gain_d` = `cell_info->gain[Ci]`, producing a boolean equality flag. Similarly, the quality and provenance values are packed on the CPU side into a single `QualityProvenance` `qp_h` and compared with the device value `qp_d` = `cell_info->qualityProvenance[Ci]`.

All numerical quantities are recorded through the `Athena Monitoring` framework using `Monitored::Scalar` variables grouped into a `Monitored::Group`. The metrics include absolute and relative errors, ULP counts, logarithmic magnitudes $\log_{10}|$`energy_h`$|$ and $\log_{10}|$`time_h`$|$, logarithmic absolute errors, and binary flags indicating whether the ULP distance is zero.

When any discrepancy is detected, such as non-zero ULP differences or mismatched discrete fields, a detailed debug message is emitted, showing the cell index, both CPU and GPU values for energy and time, their absolute and relative errors, ULP distances, and the values of gain and quality-provenance, marking any differences.

Throughout the loop, the function maintains a counter of cells with non-zero energy ULPs. After all cells are processed, if any differences are found, a warning is issued summarising how many LAr cells contain bit-wise discrepancies. Otherwise, a verbose message confirms successful decoding. This method therefore performs a layered validation: bit-wise precision checks via ULP distance, scale-aware relative and absolute error analysis, and exact integer comparisons for discrete quantities, ensuring that both host and device representations of each calorimeter channel are consistent.

Chapter 5

Experiments and Results

This Chapter presents the experimental evaluation of the calorimeter bytestream decoder. It introduces the execution environments, dataset, and timing methodology, then compares the baseline CPU implementation with several GPU execution models and memory policies. The results are complemented with Nsight-based profiling and a direct CPU-GPU performance comparison, highlighting the main bottlenecks and implications for future calorimeter reconstruction pipelines on accelerators.

5.1 Testing environment

The performance evaluation and profiling of the bytestream decoder were carried out in two distinct execution environments (see Table 5.1). End-to-end benchmarks were performed on a Kubernetes cluster providing access to an A100 multi-instance GPU (MIG)-partitioned, whereas kernel-level profiling was executed on a shared `lxplus` GPU node due to access restrictions in the containerised environment. The two environments differ substantially in GPU architecture, resource isolation, scheduling behaviour, and timing stability, which must be taken into account when interpreting the results presented here.

Benchmarking environment The main end-to-end performance measurements were obtained on a Kubernetes-managed GPU cluster running `AlmaLinux 9.6` and equipped

Table 5.1: Summary of the execution environments used for benchmarking and profiling (captured on 12 Dec 2025).

	Kubernetes	**`lxplus`**
OS; Linux kernel	AlmaLinux 9.6; 6.12.9	RHEL 9.7; 5.14.0-570.60.1.el9_6
CPU	AMD EPYC 7313	Intel Xeon Silver 4216
CPU characteristics	16 Cores, 3 GHz	28 Cores, 2.10 GHz
GPU (architecture)	NVIDIA A100-PCIE-40GB (Ampere)	Tesla T4 (Turing)
GPU partitioning	MIG enabled: 1g.5gb (14 SMs)	-
Usable GPU memory	4,864 MiB (MIG slice)	15,360 MiB
Compiler	GCC 14.2.0	GCC 14.2.0
CUDA driver; Runtime	570.124.06; CUDA 12.8	580.95.05; CUDA 13.0
CUDA toolkit (nvcc)	12.8 (V12.8.93)	12.8 (V12.8.93)
Nsight Systems	-	2024.6.2.225
Nsight Compute	-	2025.3.1.0
Isolation; Access	Containerised; MIG partition	Shared node; no isolation
Role	End-to-end performance benchmarks	Kernel-level profiling (Nsight)

with an `AMD EPYC 7313` CPU. GPU access is provided through MIG partitioning of an `NVIDIA A100-PCIE-40GB`. The jobs used in this work were assigned a `1g.5gb` MIG slice, exposing only 14 SMs, one copy engine, and 5 GB of High Bandwidth Memory 2 (HBM2) memory. A full A100 contains 108 SMs, so the MIG slice used here represents approximately 13% of the device's compute resources.

This restricted configuration has several consequences. First, kernel occupancy and available parallelism are bounded by the 14 SMs of the slice, which limits overall throughput and the degree to which independent decoding tasks (e.g. LAr and TileCal) can run concurrently. Second, the MIG slices share the underlying Peripheral Component Interconnect Express (PCIe) interface, copy engines, scheduler, and power/thermal control logic of the physical GPU. As a result, timing outliers occur when other workloads on different MIG slices trigger large memory transfers or otherwise place load on shared GPU subsystems. The Kubernetes node is also shared at the CPU level, and containerised scheduling may intermittently pre-empt the decoding process, introducing additional fluctuations in event-level wall-clock time.

Profiling environment Kernel-level profiling was performed on a shared `lxplus` GPU node. `lxplus` is CERN's computing cluster, available to collaborators for software development, testing, and lightweight GPU or CPU workloads. The node used for tests runs `RHEL 9.7` and is equipped with an `Intel Xeon Silver 4216` CPU and an `NVIDIA Tesla T4`. This environment exposes all 40 SMs of the T4 to user processes but provides no isolation from other users. In practice, the lack of containerisation and frequent contention for the GPU and PCIe bus results in highly variable host-to-device transfer times and large event-level timing fluctuations. Consequently, `lxplus` was not suitable for end-to-end benchmarking, but it remained the only environment where Nsight Compute and Nsight Systems could be used.

Nsight profiling sessions mitigate the effects of node sharing by executing kernels under a controlled instrumentation run and reporting timing at nanosecond granularity. The `lxplus` environment was therefore employed exclusively for obtaining architectural information such as kernel occupancy, memory-access behaviour, synchronisation patterns, and potential kernel overlap. The profiling results are used in this chapter to interpret the behaviour of the decoding kernels but are not directly compared to the end-to-end timings obtained on the Kubernetes cluster.

5.2 Test dataset

The performance and correctness evaluation of the decoder was carried out using ATLAS Run-3 *pp* collision data, at a centre-of-mass energy of $\sqrt{s} = 13.6$ TeV. The data were recorded in September 2025 and belong to the primary Physics stream, which contains DAQ-format full-event readouts. The samples used are a subset of the events accepted by the HLT on run 506278. The complete dataset comprises approximately 1.2×10^7 events, with a total volume of about 198 TB.

From the run, three files of different luminosity blocks (LBs) (600, 1000, and 1200) were selected in order to test the decoder under different pile-up conditions. A luminosity block is a short interval of ATLAS data taking (typically about one minute) during which beam

Table 5.2: Summary of the luminosity blocks used for decoder evaluation.

Luminosity Block	File size	Events	$\langle\mu\rangle$
600	4.9 GB	3049	63.60
1000	4.9 GB	3325	50.76
1200	3.1 GB	2315	39.27

conditions, detector configuration, and trigger settings are considered stable. Luminosity blocks exhibit different pile-ups because they occur at different times within the same LHC fill: as it progresses, the beam intensity gradually decreases due to proton losses, leading to a steady reduction in instantaneous luminosity and thus in the value of μ.

The utilised RAW files contain the complete calorimeter bytestream readouts, and together they account for 8689 events. The file sizes and individual event counts are summarised in Table 5.2. The average pile-up $\langle\mu\rangle$ for each record was obtained from the `xAOD::EventInfo` object during event processing.

The selected LBs cover the pile-up range of $30 < \mu < 77$. Higher pile-ups often increase the number of LAr channels with packed time and quality information, leading to larger LAr bytestream payloads. The variation in pile-up and event size across the selected LBs enables the study of how the decoder performs depending on event complexity.

5.3 Benchmark methodology

Because of the constrained resources of the Kubernetes environment, and to avoid the effect of inter-thread contention on the measurements, all benchmarks were executed using AthenaMT with a single execution thread. Athena jobs process the list of input files sequentially by default. Therefore, each benchmark run corresponds to a full pass over all 8689 events from the aforementioned luminosity blocks.

Every relevant implementation variant was run 3 times within the same Kubernetes job, with each variant tested in a separate job. The repetition protocol was conducted to reduce the impact of scheduling noise and transient GPU utilisation from other MIG slices.

For each benchmark run, the full set of per-event timings was recorded. The results quoted in this section are obtained from the union of the three benchmarked runs, with the exclusion of their first event to reduce the influence of first event structure initialisation, both on host and device.

CPU timing instrumentation The CPU decoder was instrumented using the `Monitored::Timer` tool of the Athena Monitoring framework. With it, wall-clock microsecond timings around the "hotspots" of the LAr and TileCal unpacking chain were recorded, including: i) Retrieval of ROB fragments; ii) Preparation of per-detector bytestream structures; iii) ROD-level bytestream conversion; iv) BCID offset corrections (when enabled); and v) Construction and merging of the full calorimeter cell container.

Apart from these steps, the execution time of both the decoder driver methods (`prepareLArFullCollections` and `prepareTileFullCollections`) was measured to capture any possible timing overheads caused by locking, cache and container management, and other framework-level operations that are not part of the decoding itself.

