1st Cycle Integrator Project - Scientific Project

Acceleration of the ATLAS trigger using Graphical Processing Units

Vasco Pires (ist1106341)

Supervisors:
Patrícia Conde Muínio (DF), Nuno dos Santos Fernandes

Instituto Superior Técnico
2025

Abstract

As the High-Luminosity upgrade of the Large Hadron Collider approaches, the ATLAS experiment faces increasing computational demands that surpass the capabilities of traditional CPU-based systems, with the use of hardware accelerators such as GPUs being a possible alternative. In response, this project explores the integration of Marionette — a declarative C++ framework designed to abstract and decouple data interface from memory layout, enabling optimized, customizable, and accelerator-aware data structures — into the ATLAS software to support efficient data transfer between CPU and GPU environments. Focusing on calorimeter reconstruction, two GPU implementations were developed and analyzed: a proof-of-concept reimplementing the simple data structures that are currently used in GPU code and a more complex integration of the full CPU data structure `CaloCluster`. Both implementations achieved numerical correctness, matching the CPU reference outputs across key physical quantities such as E, E_t, η, and ϕ. Performance measurements confirmed that Marionette introduced no measurable overhead, with average execution times of 0.110 ± 0.007 ms (simple case) and 0.114 ± 0.008 ms (complex case), compared to the baseline provided by the pre-existing handwritten data structures of 0.111 ± 0.006 ms. Furthermore, a detailed comparison of the generated PTX assembly verified the functional equivalence of the GPU instruction sets. This work constitutes the first concrete prototype of Marionette within the Athena framework and demonstrates its potential as a robust, zero-cost abstraction for heterogeneous memory environments. By enabling flexible, GPU-optimized data structures while preserving compatibility with legacy code, Marionette positions itself as a valuable asset for future high-performance computing in ATLAS and beyond.

1 The ATLAS Experiment

Founded in 1954, The European Organization for Nuclear Research (CERN) is the world's leading center for high-energy physics research. Located on the Franco-Swiss border near Geneva, CERN operates the Large Hadron Collider (LHC), the biggest particle accelerator built to date [1]. The LHC, constructed between 1998 and 2008, is a 27-kilometer underground tunnel where protons and heavy ions are accelerated to nearly the speed of light and collide at extremely high energies. These collisions allow physicists worldwide to study, analyze, and discover fundamental interactions, testing well-established theories and exploring new ones beyond the Standard Model (SM)[1] [2].

Among the major experiments at the LHC, ATLAS (A Toroidal LHC Apparatus) stands as one of the largest and most complex particle detectors ever constructed. The device itself provides precise measurements of a particle's properties, such as trajectory, momentum, and energy. The LHC, in its current Run 3 phase (2022-2026), is focused on proton-proton collisions at 13.6 TeV and heavy-ion collisions at 5.36 TeV. The ATLAS detector is composed of several key components: an Inner Detector for tracking, a Muon Spectrometer for detecting muons, and Eletromagnetic and Hadronic calorimeters to measure eletromagnetic and hadronic energy, respectively. With its cylindrical geometry and a forward-backward symmetry, the detector provides almost full coverage of the solid angle around the collision point through a set of sub-detectors structured in layers [3, 4].

With collisions occurring every 25 ns, corresponding to data rates of up to 60 TB/s, current computational capabilities are insufficient to record all the information from these events. To address this challenge, the ATLAS experiment employs a Trigger System to filter and process relevant events in real time. This system incorporates a Level-1 hardware trigger, which reduces the initial 40 MHz bunch crossing rate[2] to around 100 kHz by performing a coarse filtering using simpler algorithms. The Level-1 also identifies potential regions of interest for a more targeted scan in the subsequent phase. Following this, the High-Level Trigger (HLT) utilizes more sophisticated, offline-like software-based selection algorithms running on a CPU-based computing farm to further reduce the rate to 3 kHz. The selection is based on generic physical signatures, such as high transverse momenta, energetic electrons and muons, jets, photons, among others. These algorithms are designed to perform detailed event reconstruction, enabling the efficient storage and analysis of relevant data [3].

1.1 GPUs in ATLAS

The development of efficient data processing techniques is crucial for handling the vast datasets generated by ATLAS, particularly as the LHC prepares for its High Luminosity upgrade (HL-LHC), which will significantly increase its luminosity (i.e. collision rates). To address this challenge, new approaches based on hardware accelerators, which may

[1]Theoretical model/framework that describes the electromagnetic, weak, and strong fundamental forces, as well as the elementary particles they act upon.

[2]The bunch crossing rate refers to how often particle bunches pass through the detector.

offer better computational performance for certain classes of problems, have been proposed to optimize event processing. Before discussing specific applications, it is important to introduce the concept of calorimeter reconstruction. This process involves converting raw detector signals into calibrated energy measurements, clustering energy deposits, and ultimately identifying particle candidates. Calorimeter reconstruction is computationally intensive due to the high granularity of the detectors and the complexity of the underlying algorithms [5].

For the particular case of calorimeter reconstruction, a GPU-based approach implemented using the CUDA (Compute Unified Device Architecture) programming language, which will be further explained, has demonstrated a significant improvement in processing times. However, the underlying structures, in which the data generated is stored, still reside in both CPU and GPU memory and pose a critical bottleneck for code efficiency, requiring costly data transfer and transformation between the CPU structures and GPU ones during execution [5].

1.2 Objectives of this work

To fully exploit the benefits of GPU-based processing, it is necessary to rethink how data is structured and accessed. The ATLAS software, built around the Athena[3] Framework, currently consists of a mix of legacy CPU-based code, newly ported GPU-based code, and eventually future code, creating a fragmented software environment. Without a unified data representation, maintaining compatibility across different implementations becomes increasingly complex.

With this problem in mind, CERN researcher and PhD student, Nuno dos Santos Fernandes, created Marionette (Memory Abstracted Representation with Interfaces in Objects Necessitating Extensively Templated Types EDM), a framework for describing general data structures, designed to provide both CPU and GPU-compatible representations of the event data with optimized transfers and conversions, while ensuring an intuitive and accessible implementation for both new and experienced programmers.

This project aims to achieve the first integration of Marionette in ATLAS, specifically within calorimeter reconstruction, to assess its effectiveness in reducing bottlenecks, optimizing performance, and evaluating its potential to unify implementations.

2 GPU Fundamentals

To better understand the discussions in this paper, it is helpful to first grasp the basics of GPU technology and its relevance in high-energy physics data processing.

The standard CPUs our computers use to process data and run daily are optimized for sequential, low-latency tasks, employing a small number of fully independent threads, whereas GPUs are designed to perform thousands of operations simultaneously. To achieve this, GPUs employ a parallel architecture that excels at handling large volumes, making them ideal for some kinds of problems involving complex computations, including the analysis of collision data in experiments such as ATLAS [7].

A GPU is composed of many smaller, simpler cores. In NVIDIA GPUs, these are known as CUDA cores, which will be used in this work. These cores are organized into groups called blocks, and each block is composed of multiple threads, which are basic units of execution capable of running simultaneously. The threads within a block have their own private memory and are able to share memory in order to work on the same task in parallel, and the blocks are then grouped together into a grid. The execution model for these parallel tasks is defined through kernels, which specify the operations that each thread will perform. GPUs follow the Single Instruction, Multiple Threads (SIMT) model, meaning that threads within a warp (subset of threads within a block) execute the same instructions synchronously, maximizing efficiency. Kernels are, concunrrently, executed by multiple threads, allowing GPUs to process vast amounts of data efficiently [7].

Another important aspect of GPU performance is efficient data access. Regarding memory organization, GPUs outperform when using Structure of Arrays (SoA) instead of Array of Structures (AoS). In AoS, each structure stores all the attributes for a single entity, which can lead to inefficient memory access. On the other hand, in SoA, each attribute is stored in a separate array, allowing threads to access contiguous memory locations. This improves coalesced reading from memory[4], where threads can efficiently access data in a way that minimizes waiting time and maximizes data throughput, leading to faster computations and fewer delays due to memory access bottlenecks [9].

[3]Athena is the software framework used by the ATLAS experiment at CERN for event processing, reconstruction, and analysis. It is built on top of the Gaudi architecture and supports both C++ and Python components. [6]

[4]Memory coalescing refers to the process of combining multiple memory read accesses into a single transaction, reducing latency and improving throughput by optimizing how threads access data in memory. [8]

3 Introduction to Marionette

As stated before, this work aims to integrate Marionette into the ATLAS experiment, not only to generalize the structures in which our data resides and optimize the memory transfers between CPU-GPU but also to efficiently port them for GPU processing. Before diving into the actual integration, we first illustrate a simple example to demonstrate Marionette's behavior and implementation, using a scenario where two particle structures are declared.

```cpp
struct Particle_Cartesian { //One single particle observed in the real world
   float px, py, pz, mass, charge;
};
struct Particle_Detector { //One single particle captured by the detector
   float pt, eta, phi, energy, charge;
};
```

Listing 1: Handwritten Data Structure (AoS)

```cpp
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(energy, Energy, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(charge, Charge, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(mass, Mass, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(eta, Eta, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(phi, Phi, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(px, Px, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(py, Py, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(pz, Pz, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(pt, Pt, float);

using Marionette::InterfaceDescription::PropertyList;
using Particle_Cartesian = PropertyList<Px, Py, Pz, Mass, Charge>;
using Particle_Detector = PropertyList<Pt, Eta, Phi, Energy, Charge>;
```

Listing 2: Marionette-Optimized Structure

In a typical c++ data structure, attributes are explicitly defined within each structure, making it difficult to systematically handle or reuse property definitions across different data types. In contrast, Marionette organizes data using property description classes, which allow attributes to be specified programmatically and reused across multiple structures, enabling a more flexible and modular approach. This abstraction helps maintain consistency and allows specifying the behaviour of data structures in a completely generic way.

While Marionette was implemented to support a broad spectrum of memory contexts, to fully support the end developer needs, we will use the SoA format, as it optimizes performance and memory access. However, traditional programming paradigms and human intuition are more accustomed to an AoS organization. To align these approaches, the author leveraged programming techniques, such as metatemplate programming and CRTP[5], that allow developers to write code as if they were working in an AoS framework, while the underlying data is efficiently managed in a SoA structure. In order to properly compare both implementations, we will now, also, consider a SoA approach for the Handwritten version.

```cpp
//std::vector<T> is a standard class that allocates an array of type T on the struct
struct VectorCPU_Cartesian { // Structure for storing multiple particles
   std::vector<float> px, py, pz, mass, charge;
};
struct VectorCPU_Detector {
   std::vector<float> pt, eta, phi, energy, charge;
};

//GPU_vector<T> is a class we could write that allocates an array of type T on the GPU
struct VectorGPU_Cartesian {
   GPU_vector<float> px, py, pz, mass, charge;
};
struct VectorGPU_Detector {
   GPU_vector<float> pt, eta, phi, energy, charge;
};
```

[5]CRTP (Curiously Recurring Template Pattern) is a technique where a class template takes its derived class as a parameter, allowing the base class to call functions in the derived class at compile time. This can improve efficiency by avoiding runtime overhead. As this concept can be abstract and nontrivial to understand initially, a more detailed explanation can be found in the following website [10].