GPU timing instrumentation GPU timings were obtained through the `CaloGPUMonitored` helper, which provides per-event wall-clock timing via a microsecond-resolution `stat_timer` implemented with `boost::chrono::thread_clock`. The following operations were explicitly measured:

i) Collection of ROB fragments;

ii) Preparation of bytestream and offset-correction buffers on the host;

iii) Host-to-device transfers of bytestreams, ROD offsets and BCID-correction containers;

iv) GPU parsing kernels for LAr and TileCal;

v) GPU decoding kernels for LAr and TileCal, including PMT merging.

Device-to-host transfers were not included, as the decoding pipeline is intended to feed downstream GPU algorithms directly. All measurements correspond to wall-clock latency as observed by the CPU, rather than `CUDA` event timing, to enable one-to-one comparability with the CPU reference.

Synchronization strategy across variants The performance comparisons cover the 3 execution models introduced in Section 4.4 (*GPU-msync*, *GPU-ssync*, and *GPU-streams*). In addition, diagnostic configurations were benchmarked to isolate specific effects: per-event reallocation (*GPU-realloc*), the use of pageable memory (*GPU-unpinned*), and operation with the LAr BCID correction disabled (*CPU-NOC* and *GPU-NOC*). A shared-memory LAr decoder was profiled on `lxplus`, but not included in the end-to-end timing analysis.

Memory allocation policy All GPU buffers are allocated using a thread-local mechanism during the first event processed by each AthenaMT slot. With the single-thread configuration, this resulted in exactly one allocation of per-thread LAr buffers, TileCal buffers, global constant LAr and TileCal lookup tables, and GPU cell container structures. These buffers are sized to accommodate the maximum expected number of bytestream words and are reused across events processed by the same thread. Only the intentionally adversarial *GPU-realloc* variant performs per-event deallocation and reallocation.

Correctness validation All GPU variants were validated against the CPU reference decoder by comparing the complete set of calorimeter cells for all events using ULP-level comparison of energy, time, quality, gain, and provenance values. All variants produced bit-identical results to the CPU reference, with no observed deviations across runs or implementation variants. For this reason, the cell-by-cell comparisons will not be presented.

Table 5.3: Summary of CPU timing measurements for the calorimeter decoder, in milliseconds (ms). Percentiles quantify the tail latency of the event-level timing distributions. σ denotes the standard deviation of the wall-time measurements.

Source	Decoding Step	Median	p90	p95	p99	Mean $\pm\sigma$
Data Retrieval	LArCal ROBs	0.27	0.38	0.45	0.62	0.28 ± 0.10
	TileCal ROBs	0.84	0.93	0.99	1.20	0.86 ± 0.11
	BCID Corrections	0.16	0.23	0.27	0.39	0.17 ± 0.06
LAr	Bytestream Conversion	5.21	7.25	8.93	13.83	5.79 ± 1.69
	BCID Application	2.52	3.01	3.32	4.11	2.64 ± 0.36
	Overhead	6.33	6.83	7.18	8.42	6.42 ± 0.49
TileCal	Bytestream Conversion	1.59	2.02	2.29	3.04	1.68 ± 0.33
	Overhead	1.20	1.34	1.43	1.70	1.22 ± 0.19
Full Calorimeter	Total Decode Time	10.62	13.66	16.09	22.33	11.43 ± 2.38
	Total Overhead Time	9.46	10.34	10.93	12.63	9.65 ± 0.74
	Total Time	**20.15**	**23.94**	**26.81**	**34.12**	**21.08 ± 2.94**

5.4 Performance evaluation

5.4.1 CPU baseline performance

The CPU measurements described provide a reference for the end-to-end cost of decoding the LAr and TileCal bytestreams in the current HLT data-access service. To evaluate the complete CPU decoding chain, a composite quantity is used, denoted here as the total decoding time:

$$t_{\text{CPU}}^{\text{decode}} = t_{\text{LAr-CPU}}^{\text{decode}} + t_{\text{Tile-CPU}}^{\text{decode}} ,$$

In this formulation, $t_{\text{LAr-CPU}}^{\text{decode}}$ is the wall-time taken to unpack the raw channels and apply the BCID offset corrections, and $t_{\text{Tile-CPU}}^{\text{decode}}$ is the sum of TileCal ROBs retrieval raw data unpacking. In addition, the proxy timing $t_{\text{CPU}}^{\text{total}}$, which encompasses the full cost of the `loadFullCollections` call, was measured. From this, one can infer the overhead time the service spent on secondary processes or contention as:

$$t_{\text{CPU}}^{\text{overhead}} = t_{\text{CPU}}^{\text{total}} - t_{\text{CPU}}^{\text{decode}} .$$

The CPU timing behaviour of the full calorimeter bytestream decoder is summarised

in Table 5.3. The median event decoding time is of ≈10.6 ms, with a median total calorimeter-loading time of ≈20.2 ms.

The measurements highlight that a considerable amount of time is spent on cache management and bookkeeping, condition and helper logic that wasn't explicitly timed, container copying and pointer re-ordering, as well as other miscellaneous framework processes. This creates an overhead that is almost as large as the decoding itself, as depicted in Figure 5.1. This overhead is largely independent of event content and exhibits relatively low variability, while event-to-event variability is moderate for most decoding steps, as reflected by the differences between the mean and the 90th/95th percentiles.

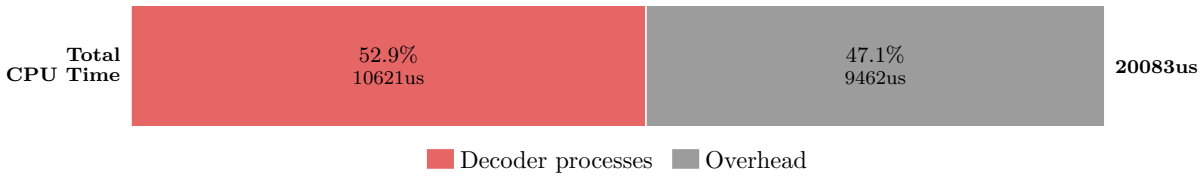

Figure 5.1: Fractions of the total time spent on CPUs-based decoding processes and overhead. Based on medians.

The mean exceeds the median in all steps, indicating a positively skewed distribution with occasional higher-latency events. This is particularly supported by the gap between the median (5.21 ms) and the p95 (8.93 ms) of the LAr bytestream conversion step, showing the presence of a long tail in the raw-channel unpacking.

The BCID-correction stage shows significantly lower variability ($\sigma \approx 0.36$ ms), with narrow percentile ranges, reflecting the fact that its computational cost depends mainly on the number of FEBs rather than on the structure of the event. Its execution time is therefore substantially more stable than the raw LAr conversion.

A visual breakdown of the relative contributions is shown in Figure 5.2. The LAr subsystem dominates the total decoding time: LAr decoding and BCID correction together account for roughly 75.0% of the total. Although TileCal represents only 6% of the bytestream channels (12.288 PMTs compared to the 195.072 raw LAr channels), its decoding time contributes to about 14% of the processing time.

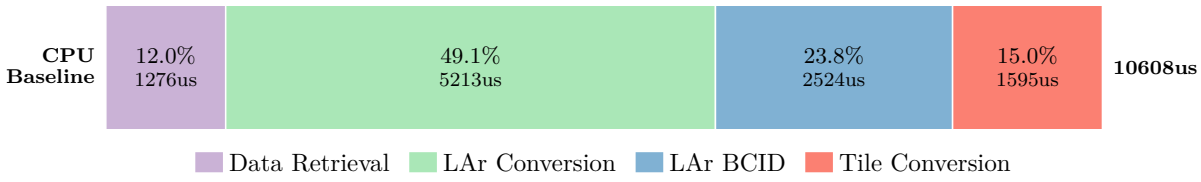

Figure 5.2: Fractions of the total time spent on each of the CPU-based decoding step. Based on medians.

5.4.2 GPU baseline performance

The GPU–synchronous configuration (*GPU-msync*) represents the reference implementation of the accelerator-based decoder. As described, this variant prohibits any overlap between data movement and device execution. Table 5.4 summarises the per-event timing behaviour of the implementation. The median end-to-end latency for the synchronous GPU decoder is approximately 1.215 ms, with tight upper-percentile bounds indicating a compact latency distribution compared to the CPU baseline.

Table 5.4: Summary of the timing measurements of the *GPU-msync* calorimeter decoder, in microseconds (μs). Percentiles quantify the tail latency of the event-level timing distributions. σ denotes the standard deviation of the wall-time measurements.