Listing 3: Handwritten Data Structure (SoA)

```cpp
using CPUParticle_Cartesian = Marionette::Collection<CPULayout, Particle_Cartesian>
using CPUParticle_Detector = Marionette::Collection<CPULayout, Particle_Detector>
using GPUParticle_Cartesian = Marionette::Collection<GPULayout, Particle_Cartesian>
using GPUParticle_Detector = Marionette::Collection<GPULayout, Particle_Detector>
```

Listing 4: Marionette-Optimized Structure

This comparison highlights the redundancy in the handwritten implementation, where four separate structures and boilerplate code are required to manage data storage across CPU and GPU. In contrast, Marionette simplifies this process by allowing us to specify only the data structure (second parameter) and the desired layout (first parameter), which will specify how the underlying content will be stored in memory. In Marionette's implementation, CPULayout would be equivalent to storing the attributes in separate std::vectors, while GPULayout would store them in the hypothetical GPU_vector used before.

```cpp
cudaMemcpy(VectorGPU_Cartesian.px, VectorCPU_Cartesian.px, size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(VectorGPU_Cartesian.py, VectorCPU_Cartesian.py, size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(VectorGPU_Cartesian.pz, VectorCPU_Cartesian.pz, size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(VectorGPU_Cartesian.mass, VectorCPU_Cartesian.mass, size * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(VectorGPU_Cartesian.charge, VectorCPU_Cartesian.charge, size * sizeof(float), cudaMemcpyHostToDevice);
```

Listing 5: Handwritten Data Transfer

```cpp
GPUParticle_Cartesian = CPUParticle_Cartesian
```

Listing 6: Marionette-Optimized Data Transfer

Another fundamental aspect of Marionette, as mentioned earlier, is the optimization of data transfers between CPU and GPU. The traditional approach requires explicitly requesting the copy of all attribute arrays using cudaMemcpy(), which is both error-prone, since one might accidentally omit an attribute, and difficult to scale across multiple classes or when modifying the data structure. In contrast, Marionette abstracts these operations, making the requests automatically for each attribute, enabling seamless and implicit memory transfers within a single assignment.

As expected, this simple example doesn't showcase all of Marionette's functionalities, but throughout this paper, more aspects will become evident. However, it is important to bear in mind that the focus is on the integration of Marionette, rather than providing a comprehensive description of the framework itself. For a more detailed overview of Marionette's design and capabilities, an upcoming dedicated paper will provide a full technical description.

4 Analysis

With the foundations of GPU computing and the Marionette framework established, we now turn to the core objective of this report: evaluating the framework's implementation and its impact within the ATLAS software.

4.1 Proof of Concept

Given the inherent complexity of the ATLAS software environment, particularly due to its extensive pre-existing codebase, integrating a new library under active development necessitates careful consideration. To minimize potential issues and expedite debugging, it was essential to select a file that would allow for clear and manageable testing. For that, the first prototype implementation was conducted in the following file: BasicGPUClusterInfoCalculatorImpl.cu, containing four well-defined kernels[6], that interact with cluster[7] properties, thereby providing an accessible framework for evaluating Marionette's functionality.

The original file implementation stored the cluster-related data in a simple Structure of Arrays (SoA) format, where an object of type struct holds multiple arrays, each representing an attribute of said cluster. The structure was defined as follows:

[6]Refer to section 2 where the concept was first introduced.

[7]A cluster refers to a group of adjacent calorimeter cells with energy deposits, originating from a single particle's interaction with the detector material.

```cpp
struct ClusterInfoArr {
    int number;
    float clusterEnergy[NMaxClusters];
    float clusterEt[NMaxClusters];
    float clusterEta[NMaxClusters];
    float clusterPhi[NMaxClusters];
    int seedCellID[NMaxClusters];
};
```

Listing 7: Original Implementation

In line with the objectives of this report, this structure is redefined using Marionette to enable a more flexible design. The reimplementation involves declaring each attribute independently and subsequently organizing them into a templated collection that behaves similarly to a structure. Marionette facilitates this approach through its declarative macros[8] and generic collection interfaces. The equivalent implementation using the framework is shown below:

```cpp
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(clusterEnergy, ClusterEnergy, NMaxClusters, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(clusterEt, ClusterEt, NMaxClusters, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(clusterEta, ClusterEta, NMaxClusters, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(clusterPhi, ClusterPhi, NMaxClusters, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY_SIZED(seedCellID, SeedCellID, NMaxClusters, int);

using Marionette::InterfaceDescription::PropertyList;
using CollectionProperties = PropertyList<ClusterEnergy, ClusterEt, ClusterEta, ClusterPhi, SeedCellID>;

template <class Layout>
using OurCollectionBase = Marionette::Collections::Collection<Layout, CollectionProperties>;

template <class Context>
using OurCollection = OurCollectionBase<Marionette::LayoutTypes::DynamicStructInContext<Context, int, int>>;

using GPUCollection = OurCollection<Marionette::MemoryContexts::CUDAStandardGPU>;
```

Listing 8: Marionette Implementation

The GPUCollection now provides a memory-efficient abstraction of all cluster attributes, with explicit GPU allocation through the memory context CUDAStandardGPU (specifically designed for seamless integration with CUDA). Despite abstracting and optimizing data transfers between structures and execution environments, since the original implementation already employed a Structure of Arrays (SoA) format and the kernel logic in this context is relatively self-contained, Marionette is expected to achieve comparable execution times by internally utilizing a similar SoA layout.