Source	Decoding Step	Median	p90	p95	p99	Mean $\pm\sigma$
Data Retrieval	LAr ROBs	111	215	263	379	135 ± 65
	TileCal ROBs	18	35	42	80	23 ± 13
	BCID Corrections	215	379	448	597	242 ± 104
Payload preparation	LAr Data	283	621	754	1021	345 ± 198
	TileCal Data	41	98	120	181	53 ± 34
Host-to-Device Transfers	LAr Data	312	370	393	451	282 ± 97
	TileCal Data	74	93	113	166	74 ± 27
LAr Kernels	Parse	23	35	46	93	27 ± 21
	Decode	63	69	84	133	66 ± 15
TileCal Kernels	Parse	18	22	25	64	19 ± 11
	Decode	19	23	26	59	20 ± 9
	Merge	14	17	19	50	15 ± 8
Full Calorimeter	**Total Time**	**1215**	**1812**	**2067**	**2656**	**1305 ± 388**

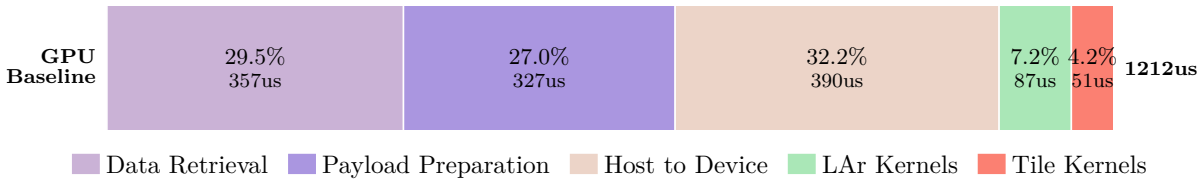

Figure 5.3: Fractions of the total time spent on each of the GPU-based decoding step. Based on medians.

Figure 5.4: Time fractions of the GPU-msync decoding chain.

A prominent feature of the baseline GPU path is that the majority of the event-level latency originates not from the GPU kernels themselves but from host-side preparation and PCIe transfers. The decoder spends $\sim 61\%$ of the total time on data retrieval and payload preparation (about 400 μs and 398 μs on average). These stages include ROB fragment collection, bytestream assembly, and BCID-correction updates.

Host-to-device transfers constitute a further 27% of the per-event latency (356 μs), with this step alone exceeding the total cost of all GPU kernels combined. On the MIG-partitioned A100 used for the benchmarks, only a single copy engine is available, and all transfers are serialised by design. Furthermore, all MIG instances share the same physical PCIe 4.0 $\times$16 link to the host, so heavy traffic from other slices can, in principle, contribute to fluctuations in the observed transfer times, although this effect was not studied in detail here.

In contrast, the device kernels themselves account for only $\sim 11\%$ of the total event time. The LAr parsing and decoding kernels together take an average of 94 μs, while the TileCal kernels take ~ 55 μs. Kernel timing exhibits low dispersion, indicating that device-side execution is stable and insensitive to the structure of the event, reflecting the fine-grained parallelism of the decoding work.

Overall, the *GPU-msync* measurements show that the decoding pipeline is not compute-bound. Instead, the dominant contributions come from CPU-side payload construction and synchronous PCIe transfers, which collectively account for nearly 90% of the total latency. The results suggest that eliminating unnecessary synchronisation, overlapping

data transfers with computation, and exposing the parallelism between LAr and TileCal subsystems can further reduce the GPU dependent latency.

5.4.3 Impact of execution model and memory policy

To assess how synchronisation strategy, memory policy and subsystem-level parallelism influence the decoding performance, 3 representative variants were evaluated: *GPU-ssync*, *GPU-realloc* and *GPU-streams*. Their aggregate per-event timings are shown in Table 5.5. Figure 5.5 compares the GPU-side contribution to the event latency (combined cost of host-to-device transfers and device kernel execution) for the 3 execution models.

Table 5.5: Summary of the timing measurements for the *GPU-msync*, *GPU-ssync* and *GPU-streams* calorimeter decoder variants, in microseconds (μs). Percentiles quantify the tail latency of the event-level timing distributions. σ denotes the standard deviation of the wall-time measurements.

Variant	Source	Median	p90	p95	p99	Mean $\pm\sigma$
GPU-msync	Data retrieval	357	597	708	960	400 ± 154
	Payload preparation	327	715	864	1158	398 ± 224
	Kernels + H2D	533	617	649	737	505 ± 121
	Total	**1215**	**1812**	**2067**	**2656**	**1305 ± 388**
GPU-ssync	Data retrieval	329	552	655	908	370 ± 147
	Payload preparation	322	690	844	1136	391 ± 216
	Kernels + H2D	354	416	436	494	322 ± 101
	Total	**998**	**1493**	**1722**	**2330**	**1083 ± 334**
GPU-streams	Data retrieval	514	668	764	1066	545 ± 127
	Payload preparation	424	634	794	1477	474 ± 204
	Kernels + H2D	926	1008	1040	1142	894 ± 176
	Total	**1864**	**2222**	**2456**	**3373**	**1912 ± 352**

Asynchronous execution (*GPU-ssync*). Replacing the per-kernel `cudaStreamSynchronize()` calls with asynchronous copies and a single end-of-event synchronisation reduces the total latency from 1.30 ms (baseline) to 1.08 ms, a gain of approximately 17%. The improvement stems entirely from the removal of host-side synchronisation points: the kernels themselves are unchanged, and the device is still unable to overlap computation

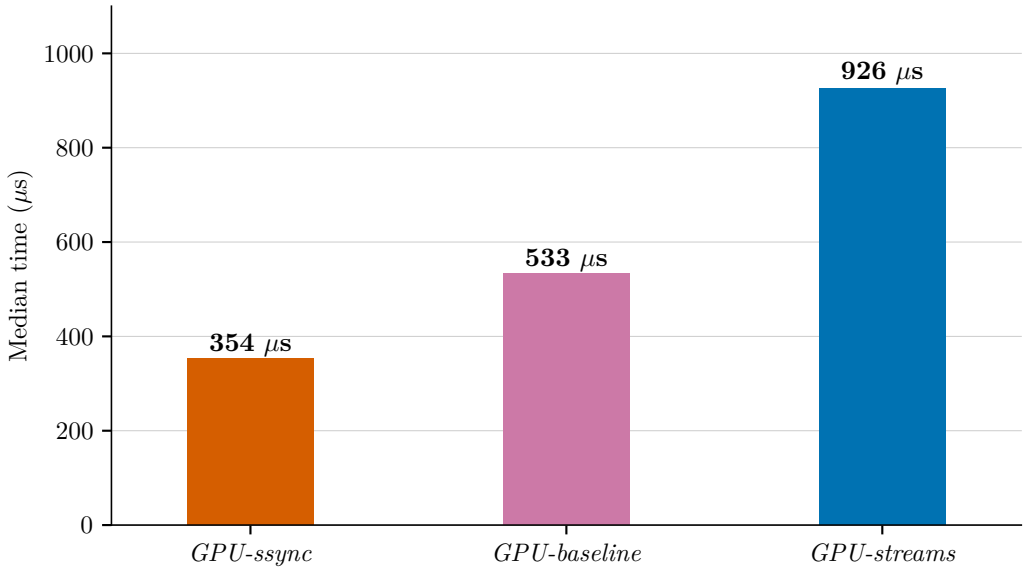

Figure 5.5: Timing comparison between the different GPU decoder versions.

with transfers because all operations use the same stream. These results confirm that synchronisation overhead, rather than device computation, is a non-negligible contributor to the baseline latency.

Figure 5.6 shows the distribution of the total GPU-side time for the synchronous and asynchronous models. The *GPU-msync* distribution peaks at 500–600 μs and exhibits a broad right-hand tail. In contrast, the *GPU-ssync* configuration shifts the entire distribution to a lower latency (300–400 μs) and reduces the event-to-event variability, resulting in a tighter and shorter-tailed distribution. The improvement is uniform across the dataset and is not limited to a subset of events, demonstrating that synchronisation overhead is a systematic and dominant component of the synchronous model.

Two-stream execution (*GPU-streams*). Using independent `CUDA` streams for LAr and TileCal yields a total latency of 1.91 ms, which is slower than both the synchronous and asynchronous variants. On the 1g.5gb MIG slice used for the benchmarks, the individual decoding kernels are already short, and the transfer phases are serialised. As a result, there is limited opportunity for effective overlap between the LAr and TileCal pipelines,

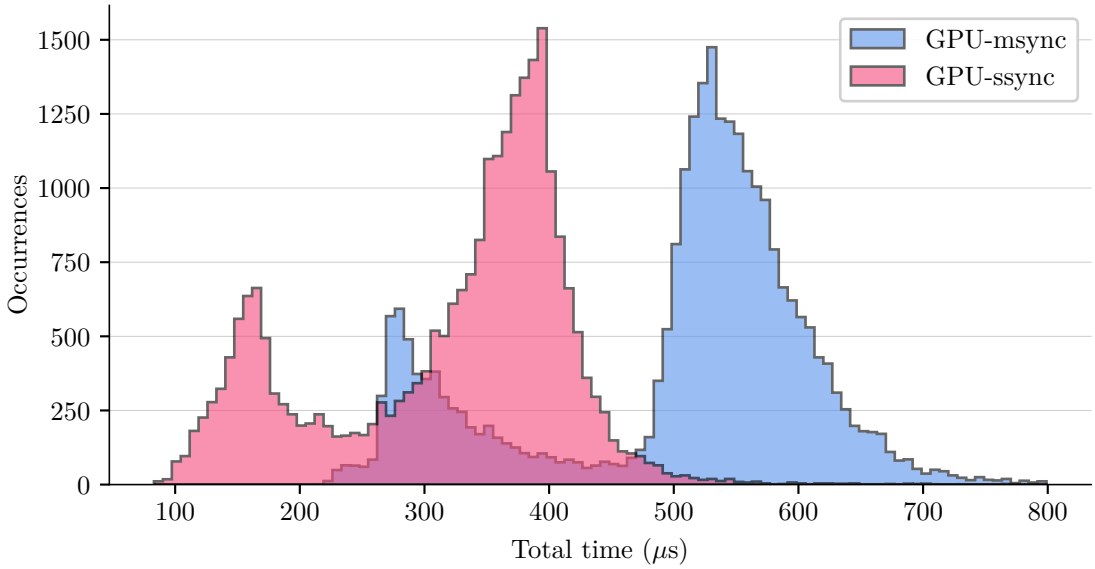

Figure 5.6: Time distribution of GPU-related operations for GPU sync and async versions.

and any potential stream-level concurrency is largely masked by normal event-to-event and run-to-run variability on the shared Kubernetes node.