These restructured collections can now be utilized within the computation kernels. As aforementioned, Marionette provides an intuitive programming interface by preserving an AoS organization while operating within a SoA framework. This design simplifies the integration process, since the resulting implementation closely resembles the structure of the legacy code. Nonetheless, ensuring correct functionality requires careful attention to framework-specific implementation details.

To really grasp the impact of Marionette's implementation, the initial step is to verify that its output matches that of the original CPU version, which serves as the reference, ensuring the program's correctness has been preserved. With this in mind, with the assistance of the ROOT tool, the following four graphs were generated regarding the 4 attributes of the structure in question, Energy (E), Transverse Energy (Et — energy perpendicular to the beam axis), Eta (η — a coordinate describing the particle's angle relative to the beam axis), and Phi (ϕ — the azimuthal angle around the detector's axis)[9]:

[8] Macros are preprocessor directives that define patterns to be expanded into code before compilation, enabling code reuse and reducing boilerplate. For more information refer to the following paper [11].

[9] In order to visualize the variables mentioned, section A.1.1 provides a diagram of the ATLAS coordinate system and respective variables.

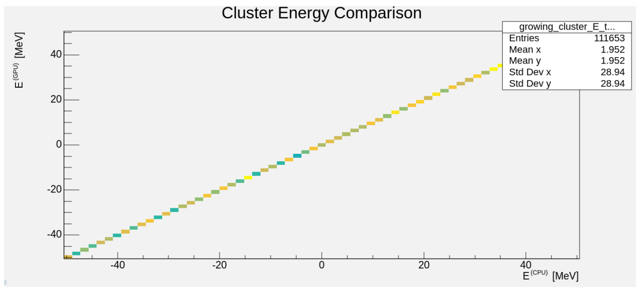


Figure 1: Cluster Energy : Marionette (GPU) vs. Reference (CPU) - Proof of Concept

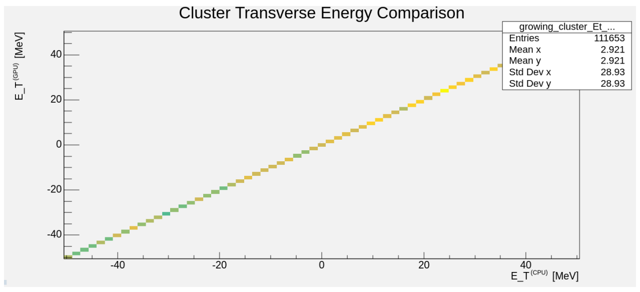


Figure 2: Cluster Transveral Energy : Marionette (GPU) vs. Reference (CPU) - Proof of Concept

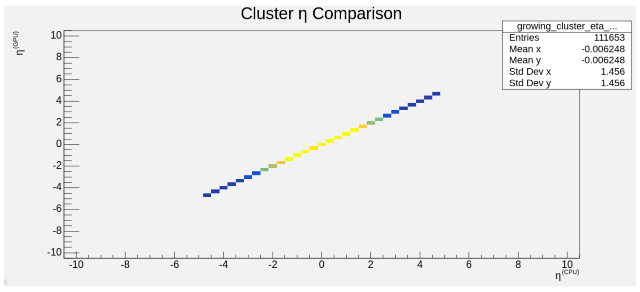


Figure 3: Cluster Eta : Marionette (GPU) vs. Reference (CPU) - Proof of Concept

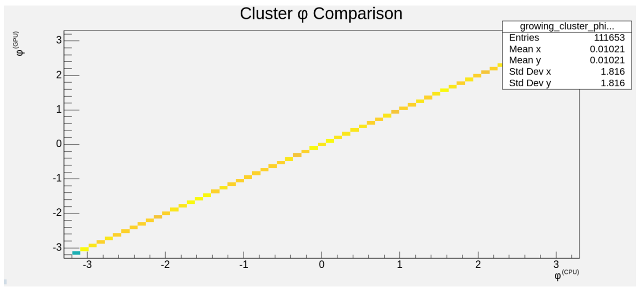


Figure 4: Cluster Phi : Marionette (GPU) vs. Reference (CPU) - Proof of Concept

Through these graphs, we can infer that both implementations produce results that are consistent up to differences due to floating-point error accumulation, which arise from the limited precision in representing decimal numbers in computer calculations.

Following the output validation, we now proceed to compare the execution times of both versions. This allows us to determine whether the framework introduces any performance overhead or compromises the efficiency of the system. The following table holds the total execution time, after undergoing outlier removal and data treatment for 3000 events of type $t\bar{t}$[10] :

Implementation	Mean Execution Time (ms)
Original	0.111 ± 0.006
Marionette	0.110 ± 0.007

Table 1: Per-event total kernel execution time: Original vs. Marionette (Proof-of-Concept)

By examining the table shown, the execution times prove to be similar and therefore serve as an indicator of the expected behavior of the framework. One important remark regarding the presented data is that the observed discrepancies mainly stem from the execution environment, rather than from differences in the actual performance, due to the number of users that could be simultaneously executing code within the CERN server.

To achieve a bulletproof analysis shielded from externnal factors, we can look at the .ptx[11] files from both legacy and Marionette versions. Given the intrinsic complexity of PTX assembly, said files will not be explicitly displayed in this report. Nevertheless, a careful comparison of the instruction sequences confirms that the GPU receives identical instructions in both implementations, ensuring equivalent performance. This allows us to confirm that Marionette operates as intended with zero-cost abstraction.

[10] A $t\bar{t}$ event refers to the production of a top quark and its antiparticle (antitop quark) in a proton-proton collision, commonly studied at the LHC due to their relevance in testing the Standard Model [12]. This type of events was chosen due to the large number of particles produced in their decay, which results in a greater number of clusters and allows for a more complete and robust testing of Marionette's framework.

[11] PTX (Parallel Thread Execution) is an intermediate assembly-like language used by NVIDIA's CUDA platform. It defines low-level instructions that are later translated into executable GPU code. The following NVIDIA blog, offers a more detailed explanation with a clear example [13].

4.2 Deeper Implementation

With this prototype showcasing promising results, while noting that the previous test case used was intentionally simplistic, we now logically proceed to refactor deeper and more complex structures with broader codebase impact, to further validate the framework.

Before diving into the technical details, it's important to clarify the context surrounding this implementation. In the ATLAS calorimeter reconstruction chain, clustering algorithms are used to group together neighboring energy deposits (cells) that likely originate from the same particle shower. One such method is the *Topo-Automaton Clustering* (TAC), a GPU-optimized algorithm that classifies calorimeter cells based on their signal-to-noise ratio: cells with strong signals are marked as *seeds*, moderate signals as *growing cells*, and low signals as *terminal cells*. Clusters are formed by starting from seeds and recursively adding adjacent growing and terminal cells. TAC implements this logic using a sequence of GPU kernels, organized into four main stages: *Cluster Growing*, *Basic Cluster Property Calculation*, *Cluster Splitting*, and *Cluster Moments Calculation* [14].

The focus of this work lies specifically on the calculation of the basic kinematic properties of the clusters, such as total energy (E), transverse energy (E_T), pseudorapidity (η), and azimuthal angle (ϕ). By concentrating on this specific stage, we ensure that the computational benefits and abstractions introduced by Marionette are applied where they can have real analytical impact, without needing to reimplement the entire clustering pipeline at once. This modular approach also makes debugging and validation significantly more manageable. As a continuation of the previous proof-of-concept, this implementation will impact the same source file — `BasicGPUClusterInfoCalcImpl.cu` — now adapted to handle a broader data structure.

Building on this foundation, the next step involves reimplementing the **CaloCluster**[12] structure, which encapsulates not only the previously refactored **ClusterInfoArr** structure, but also a wide range of attributes and functionalities associated with individual clusters in the calorimeter. More specifically, it includes geometric properties that describe the position and spread of the cluster within the detector; energy-related quantities, such as the total and peak energy deposits; and detailed data from the various calorimeter sampling layers[13]. Additionatlly, CaloCluster also stores timing information, useful for distinguishing between different interactions, as well as several derived variables — known as moments — that help characterize the cluster's shape and internal structure. This structure is central to ATLAS's calorimeter reconstruction process, making it an ideal candidate for a more comprehensive test of Marionette's capabilities.

The adopted protocol followed the same principles previously discussed. It began with a thorough review and understanding of the legacy code, which was already structured in a SoA format, to accurately emulate the required behaviour. This was followed by a decomposition of the implementation into logical blocks, grouping together functions that served related purposes or operated on common data. Next, each block was carefully examined to determine the most suitable strategy for reimplementation using Marionette. Implementation then proceeded by leveraging techniques such as CRTP, metatemplate programmming, and macros (introduced earlier in Section 3 and Subsection 4.1), with functions organized into a templated collection defined by a clear set of properties, resulting in a final implementation of approximately 1141 lines, which represents a reduction of nearly 25% compared to the 1523 lines of the original version. As a final stage, the target file was adapted to interface with the newly constructed collection—a minor adjustment, given that the major necessary kernel-level changes had already been addressed during the proof of concept phase.

Beyond the implementation, a significant portion of the work focused on testing and debugging. Given the structure's complexity and the library's early-stage development, `CaloCluster` posed integration challenges suggesting new use cases that were not yet fully supported by the library and revealing some bugs and edge cases that require further consideration. Resolving some of these issues required persistent debugging and iteration, often adjusting both the framework and the implementation of the data structure within it. Progress came incrementally, with challenges gradually uncovered and addressed through careful testing. Additionally, the limitations found currently prevent a subset of the structure's properties from being fully supported with the intended semantics, such as - keeping links to elements of other collections, since support for such functionality is nontrivial in the context of hardware acceleration and the author is still developing a solution for this. Nevertheless, all essential behaviors required to ensure the code executes correctly, on the target file, were successfully implemented. This phase was essential not only for ensuring correctness and providing feedback to the author, but also for building a thorough understanding of the framework.

[12]Given it's dimensions, it is not possible to showcase the full original data structure in this report. For a complete reference, please consult [15].

[13]A sampling layer (or sample) refers to one of the multiple segmented layers of the calorimeter detector, each designed to record energy deposited by particles as they traverse different depths of the detector.

Moving onto the analysis itself, analogous to the validation procedure carried out ealier, the priority is to verify that the Marionette-based implementation of the full `CaloCluster` structure yields results consistent with the original CPU reference. Using the ROOT tool, a comparison over the same set of cluster-level properties previously analyzed (E, Et, η and ϕ) was performed, reproducing the same four exact plots shown beforehand. The linear correlation between the two implementations exhibits a strong numerical agreement, confirming that despite the structural complexity, the core behavior that analyzes the physical phenomena is preserved. As before, any minor deviations can be attributed to floating-point precision limits, which are inherent to numerical computations. For a detailed view of the plots obtained, refer to A.2.1

With output validated, following the same methodology, we now focus on performance. Employing the previously established dataset of 3000 $t\bar{t}$ events and consistent runtime conditions, execution times were recorded for both implementations.

Implementation	**Mean Execution Time (ms)**
Original	0.111 ± 0.006
Marionette	0.114 ± 0.008

Table 2: Per-event total kernel execution time: Original vs. Marionette (Deeper Implementation)

With the execution times remaining closely aligned, the results obtained further reinforce the previously drawn conclusions - Marionette introduces no significant performance overhead and maintains efficiency. The minor variations, still within the uncertainty, observed can be explained due to fluctuations in the shared execution environment, rather than actual performance differences, as aforementioned. For a detailed breakdown of the execution times, see A.2.2.