Within the statistical precision of the measurements, the multi-stream configuration does not provide a clear advantage over the simpler asynchronous model, and the observed (0.5 ms) increase in total time is interpreted as a possible host-side jitter caused by the shared container cluster rather than a change in the underlying decoding workload. On a full A100 (with more SMs and multiple copy engines), one would expect a richer concurrency pattern, but assessing the benefit of such a configuration would require a study in an isolated benchmarking environment.

5.4.4 Diagnostic variants

Per-event reallocation (*GPU-realloc*). The deliberately adversarial configuration, which frees and reallocates all buffers on every event, exhibits a tenfold slowdown with a total latency of 12 ms. The dominant contribution is the repeated initialisation of device and host buffers, with `init_structures` alone costing around 10 ms per event. This just confirms that buffer reuse is a necessity for effective GPU decoding (and any other

re-entrant algorithm) in AthenaMT.

Impact of pageable memory To quantify the impact of host–memory configuration on the decoding pipeline, the *GPU-msync* implementation was executed using ordinary pageable memory instead of pinned (page-locked) allocations. Table 5.6 summarises the per-event timings obtained in the Kubernetes environment.

Table 5.6: Comparison of pinned and unpinned memory configurations for the *GPU-msync* decoder, using percentiles to characterise timing. All values are in microseconds (μs).

Step	**Pinned**			**Unpinned**		
	Median	**p90**	**p95**	**Median**	**p90**	**p95**
Data retrieval	357	597	708	528	710	818
Payload preparation	327	715	864	464	858	1042
Host-to-device	390	459	484	536	626	1016
GPU kernels	140	180	204	260	310	326
Total	**1215**	**1812**	**2067**	**1796**	**2452**	**2892**

Using pageable memory degrades both the latency and the stability of the pipeline. Although pageable buffers affect the host-to-device transfers the most, the slowdown propagates to every earlier stage of the event: data retrieval, bytestream preparation, and BCID processing all show increased latency and significantly higher dispersion.

A possible explanation is that because pageable memory cannot be used directly for DMA transfers, before each `cudaMemcpy()` operation, the `CUDA` driver must implicitly allocate a temporary pinned buffer, copy the payload into it, and then issue the actual PCIe transfer. This extra memory copy introduces synchronisation with the `CUDA` runtime and interacts with the operating system's virtual-memory subsystem, leading to additional page faults, Translation Lookaside Buffer (TLB) churn, and cache pollution.

No BCID offset correction variation The configuration in which the LAr BCID-correction stage was disabled was evaluated to compare its impact on both CPU and

GPU. The measured impact is substantial. In the CPU decoder, the average end-to-end time decreases from 21.1 ms to 16.1 ms (∼24%). In the *GPU-msync* variant, the latency improves from 1.30 ms to 0.89 ms (∼32%) (see Figure 5.7). These reductions originate almost entirely from lighter host-side preparation: offset correction touches every FEB and increases the amount of metadata handled per event.

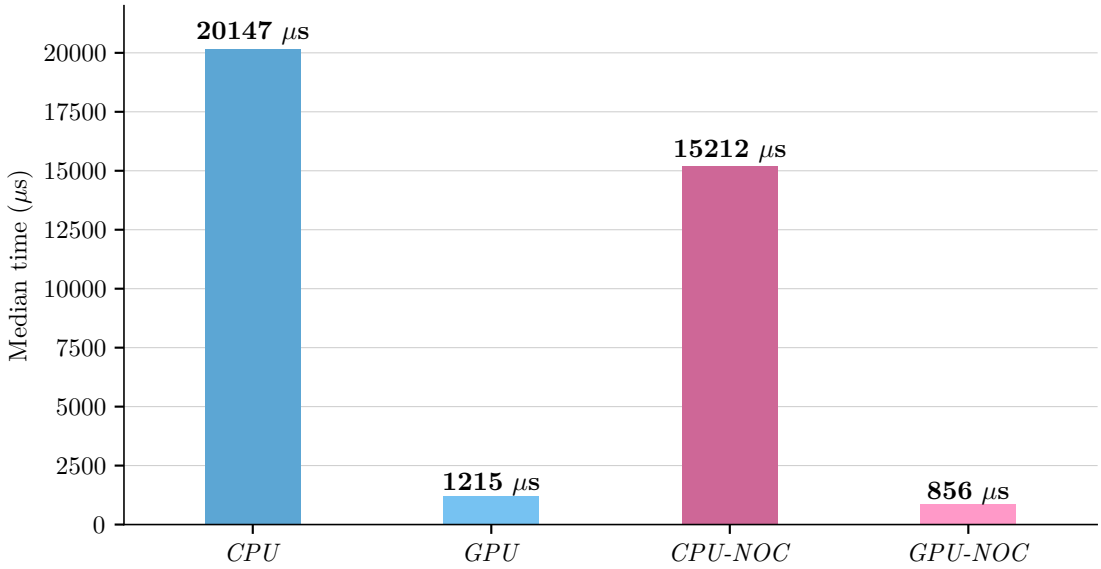

Figure 5.7: Average per-event decoding time for the baseline CPU and GPU implementations, with and without LAr BCID offset correction (NOC).

5.4.5 Architectural profiling with Nsight tools

To expose how the workload maps onto the device architecture or how much of the total latency can, in principle, be removed, the implementation was profiled with Nsight Compute and Nsight Systems on the `lxplus` T4 environment described in Section 5.1.

Kernel-level behaviour (Nsight Compute). Nsight Compute shows that the calorimeter decoding workload is dominated by the LAr cell decoder, while the remaining kernels

are short and underutilise the device. The LAr parser, TileCal parser, and TileCal merging kernels execute in tens of microseconds and launch only a few blocks per event. They achieve low occupancy and negligible SM utilisation, but their contribution to the total event time is correspondingly small, so they do not constitute a performance bottleneck. The main profiled metrics are summarised in Table 5.7.

In contrast, the LAr decoder kernel exhibits high achieved occupancy and sustained memory throughput in the 100-150 GB/s range. However, the profiling also reveals that the kernel is strongly memory-bound: a large fraction of global load transactions are classified as uncoalesced or "excessive" at the Level-2 (L2), and a significant portion of the cycles are spent in stalls attributed to L1/L2 memory dependencies. Threads within a warp frequently access scattered addresses corresponding to different FEB payload indices. This is due to different fields of a single cell being distributed across the bytestream array rather than grouped together, causing each logical load to expand into multiple cache-line fetches and reducing the effective bandwidth.

Table 5.7: Summary of Nsight Compute metrics for the kernels in the synchronous GPU decoding pipeline. Here, the word "Throughput" is abbreviated as "Tp.", "Compute" as "Comp." and "Memory" as "Mem.".

Metric	**LAr**		**Tile**		
	Parser	**Decoder**	**Parser**	**Decoder**	**Merger**
Grid size	$(6, 1, 1)$	$(1524, 1, 1)$	$(1, 1, 1)$	$(96, 1, 1)$	$(41, 1, 1)$
Duration (µs)	10.1	67.2	19.6	13.4	6.5
Occupancy (%)	12.1	84.9	12.1	26.8	12.7
Comp. Tp. (%)	11.8	49.0	1.7	14.5	15.4
Mem. Tp. (GB/s)	37.1	151.9	5.5	38.7	49.0
Busy SM (%)	0.7	12.7	0.1	7.6	3.2
Busy Mem. (%)	4.5	25.3	0.7	9.7	7.0
L1 Hit Rate (%)	68.3	33.8	71.6	73.5	22.7
L2 Hit Rate (%)	62.5	69.6	68.4	73.7	41.3

The micro-optimisations of the LAr decoder were profiled to assess whether the memory bottleneck could be mitigated without changing the bytestream layout. The shared

memory variant showed no improvement: it introduces additional loads, stores, and synchronisation and does not increase inter-thread data reuse (most bytestream addresses are only read once), and the number of global transactions observed at L2 remains unchanged. As a result, this version is approximately 10% slower than the baseline. Overall, the Nsight Compute results indicate that, for the current bytestream format, the LAr decoder performance is limited primarily by the layout of the input data and the lack of spatial locality between channels, rather than by insufficient arithmetic throughput or sub-optimal use of on-chip memory.

System-level orchestration (Nsight Systems). Nsight Systems shows that all execution models generate nearly identical GPU timelines. In all variants, the GPU work occupies well under 1% of the wall time of a complete job. Instead, 65–75% of the total runtime is absorbed by Athena event-loop synchronisation (`pthread_cond_timedwait`, `futex`, `poll`), with a two-threaded execution exhibiting even higher contention.

The `CUDA` API behaviour reflects the same distinction. When buffers are reused, API activity consists mainly of lightweight kernel launches. Diagnostic configurations that perform per-event allocation (*GPU-realloc* or *GPU-unpinned*) invert this balance: calls to `cudaMallocHost` or `cudaMalloc` are the dominant cost, fully overshadowing the kernels.

Changing the execution model, therefore, has only a modest impact on device utilisation. Asynchronous execution removes several host-side synchronisation points but does not produce meaningful overlap between transfers and kernels because all operations target a single stream and the GPU kernels are already short. The two-stream configuration shows minor kernel overlap, but the benefit is masked by host-side waiting.

Implications for future pipelines. Taken together, the Nsight Compute and Nsight Systems profiles reinforce the conclusion that the current decoder is not limited by device computation. At the kernel level, the main LAr decoder is already close to saturating the available memory bandwidth given the constraints of the bytestream format. At the system level, the GPU kernels and transfers occupy only a small fraction of the

event lifetime, and the improvements observed when moving from the synchronous to the asynchronous model stem largely from the removal of unnecessary host synchronisation points rather than from genuine overlap of computation and data movement.

The lack of clear gains from the two-stream configuration, however, reflects the shallowness of the present pipeline. A future calorimeter chain that offloads other steps of the event reconstruction to GPU would naturally consist of many more kernels and larger intermediate buffers. On such a deeper pipeline, separate streams for different stages of the reconstruction could allow effective pipelining of transfers and computation across events and sub-detectors.

5.5 Direct CPU and GPU Comparison

The total per-event latency of the reference implementations is summarised in Table 5.8. Even in the strictly synchronous configuration, the GPU decoder achieves an order-of-magnitude reduction in decoding time relative to the CPU processing time (11.4 ms), and is 16× faster than the complete CPU chain (21.08 ms).

Table 5.8: Direct comparison between CPU and main GPU variant timings. Speed-ups are calculated with respect to CPU.

Decoder	**Median (ms)**	**Speed-up**
CPU	20.1	1×
CPU (decoding only)	10.6	1.89×
GPU-msync total	1.21	16.60×
GPU-ssync total	0.99	22.30×

The magnitude of this improvement is primarily driven by two factors:

- **Massively parallel device kernels:** The bytestream unpacking and channel decoding tasks map naturally onto fine-grained parallelism. Even the limited 14-SM MIG slice achieves an average device-side decoding of 100 μs, more than an order of magnitude faster than the corresponding CPU functions.

- **Reduced framework overhead:** The implemented GPU path bypasses several intermediate data structures and avoids repeated allocation and cache operations performed by the CPU services. As a result, the effective overhead is basically non-existent compared to that of the CPU implementation.

Despite these gains, both the synchronous and asynchronous variants show that most of the residual latency originates from CPU-side bytestream preparation and host–device transfers. When these stages are removed or reduced (e.g. by disabling BCID corrections), the end-to-end cost of the GPU path decreases proportionally, whereas the device kernel times remain nearly unchanged.

Chapter 6

Discussion and Conclusions

This dissertation investigated the feasibility of offloading the ATLAS calorimeter bytestream decoding step to GPUs, motivated by the substantial increases in event complexity and data throughput expected during the HL-LHC era. Although the existing CPU-based decoders are robust and operationally mature, they inherit assumptions and design choices made more than two decades ago, at a time when single-threaded performance and modest event sizes were the dominant constraints.

As shown in the previous chapters, their largely sequential structure, irregular memory access patterns, and CPU-centric data layouts impose intrinsic limits on scalability under future conditions. This work explored whether massively parallel architectures, GPUs in particular, can alleviate some of these bottlenecks and whether the decoding algorithms can be reformulated to exploit parallelism while remaining functionally equivalent to the established reference.

6.1 Summary of Achievements

A complete GPU-based decoding pipeline was implemented for both the LAr and TileCal calorimeters, covering all stages from fragment staging to device transfer, parsing, channel-level decoding, PMT merging, and cell-level finalisation. The implementation preserved the semantics of the existing CPU algorithms and integrated into the AthenaMT execution

model using `StoreGate` handles and thread-local GPU buffers.

The pipeline achieved bit-level or ULP-level agreement with the CPU reference across all tested events, demonstrated substantial reductions in per-event decoding latency even on a constrained 1g.5gb A100 MIG slice, and exhibited consistent behaviour across synchronous, asynchronous, and multi-stream execution models. Extensive instrumentation and profiling clarified where the pipeline is compute bound, memory bound, or dominated by PCIe transfers, enabling a detailed understanding of its performance characteristics.

6.2 Interpretation of Results

Micro-optimisations and Kernel Granularity At first sight, the GPU kernels used for bytestream parsing are small, branch-light, and operate on modest amounts of data per event. For this reason, several of the micro-optimisations evaluated during development yielded small absolute runtime changes. Because the decoding kernels typically execute below the 100 μs scale, reducing instruction count or improving memory locality only saves a few microseconds.

However, interpreting these results solely in the context of the current Run-3 bytestream format would be misleading. If GPUs become the preferred accelerator for the HL-LHC trigger or offline systems, the bytestream format itself will be redesigned together with the front-end electronics and ROD firmware. In such a scenario, the design principles extracted from these micro-optimisations become guidelines for future data layouts rather than marginal improvements. Eliminating irregular header structures, grouping channel data in tightly packed and coalesced SoA regions, simplifying optional fields, or enforcing predictable payload sizes would drastically reduce divergence and make decoding more warp-friendly.

Downstream GPU Residency The benefit of GPU decoding extends beyond the decoding step itself. Constructing calorimeter cells directly in device memory eliminates the retrieve–transform–transfer sequence that the current GPU workflows require. This

reduces PCIe traffic, removes costly container copies, and makes downstream clustering or topological selection immediately available without additional data movement. The observed order-of-magnitude speed-up in decoding therefore represents only the first visible contribution to what could become a fully GPU-resident calorimeter reconstruction chain.

Toward Fused GPU Pipelines The present implementation launches parsing and decoding as separate kernels primarily for clarity and debugging. However, this may introduce non-negligible kernel launch overheads. A possible extension of this work is the implementation of a single, fused GPU kernel that performs the parsing and decoding in one continuous pass. Such an approach would also eliminate intermediate global-memory writes and potentially reduce divergence by assigning coherent detector regions to specific thread blocks.

Impact of the Execution Environment Interpretation of the benchmark results must account for the constrained evaluation environment. The A100 MIG slice provided only 14 SMs and one copy engine, PCIe bandwidth was shared across MIG tenants, and container-level CPU scheduling introduced additional non-determinism. Profiling was only possible on a shared T4 GPU on `lxplus`, where contention further affects measurements. These conditions imply that the achieved latency reductions are conservative estimates and that significantly larger speed-ups are expected on full GPUs or dedicated accelerator nodes.

6.3 Limitations and Outlook

This work did not attempt integration into production HLT menus or downstream clustering, as this depends on further discussions about how to properly integrate the GPU structures with the current Athena framework object handling.

The implementation remains constrained by the irregular and legacy structure of the

current bytestream, and only a limited set of GPU architectures could be tested. Host-to-device transfers also remain a bottleneck for LAr payloads, particularly in shared or PCIe-congested environments. These points outline where future work can further improve the pipeline, and they highlight the broader opportunities for accelerator-aware redesign of calorimeter data preparation in the coming years.

Bibliography

[1] D. H. Perkins, "Quarks and leptons," in *Introduction to High Energy Physics*, Cambridge: Cambridge University Press, 2000, pp. 1–34, ISBN: 978-0-511-80904-0. DOI: 10.1017/CBO9780511809040.002.

[2] M. K. Gaillard, P. D. Grannis, and F. J. Sciulli, "The standard model of particle physics," *Reviews of Modern Physics*, vol. 71, no. 2, S96–S111, Mar. 1, 1999, ISSN: 0034-6861, 1539-0756. DOI: 10.1103/RevModPhys.71.S96. arXiv: hep-ph/9812285. [Online]. Available: http://arxiv.org/abs/hep-ph/9812285.