Moreover, a renewed inspection of the generated PTX files unequivocally validated the framework's efficiency. By comparing said files from both implementations, it was confirmed that the set of GPU instructions remained functionally equivalent. The same arithmetic operations, memory access patterns, and control flow were preserved, ensuring that no major unintended transformations were introduced during compilation.

The only variations observed were memory offsets - small differences in where data is allocated in memory - which are expected due to internal structural rearrangements dictated by the goal of replicating the entirety of the CPU data structures. These limited differences do not affect the behavior or performance of the kernel, as confirmed above, thereby substantiating that Marionette works with zero-cost abstraction, even in the context of more complex and feature-rich structures. To better understand the comparison that was conducted, consult A.2.3.

5 Conclusion & Main Insights

As the High-Luminosity era of CERN approaches, the computational demands placed on the ATLAS experiment continue to grow. Meeting these challenges requires not only hardware acceleration, particularly through GPUs, but also a thoughtful redesign of the surrounding software system. In this context, porting algorithms alone is insufficient; the data structures in which the essential retrieved information lies must also be reimplemented for GPU compatibility and performance. Since portions of legacy CPU-based code will remain in use, and given that developers have diverse approaches to organizing and implementing code, the need for a flexible, modular, and efficient interface between CPU and GPU becomes imperative.

Marionette emerges as a response to these needs. It introduces a declarative, abstraction-oriented framework that allows developers to define data structures using an Array of Structures (AoS) interface while internally adopting the memory layout best suited to the target architecture.

To evaluate Marionette, a two-stage implementation strategy was employed. First, a simple, isolated kernel logic example was selected to ensure an accessible debugging environment. This initial test demonstrated that Marionette could reproduce the original algorithm's behavior with no loss of correctness or performance, confirming its zero-cost abstraction. Subsequently, a more complex structure—`CaloCluster`—was ported, further testing Marionette's capabilities. Despite the increased complexity of the structure and its embedded logic, Marionette successfully supported the full execution of the target file, validating its core functionality while also exposing key areas for future improvement within the framework. Execution time measurements further corroborate the framework's efficiency. Also, a detailed comparison of the generated PTX code confirmed that the GPU instructions issued by both the legacy and Marionette implementations are equivalent, ruling out any hidden overhead at the low-level execution stage. This demonstrates that the abstraction introduced by the framework enables a powerful and future-proof design.

In summary, the results of both the simple and complex implementation scenarios confirm that Marionette maintains correctness and performance levels, enabling a flexible and GPU-optimized data handling strategy. These qualities validate its integration into the evolving ATLAS software, particularly as demands for heterogeneous computing continue to grow.

The work carried out throughout this report, ranging from analysis and debugging to implementation, has also revealed potential enhancements to Marionette that could improve the end developer experience and extend its applicability. More broadly, this project represents the first concrete use case of Marionette within the Athena framework, as such, it not only serves as a prototype but also provides a valuable proof of concept that validates the library's design goals under realistic conditions.

Overall, Marionette has proven to be a robust and efficient tool for abstracting heterogeneous memory models and managing cross-device data structures, establishing itself as a valuable asset for the future of high-performance computing in ATLAS and beyond.

References

[1] "Where did it all begin?". https://home.cern/about/who-we-are/our-history. Accessed: 2025-04-07.

[2] "Facts and figures about the LHC". https://home.cern/resources/faqs/facts-and-figures-about-lhc. Accessed: 2025-04-07.

[3] Collaboration Th Atlas et al. The atlas experiment at the cern large hadron collider: a description of the detector configuration for run 3. *Journal of Instrumentation*, 19(5):1–222, 2024.

[4] " The LHC lead-ion collision run starts ". https://home.cern/news/news/experiments/lhc-lead-ion-collision-run-starts. Accessed: 2025-04-07.

[5] P. Conde Muíño and on behalf of the ATLAS Collaboration. Multi-threaded algorithms for gpgpu in the atlas high level trigger. *Journal of Physics: Conference Series*, 898(3):032003, oct 2017.

[6] " ATLAS Software Documentation ". https://atlas-software.docs.cern.ch/athena/. Accessed: 2025-04-07.

[7] Axel Habermaier and Alexander Knapp. On the correctness of the simt execution model of gpus. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 316–335, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[8] Jack W. Davidson and Sanjay Jinturkar. Memory access coalescing: a technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 186–195, New York, NY, USA, 1994. Association for Computing Machinery.

[9] Robert Strzodka. Chapter 31 - abstraction for aos and soa layout in c++. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 429–441. Morgan Kaufmann, Boston, 2012.

[10] "What is the curiously recurring template pattern (CRTP)?". https://stackoverflow.com/questions/4173254/what-is-the-curiously-recurring-template-pattern-crtp. Accessed: 2025-04-15.

[11] Stanislav Sỳkora. Writing c/c++ macros: Rules, tricks and hints. *Stan's Library*, 1, 2004.

[12] "CMS finds unexpected excess of top quarks ". https://home.cern/news/news/physics/cms-finds-unexpected-excess-top-quarks. Accessed: 2025-04-23.

[13] "Understanding PTX, the Assembly Language of CUDA GPU Computing". https://developer.nvidia.com/blog/understanding-ptx-the-assembly-language-of-cuda-gpu-computing/. Accessed: 2025-04-23.

[14] "Search for the higgs boson at ATLAS/LHC in WH associated production and decay to b-quark pairs". http://hdl.handle.net/10451/34140. Accessed: 2025-04-20.