[3] Cush, *Standard model of elementary particles*, Sep. 17, 2019. [Online]. Available: https://commons.wikimedia.org/wiki/File:Standard_Model_of_Elementary_Particles.svg.

[4] D. J. Griffiths, *Introduction to Elementary Particles*, 2. Auflage. Weinheim: Wiley-VCH GmbH, 2020, 1 p., ISBN: 978-3-527-83464-8.

[5] D. H. Perkins, "Electroweak interactions and the standard model," in *Introduction to High Energy Physics*, Cambridge: Cambridge University Press, 2000, pp. 242–275, ISBN: 978-0-511-80904-0. DOI: 10.1017/CBO9780511809040.007.

[6] F. Englert and R. Brout, "Broken symmetry and the mass of gauge vector mesons," *Physical Review Letters*, vol. 13, no. 9, pp. 321–323, Aug. 31, 1964, ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.13.321. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.13.321.

[7] G. S. Guralnik, C. R. Hagen, and T. W. B. Kibble, "Global conservation laws and massless particles," *Physical Review Letters*, vol. 13, no. 20, pp. 585–587, Nov. 16, 1964, ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.13.585. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevLett.13.585.

[8] G. Aad et al., "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC," *Physics Letters B*, vol. 716, no. 1, pp. 1–29, Sep. 17, 2012, ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037026931200857X.

[9] S. Chatrchyan et al., "Observation of a new boson at a mass of 125 GeV with the CMS experiment at the LHC," *Physics Letters B*, vol. 716, no. 1, pp. 30–61, Sep. 17, 2012, ISSN: 0370-2693. DOI: 10.1016/j.physletb.2012.08.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0370269312008581.

[10] S. Koren. "The hierarchy problem: From the fundamentals to the frontiers." arXiv: 2009.11870 [hep-ph]. [Online]. Available: http://arxiv.org/abs/2009.11870, pre-published.

[11] S. P. Martin, "A supersymmetry primer," in *Perspectives on Supersymmetry*, vol. 18, WORLD SCIENTIFIC, Jul. 1998, pp. 1–98. DOI: 10.1142/9789812839657_0001. arXiv: hep-ph/9709356. [Online]. Available: http://arxiv.org/abs/hep-ph/9709356.

[12] T. G. Rizzo. "Pedagogical introduction to extra dimensions." arXiv: hep-ph/0409309. [Online]. Available: http://arxiv.org/abs/hep-ph/0409309, pre-published.

[13] O. Witzel. "Review on composite higgs models." arXiv: 1901.08216 [hep-lat]. [Online]. Available: http://arxiv.org/abs/1901.08216, pre-published.

[14] D. N. Spergel, "The dark side of cosmology: Dark matter and dark energy," *Science*, vol. 347, no. 6226, pp. 1100–1102, Mar. 6, 2015, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aaa0980. [Online]. Available: https://www.science.org/doi/10.1126/science.aaa0980.

[15] J. M. Butterworth, "The standard model: How far can it go and how can we tell?" *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2075, p. 20150260, Aug. 28, 2016, ISSN: 1364-503X, 1471-2962. DOI: 10.1098/rsta.2015.0260. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2015.0260.

[16] L. Evans and P. Bryant, "LHC Machine," *Journal of Instrumentation*, vol. 3, no. 08, S08001, Aug. 2008, ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/08/S08001. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/3/08/S08001.

[17] "About CERN. "[Online]. Available: https://home.cern/about.

[18] E. Mobs, *The CERN accelerator complex in 2019*, 2019. [Online]. Available: https://cds.cern.ch/record/2684277.

[19] T. S. Pettersson and P. Lefèvre, "The Large Hadron Collider," 1995. DOI: 10.17181/CERN-AC-95-05-LHC. [Online]. Available: https://cds.cern.ch/record/291782.

[20] C. ATLAS and T. A. Collaboration, "Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System," CERN Document Server, CERN-LHCC-2017-020, ATLAS-TDR-029, 2017. DOI: 10.17181/CERN.2LBB.4IAL. [Online]. Available: http://cds.cern.ch/record/2285584.

[21] G. Aad et al., "The ATLAS Experiment at the CERN Large Hadron Collider," *JINST*, vol. 3, no. 08, S08003, 2008, ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/08/S08003. [Online]. Available: https://cds.cern.ch/record/1129811.

[22] G. Chatham Strong, "On the impact of selected modern deep-learning techniques to the performance and celerity of classification models in an experimental high-energy physics use case," *Machine Learning: Science and Technology*, vol. 1, no. 4, p. 45006,

Sep. 19, 2020, ISSN: 2632-2153. DOI: 10.1088/2632-2153/ab983a. [Online]. Available: https://iopscience.iop.org/article/10.1088/2632-2153/ab983a.

[23] "ATLAS Experiment at CERN - About. "[Online]. Available: https://atlas.cern/about.

[24] ATLAS Collaboration, *ATLAS Calorimeter Performance : Technical Design Report* (Technical Design Report. ATLAS 1). Geneva: CERN, 1996. [Online]. Available: https://cds.cern.ch/record/331059.

[25] ATLAS Collaboration and CERN. Geneva. LHC Experiments Committee, "ATLAS liquid-argon calorimeter: Technical Design Report," CERN, 1997. DOI: 10.17181/CERN.FWRW.FOOQ. [Online]. Available: http://cds.cern.ch/record/331061.

[26] Ammara Ahmad and ATLAS Collaboration, "The ATLAS tile calorimeter performance and its upgrade towards the high-luminosity LHC," *Moscow University Physics Bulletin*, vol. 77, no. 2, pp. 156–158, Sep. 9, 2022, ISSN: 0027-1349, 1934-8460. DOI: 10.3103/S0027134922020047. [Online]. Available: https://link.springer.com/10.3103/S0027134922020047.

[27] C. P. Bee et al., "The raw event format in the ATLAS trigger & DAQ," CERN, Geneva, ATL-DAQ-98-129 ; ATL-D-ES-0019, 1998, p. 26. [Online]. Available: https://cds.cern.ch/record/683741.

[28] G. Aad et al., "Performance of the ATLAS level-1 topological trigger in run 2," *The European Physical Journal C*, vol. 82, no. 1, p. 7, Jan. 2022, ISSN: 1434-6044, 1434-6052. DOI: 10.1140/epjc/s10052-021-09807-0. [Online]. Available: https://link.springer.com/10.1140/epjc/s10052-021-09807-0.

[29] G. Aad et al., "The ATLAS trigger system for LHC Run 3 and trigger performance in 2022," *Journal of Instrumentation*, vol. 19, no. 06, P06029, Jun. 2024, ISSN: 1748-0221. DOI: 10.1088/1748-0221/19/06/P06029. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/19/06/P06029.

[30] M. E. Pozo Astigarraga, "The ATLAS Data Acquisition System int LHC Run 2," in *Proceedings of the XXVI International Symposium on Nuclear Electronics & Computing (NEC'2017)*, Geneva, 2017, p. 7. [Online]. Available: https://cds.cern.ch/record/2292434.

[31] A. G. et al., "The ATLAS experiment at the CERN Large Hadron Collider: A description of the detector configuration for Run 3," *Journal of Instrumentation*, vol. 19, no. 05, P05063, May 1, 2024, ISSN: 1748-0221. DOI: 10.1088/1748-0221/19/05/P05063. arXiv: 2305.16623 [physics]. [Online]. Available: https://iopscience.iop.org/article/10.1088/1748-0221/19/05/P05063.

[32] *Athena*, version 22.0.1, Zenodo, Apr. 2019. DOI: 10.5281/zenodo.2641997. [Online]. Available: https://doi.org/10.5281/zenodo.2641997.

[33] "Athena software repository. "[Online]. Available: https://gitlab.cern.ch/atlas/athena.

[34] "Athena Introduction," ATLAS Software Documentation. [Online]. Available: https://atlas-software.docs.cern.ch/athena/basics/intro/.

[35] CERN for the benefit of the LHCb and ATLAS collaborations. "Gaudi Project Documentation. "[Online]. Available: https://gaudi.web.cern.ch/gaudi/index.html.

[36] CERN for the benefit of the LHCb and ATLAS collaborations. "The framework architecture," Gaudi Project Documentation. [Online]. Available: https://gaudi.web.cern.ch/gaudi/old/GDG_Architecture.html.

[37] P. Calafiura, C. G. Leggett, D. R. Quarrie, H. Ma, and S. Rajagopalan. "The StoreGate: A data model for the atlas software architecture." arXiv: cs/0306089. [Online]. Available: http://arxiv.org/abs/cs/0306089, pre-published.

[38] Walter Lampl, "Optimizing the Energy Measurement of the ATLAS Electromagnetic Calorimeter," Vienna Tech University, Vienna, 2006. [Online]. Available: https://repository.cern/records/7xda0-mcy52.

[39] A. Bazan et al., "The ATLAS liquid argon calorimeter read-out system," *IEEE Transactions on Nuclear Science*, vol. 53, no. 3, pp. 735–740, Jun. 2006, ISSN: 0018-9499. DOI: 10.1109/TNS.2006.873312. [Online]. Available: http://ieeexplore.ieee.org/document/1644935/.

[40] H. Abreu et al., "Performance of the electronic readout of the ATLAS liquid argon calorimeters," *Journal of Instrumentation*, vol. 5, no. 9, P09003–P09003, Sep. 9, 2010, ISSN: 1748-0221. DOI: 10.1088/1748-0221/5/09/P09003. [Online]. Available: https://iopscience.iop.org/article/10.1088/1748-0221/5/09/P09003.

[41] S. Berglund et al., "The ATLAS tile calorimeter digitizer," *Journal of Instrumentation*, vol. 3, no. 1, P01004–P01004, Jan. 29, 2008, ISSN: 1748-0221. DOI: 10.1088/1748-0221/3/01/P01004. [Online]. Available: https://iopscience.iop.org/article/10.1088/1748-0221/3/01/P01004.

[42] S. Asatryan, *Performance studies of minimum bias trigger scintillators in LHC run 3*, CERN, Jan. 2024. [Online]. Available: http://cds.cern.ch/record/2887740.

[43] N. Fernandes, "Development of GPU-accelerated trigger algorithms for the ATLAS experiment at the LHC," M.S. thesis, CERN / Instituto Superior Técnico, 2021. [Online]. Available: https://repository.cern/records/30991-z6f29.

[44] G. E. Moore et al., *Cramming more components onto integrated circuits*, 1965.

[45] R. Schaller, "Moore's law: Past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, Jun. 1997, ISSN: 0018-9235. DOI: 10.1109/6.591665. [Online]. Available: http://ieeexplore.ieee.org/document/591665/.

[46] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974, ISSN: 0018-9200, 1558-173X. DOI: 10.1109/JSSC.1974.1050511. [Online]. Available: https://ieeexplore.ieee.org/document/1050511/.

[47] M. Bohr, "A 30 year retrospective on dennard's MOSFET scaling paper," *IEEE Solid-state Circuits Newsletter*, vol. 12, no. 1, pp. 11–13, 2007, ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785534. [Online]. Available: http://ieeexplore.ieee.org/document/4785534/.

[48] M. Zahran, *Heterogeneous Computing: Hardware and Software Perspectives* (ACM Books 26). New York: Association for computing machinery, 2019, ISBN: 978-1-4503-6097-5. DOI: 10.1145/3281649.

[49] S. Mittal and J. S. Vetter, "A Survey of CPU-GPU Heterogeneous Computing Techniques," *ACM Computing Surveys*, vol. 47, no. 4, pp. 1–35, Jul. 21, 2015, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2788396. [Online]. Available: https://dl.acm.org/doi/10.1145/2788396.

[50] B. Gaster, Ed., *Heterogeneous Computing with OpenCL*. Waltham, MA: Morgan Kaufmann, 2012, 277 pp., ISBN: 978-0-12-387766-6.

[51] "CUDA C++ programming guide," NVIDIA Docs. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[52] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, Las Vegas, NV, USA: IEEE, Jun. 2019, pp. 1–8, ISBN: 978-1-7281-2437-7. DOI: 10.1109/ICESS.2019.8782524. [Online]. Available: https://ieeexplore.ieee.org/document/8782524/.

[53] A. Navarro-Tobar and J. León Holgado, "FPGA implementation of the HL-LHC CMS drift tubes level-1 trigger algorithm," *Journal of Instrumentation*, vol. 20, no. 1, p. C01024, Jan. 1, 2025, ISSN: 1748-0221. DOI: 10.1088/1748-0221/20/01/C01024. [Online]. Available: https://iopscience.iop.org/article/10.1088/1748-0221/20/01/C01024.

[54] J. Brooke, E. Clement, M. Glowacki, S. Paramesvaran, and J. Segal. "LHC triggers using FPGA image recognition." arXiv: 2503.09428 [physics]. [Online]. Available: http://arxiv.org/abs/2503.09428, pre-published.

[55] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*. Amsterdam ; Boston: Morgan Kaufmann Publishers, 2007, 569 pp., ISBN: 978-0-12-370497-9.

[56] V. Bocci, E. Petrolo, A. Salamon, R. Vari, and S. Veneziano, "The coincidence matrix ASIC of the level-1 muon barrel trigger of the ATLAS experiment," *IEEE Transactions on Nuclear Science*, vol. 50, no. 4, pp. 1078–1085, Aug. 2003, ISSN: 0018-9499, 1558-1578. DOI: 10.1109/TNS.2003.815164. [Online]. Available: https://ieeexplore.ieee.org/document/1221925/.

[57] M. Newcomer, "Design and implementation of the ATLAS TRT front end electronics," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 563, no. 2, pp. 306–309, Jul. 2006, ISSN: 01689002. DOI: 10.1016/j.nima.2006.02.168. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0168900206004426.

[58] W.-m. W. Hwu, D. Kirk, and I. El Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*, Fourth edition. Cambridge, MA: Morgan Kaufmann, Elsevier, 2023, 580 pp., ISBN: 978-0-323-98463-8. DOI: 10.1016/C2020-0-02969-5.

[59] "The open standard for parallel programming of heterogeneous systems," OpenCL. [Online]. Available: https://www.khronos.org/opencl/.

[60] "AMD ROCm™ software. "[Online]. Available: https://www.amd.com/en/products/software/rocm.html.

[61] "C++ single-source heterogeneous programming for acceleration offload," SYCL. [Online]. Available: https://www.khronos.org/sycl/.

[62] "What is oneAPI: Demystifying oneAPI for developers," Intel. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/oneapi-what-is-it.html.

[63] R. R. Castro. "SYCL™ performance for nvidia® and AMD GPUs matches native system language," oneAPI. [Online]. Available: https://oneapi.io/blog/sycl-performance-for-nvidia-and-amd-gpus-matches-native-system-language/.

[64] M. Costanzo, E. Rucci, C. G. Sánchez, M. Naiouf, and M. Prieto-Matías, "Comparing performance and portability between CUDA and SYCL for protein database search on NVIDIA, AMD, and intel GPUs," in *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 17, 2023, pp. 141–148. DOI: 10.1109/SBAC-PAD59825.2023.00023. arXiv: 2309.09609 [cs]. [Online]. Available: http://arxiv.org/abs/2309.09609.

[65] "CFD poisson solver achieved up to 1.9x performance improvement by migrating from CUDA to SYCL," oneAPI. [Online]. Available: https://oneapi.io/blog/cfd-poisson-solver-achieved-up-to-1-9x-performance-improvement-by-migrating-from-cuda-to-sycl/.

[66] Y. Bi, S. Xu, and Y. Ma, "Running qiskit on ROCm platform," *EPJ Web of Conferences*, vol. 295, R. De Vita, X. Espinal, P. Laycock, and O. Shadura, Eds., p. 11 022, 2024, ISSN: 2100-014X. DOI: 10.1051/epjconf/202429511022. [Online]. Available: https://www.epj-conferences.org/10.1051/epjconf/202429511022.

[67] "Nvidia CUDA-X libraries," NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries.

[68] "Debugging solutions," NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/debugging-solutions.

[69] "CUDA toolkit documentation," NVIDIA Docs. [Online]. Available: https://docs.nvidia.com/cuda/index.html.

[70] "Latest developer tools topics," NVIDIA Developer Forums. [Online]. Available: https://forums.developer.nvidia.com/c/developer-tools/106.

[71] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *2011 International Conference on Parallel Processing*, Taipei, Taiwan: IEEE, Sep. 2011, pp. 216–225, ISBN: 978-1-4577-1336-1. DOI: 10.1109/ICPP.2011.45. [Online]. Available: http://ieeexplore.ieee.org/document/6047190/.

[72] K. Karimi, N. G. Dickson, and F. Hamze. "A performance comparison of CUDA and OpenCL." arXiv: 1005.2581 [cs]. [Online]. Available: http://arxiv.org/abs/1005.2581, pre-published.

[73] A. Krasznahorkay, C. Leggett, A. S. Mete, S. Snyder, and V. Tsulaia, "GPU usage in ATLAS reconstruction and analysis," *EPJ Web of Conferences*, vol. 245, C. Doglioni, D. Kim, G. Stewart, L. Silvestris, P. Jackson, and W. Kamleh, Eds., p. 5006, 2020, ISSN: 2100-014X. DOI: 10.1051/epjconf/202024505006. [Online]. Available: https://www.epj-conferences.org/10.1051/epjconf/202024505006.

[74] A. H. Zahid, I. Laguna, and W. Le. "Testing GPU numerics: Finding numerical differences between NVIDIA and AMD GPUs." version 1. [Online]. Available: https://arxiv.org/abs/2410.09172, pre-published.

[75] J. H. Davis et al. "Taking GPU Programming Models to Task for Performance Portability." version 4. [Online]. Available: https://arxiv.org/abs/2402.08950, pre-published.

[76] R. T. Mills et al. "PETSc/TAO developments for GPU-based early exascale systems." version 2. [Online]. Available: https://arxiv.org/abs/2406.08646, pre-published.