[15] "CaloCluster_v1 Data Structure". https://acode-browser.usatlas.bnl.gov/lxr/source/athena/Event/xAOD/xAODCaloEvent/xAODCaloEvent/versions/CaloCluster_v1.h. Accessed: 2025-04-20.

A Appendix

A.1 Proof of Concept

A.1.1 ATLAS Coordinate System

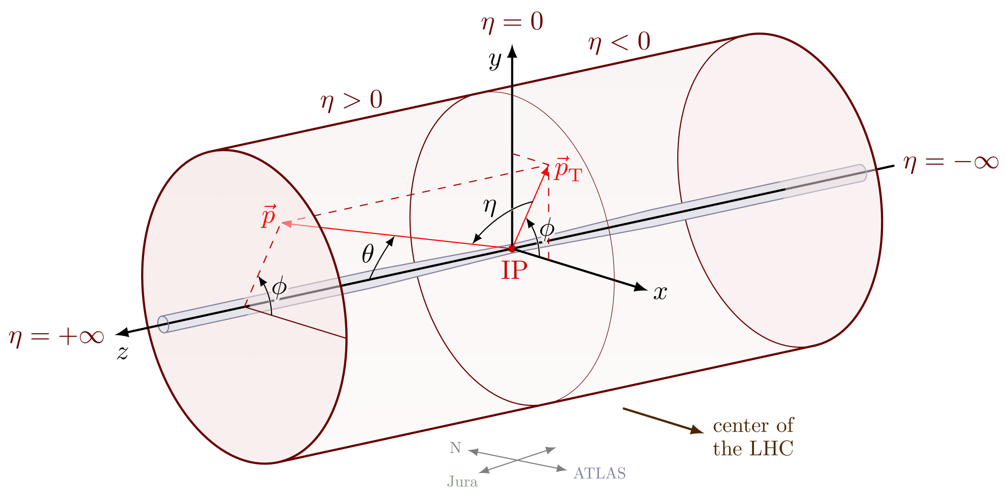
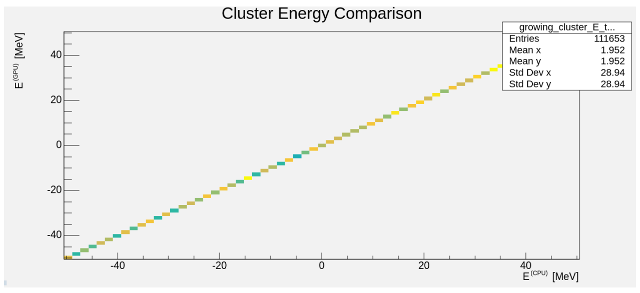


Figure 5: ATLAS Coordinate System and variables associated.

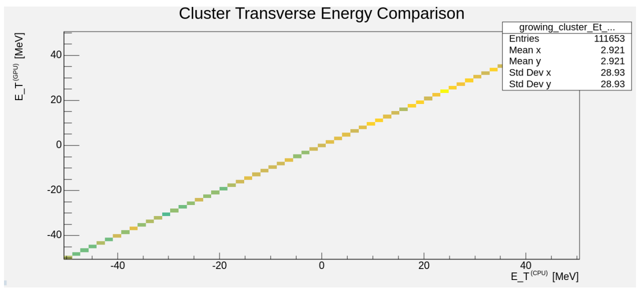
A.2 Deeper Implementation

A.2.1 Plots



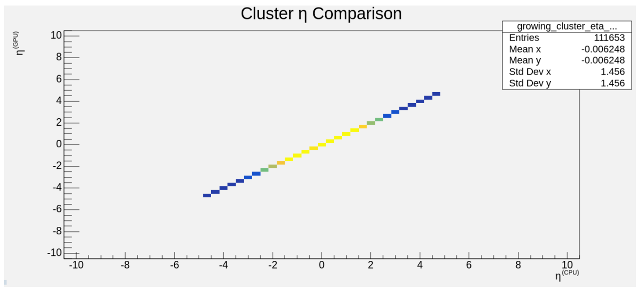
growing_cluster_E_t...	
Entries	111653
Mean x	1.952
Mean y	1.952
Std Dev x	28.94
Std Dev y	28.94

Figure 6: Cluster Energy : Marionette (GPU) vs. Reference (CPU) - Deeper Implementation



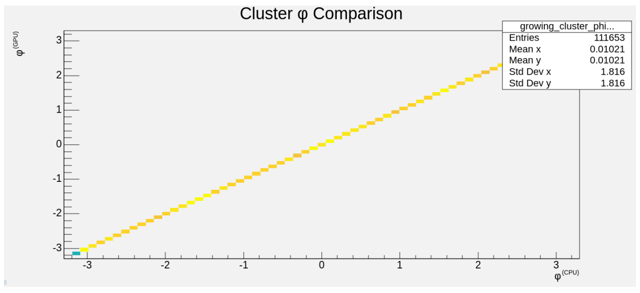
growing_cluster_Et_...	
Entries	111653
Mean x	2.921
Mean y	2.921
Std Dev x	28.93
Std Dev y	28.93

Figure 7: Cluster Transveral Energy : Marionette (GPU) vs. Reference (CPU) - Deeper Implementation



growing_cluster_eta_...	
Entries	111653
Mean x	-0.006248
Mean y	-0.006248
Std Dev x	1.456
Std Dev y	1.456

Figure 8: Cluster Eta : Marionette (GPU) vs. Reference (CPU) - Deeper Implementation



growing_cluster_phi...	
Entries	111653
Mean x	0.01021
Mean y	0.01021
Std Dev x	1.816
Std Dev y	1.816

Figure 9: Cluster Phi : Marionette (GPU) vs. Reference (CPU) - Deeper Implementation

A.2.2 Execution Time Histogram

The following histogram provides a full visualization of the execution time distributions for both implementations.