[77] A. Heakl, S. Hashmi, G. B. Stahl, S. H. E. Han, S. Khan, and A. Mahmoud. "CASS: Nvidia to AMD transpilation with data, models, and benchmark." version 3. [Online]. Available: https://arxiv.org/abs/2505.16968, pre-published.

[78] C. ATLAS, "Technical Design Report for the Phase-II Upgrade of the ATLAS Trigger and Data Acquisition System - Event Filter Tracking Amendment," CERN Document Server, 2022. DOI: 10.17181/CERN.ZK85.5TDL. [Online]. Available: https://cds.cern.ch/record/2802799.

[79] F. Ronchetti, V. Akishina, E. Andreassen, and E. Al., "Efficient high performance computing with the ALICE event processing nodes GPU-based farm," *Frontiers in Physics*, vol. 13, p. 1 541 854, 2025. DOI: 10.3389/fphy.2025.1541854.

[80] R. Aaij and E. Al., "Allen: A high-level trigger on gpus for lhcb," *Computing and Software for Big Science*, vol. 4, p. 7, 2020. DOI: 10.1007/s41781-020-00039-7.

[81] R. Aaij and E. Al., "A comparison of CPU and GPU implementations for the lhcb experiment run 3 trigger," *Computing and Software for Big Science*, vol. 6, p. 1, 2022. DOI: 10.1007/s41781-021-00070-2.

[82] T. Reis and f. t. C. Collaboration, "Developing GPU-compliant algorithms for CMS ECAL local reconstruction during LHC run 3 and phase 2," in *Journal of Physics: Conference Series*, vol. 2438, 2023, p. 12 027. DOI: 10.1088/1742-6596/2438/1/012027.

[83] C. K. Koraka and f. t. C. Collaboration, "Running GPU-enabled CMSSW workflows through the production system," in *EPJ Web of Conferences*, vol. 295, 2024, p. 11 021. DOI: 10.1051/epjconf/202429511021.

[84] T. Voigtlaender, M. Giffels, G. Quast, and E. Al., "Optimized GPU usage in high energy physics applications," in *ACAT 2022*, Villa Romanazzi Carducci in Bari, Italy, Oct. 26, 2022. [Online]. Available: https://indico.cern.ch/event/1106990/contributions/4991345/.

[85] J. Elmsheuser and f. t. A. Collaboration, "Using the ATLAS experiment software on heterogeneous resources," in *ATLAS Software Proceedings*, Kraków, Poland: CERN, 2025. [Online]. Available: https://cds.cern.ch/record/2920934.

[86] M. Atif, Z. Dong, C. Leggett, M. Lin, and V. Tsulaia, "Porting ATLAS fast calorimeter simulation to GPUs with performance portable programming models," *EPJ Web of Conferences*, vol. 295, R. De Vita, X. Espinal, P. Laycock, and O. Shadura, Eds., p. 11 018, 2024, ISSN: 2100-014X. DOI: 10.1051/epjconf/202429511018. [Online]. Available: https://www.epj-conferences.org/10.1051/epjconf/202429511018.

[87] N. Fernandes and f. t. A. Collaboration, "GPU acceleration of the ATLAS calorimeter clustering algorithm," in *Journal of Physics: Conference Series*, vol. 2438, 2023, p. 12 044. DOI: 10.1088/1742-6596/2438/1/012044.

[88] V. Pires, *Acceleration of the ATLAS trigger using graphical processing units*, IST Extended Paper, 2025-EXTPAPER-02, 2025. [Online]. Available: https://www.lip.pt/files/training/papers/2025/pdf/2025-EXTPAPER-02.pdf.

[89] H. Ather, G. Cerati, S. Berkman, and E. Al., "Exploring code portability solutions for HEP with a particle tracking test code," *Frontiers in Big Data*, vol. 7, p. 1 485 344, 2024. DOI: 10.3389/fdata.2024.1485344.

[90] G. M. R. Almeida, "Accelerating track reconstruction using gpus at ATLAS," IST, Lisbon, 2023. [Online]. Available: https://repository.cern/records/0d9ns-mkb75.

[91] Walter Lampl. "ATLAS Computing Twiki - Calorimeter Software and Data Preparation," Calorimeter Event Data Model. [Online]. Available: https://twiki.cern.ch/twiki/bin/viewauth/AtlasComputing/CaloEventDataModel#The_Raw_Data_Model.

[92] M. Furukawa, "ATLAS Liquid Argon Calorimeter Commissioning for LHC Run-3," *EPJ Web of Conferences*, vol. 295, R. De Vita, X. Espinal, P. Laycock, and O. Shadura, Eds., p. 02 004, 2024, ISSN: 2100-014X. DOI: 10.1051/epjconf/202429502004. [Online]. Available: https://www.epj-conferences.org/10.1051/epjconf/202429502004.

Appendix A

Original project proposal

INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

MESTRADO EM INFORMÁTICA

Master of Informatics

Unidade Curricular de "Dissertação/Projeto/Estágio"

Course unit "Thesis/Project/Intership"

Work proposal

Proposta de tema

☑ Dissertação *Thesis* ☐ Projeto *Project* ☐ Estágio *Internship*

Título *Title*

Acceleration of the ATLAS/CERN Calorimeter Data Preparation with GPUs

Palavras-chave *Keywords*

Orientador IPB *IPB Supervisor*	*email*
José Carlos Rufino Amaro	rufino@ipb.pt

Co-orientador IPB *IPB Co-supervisor*	*email*

Co-orientador externo 1 *External co-supervisor 1*	*email*
Patricia Conde Muino	patricia.conde.muino@tecnico.ulisboa.pt

Instituição do Co-orientador externo 1 *Institution of external co-supervisor 1*	País *Country*
Instituto Superior Técnico	Portugal

Co-Orientador externo 2 *External co-supervisor 2*	*email*
Rogerio Aparecido Gonçalves	rogerioag@utfpr.edu.br

Instituição do Co-orientador externo 2 *Institution of external co-supervisor 2*	*Country*
Universidade Tecnológica Federal do Paraná	Brasil

Aluno/*Student*

nº/ *number*	Nome *name*	email *Student email*
64937	Gabriela Paola Sereniski	a64937@alunos.ipb.pt

Objetivos *Goals*

O objetivo desta dissertação de mestrado é contribuir para o desenvolvimento, otimização e avaliação de desempenho de um algoritmo de clustering em GPU para a experiência ATLAS do CERN. Em particular, o foco será na etapa de preparação de dados, que transforma o fluxo de bytes produzido pelos calorímetros ATLAS em estruturas C++ e arrays utilizáveis pelas etapas seguintes do clustering.

INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Descrição *Description*

O LHC é o acelerador de partículas de mais alta energia alguma vez construído. A experiência ATLAS regista as colisões protão-protão e iões pesados produzidas pelo LHC para estudar as partículas mais fundamentais da matéria e as forças entre elas. Uma atualização importante, prevista para 2025-26, aumentará a taxa de colisão do LHC até um fator 7 em relação aos valores nominais, para permitir a aquisição de uma enorme quantidade de dados e ultrapassar os limites da atual compreensão da Natureza.

O sistema de seleção de acontecimentos em linha (trigger) é uma parte crucial da experiência ATLAS. Esse sistema analisa a taxa de acontecimentos, de 40 MHz, selecionando apenas as colisões potencialmente interessantes para análise posterior. Após a atualização do LHC, o aumento estimado da taxa de colisões (e, consequentemente, da quantidade de eventos), leva a tempos de reconstrução de eventos muito mais longos, que não são acompanhados pelo crescimento mais lento esperado da capacidade de computação a um custo fixo. Isto implica uma mudança de paradigma, tornando-se necessário reforçar a exploração do paralelismo na arquitetura dos computadores, acrescentando ao uso da concorrência e do multithreading em CPUs multicore, a exploração de aceleradores de hardware, tais como GPUs ou FPGAs, para executar de forma mais eficiente o código da reconstrução de eventos.

A equipa portuguesa do LIP desenvolveu um algoritmo de clusterização de calorímetros em GPU que demonstrou o excelente potencial dos GPUs como aceleradores de hardware. No entanto, a fase inicial de preparação dos dados ainda não foi portada para GPU, o que esta dissertação pretende fazer.

Metodologia / Plano de trabalhos *Methodology/ Work plan*

O desenvolvimento será feito no âmbito da nova estrutura de reconstrução concorrente do ATLAS, AthenaMT, utilizando as linguagens de programação CUDA e C++, em colaboração com investigadores da equipa do LIP ATLAS, do CERN e de outras instituições de todo o mundo envolvidas neste esforço.

Espera-se que o estudante discuta e apresente o seu trabalho nas reuniões da Colaboração ATLAS, quer por videoconferência, quer no CERN.

Pré-requisitos *Prerequisites*

Mandatórios: conhecimentos de C++ e de progamação concorrente.
Desejáveis: conhecimentos de CUDA e progamação paralela.

Recursos necessários *Resources needed*

Fornecidos pelo LIP e pelo CERN.

Data *Date*

20/11/2024