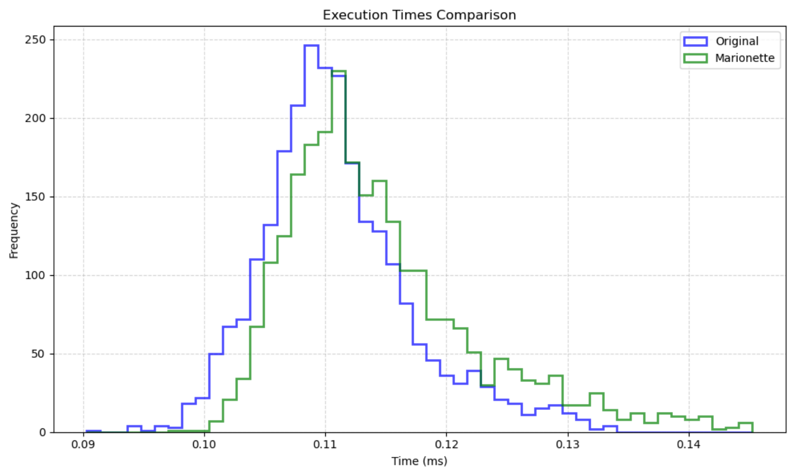


Figure 10: Histogram comparing the execution times of the Original and the Marionette-based versions.

A.2.3 Kernel Comparison: Original vs Marionette

The following listing show a minimal GPU kernel — `seedCellPropertiesKernel` — which is one of the four kernels implemented. This specific kernel is responsible for initializing cluster properties. Given its simplicity and direct logic, it offers a clear view of the set of GPU instructions.

Listing 9: Legacy PTX Kernel

```
.entry seedCellPropertiesKernel(
    .param .align 8 param_0[8],
    .param .align 8 param_1[8],
    .param .align 8 param_2[8]
)
{
    ld.param.u64     %rd19, [...];
    ld.param.u64     %rd20, [...];
    ld.param.u64     %rd21, [...];
    cvta.to.global.u64       %rd1, %rd21;
    cvta.to.global.u64       %rd2, %rd20;
    cvta.to.global.u64       %rd3, %rd19;
    mov.u32          %r18, %ntid.x;
    mov.u32          %r19, %ctaid.x;
    mov.u32          %r20, %tid.x;
    mad.lo.s32       %r38, %r19, %r18, %r20;
    mov.u32          %r21, %nctaid.x;
    mul.lo.s32       %r2, %r21, %r18;
    ld.global.u32    %r3, [%rd1];
    setp.ge.s32      %p1, %r38, %r3;
    @%p1 bra         $L__BB0_17;

    add.s32          %r22, %r3, %r2;
    add.s32          %r23, %r38, %r2;
    not.b32          %r24, %r23;
    add.s32          %r25, %r22, %r24;
    div.u32          %r4, %r25, %r2;
    add.s32          %r26, %r4, 1;
    and.b32          %r37, %r26, 3;
    setp.eq.s32      %p2, %r37, 0;
    @%p2 bra         $L__BB0_6;

    mul.wide.s32     %rd22, %r38, 4;
    add.s64          %rd40, %rd2, %rd22;
    mul.wide.s32     %rd5, %r2, 4;
    add.s64          %rd23, %rd1, %rd22;
    add.s64          %rd39, %rd23, 524292;
}
```

Listing 10: Marionette PTX Kernel

```
.entry seedCellPropertiesKernel(
    .param .align 8 param_0[8],
    .param .align 8 param_1[8],
    .param .align 8 param_2[8]
)
{
    ld.param.u64     %rd22, [...];
    ld.param.u64     %rd23, [...];
    ld.param.u64     %rd24, [...];
    cvta.to.global.u64       %rd1, %rd24;
    cvta.to.global.u64       %rd2, %rd23;
    cvta.to.global.u64       %rd3, %rd22;
    mov.u32          %r18, %ntid.x;
    mov.u32          %r19, %ctaid.x;
    mov.u32          %r20, %tid.x;
    mad.lo.s32       %r41, %r19, %r18, %r20;
    mov.u32          %r21, %nctaid.x;
    mul.lo.s32       %r2, %r21, %r18;
    ld.global.u32    %r3, [%rd1];
    setp.ge.s32      %p1, %r41, %r3;
    @%p1 bra         $L__BB0_17;

    add.s32          %r22, %r3, %r2;
    add.s32          %r23, %r41, %r2;
    not.b32          %r24, %r23;
    add.s32          %r25, %r22, %r24;
    div.u32          %r4, %r25, %r2;
    add.s32          %r26, %r4, 1;
    and.b32          %r40, %r26, 3;
    setp.eq.s32      %p2, %r40, 0;
    @%p2 bra         $L__BB0_6;

    mul.wide.s32     %rd25, %r41, 4;
    add.s64          %rd26, %rd2, %rd25;
    add.s64          %rd44, %rd26, 262144;
    add.s64          %rd27, %rd1, %rd25;
    add.s64          %rd43, %rd27, 5242888;
}
```

Even without fully comprehending this type of low-level PTX code, one can abstract away from its semantics/meaning and simply compare both kernel versions line by line. By doing so, it becomes evident that the sequence of GPU instructions remains functionally equivalent. The arithmetic operations and control flow are preserved across both implementations. The only noticeable differences lie in the memory offsets used, which simply reflect small variations in where data is allocated in memory. These changes do not impact the algorithm's logic or performance, confirming that the abstraction introduced by Marionette doesn't incur performance overhead.