GPU Acceleration and EDM Developments for the ATLAS 3D Calorimeter Clustering in the Software Trigger

Nuno dos Santos Fernandes[1,2,3,*] on behalf of the ATLAS Collaboration

[1]LIP – Laboratório de Instrumentação e Física Experimental de Partículas, Lisboa, Portugal
[2]IST – Instituto Superior Técnico, Universidade de Lisboa, Portugal
[3]CERN, Geneva, Switzerland

Abstract. The ATLAS experiment will undergo a series of upgrades in association with the High-Luminosity LHC program. Given the new high-luminosity conditions and the predicted increase in event rates at the ATLAS High-Level Trigger by a factor of 10, additional computational load will be placed on the trigger farm. One possibility to accommodate this is the use of hardware accelerators, such as GPUs, for the cost and energy efficiency they offer.
Among the algorithms being assessed for GPU acceleration is Topological Clustering, the main and most computationally demanding stage of calorimeter reconstruction. A more GPU-friendly variant of the algorithm, dubbed Topo-Automaton Clustering, has been implemented, reaching the significant milestone of 100% agreement with the CPU algorithm and maximum speed-ups in excess of a factor of 10. A significant bottleneck remains in conversion between the data representation used within the GPU and the equivalent CPU data structures, which can consume up to two thirds of the total execution time of the algorithm.
The development, optimization and integration of Topo-Automaton Clustering with the ATLAS trigger will be described, including the latest benchmarks and ongoing efforts to develop a framework for general description of GPU-friendly data structures to mitigate the current bottleneck.

1 Introduction

The Large Hadron Collider (LHC) [1] performs high-energy particle collisions, with a bunch crossing rate of 40 MHz (one every 25 ns) and an average number of collisions per bunch crossing of $\langle\mu\rangle \simeq 63$ in proton-proton collisions. The LHC is scheduled to undergo the High-Luminosity Upgrade [2] (HL-LHC) by the end of 2029, which represents a significant increase in this number, up to an average of $\langle\mu\rangle \simeq 200$. This leads to events that are typically more complex, with higher detector occupancy, and thus more computationally demanding to reconstruct.

ATLAS [3] is one of the two general-purpose experiments at the LHC, aiming to detect a wide variety of physics processes. It is composed of multiple sub-detectors, corresponding to an Inner Detector that provides tracking information for charged particles, a set of Calorimeters (with an inner Liquid Argon Calorimeter and an an outer Tile Calorimeter) to measure the energy of particles that interact electromagnetically or hadronically, and Muon Spectrometers to detect outgoing muons. Its trigger system plays a key role in selecting the events that will be

[*]e-mail: nuno.dos.santos.fernandes@cern.ch

recorded, given practical limitations in both storage speed and space. The ATLAS trigger works in two stages: the Level 1 Trigger, which corresponds to a hardware-based coarse filtering applied using custom electronics and FPGAs that reduces the 40 MHz event rate to 100 kHz, and the High-Level Trigger, which relies on offline-like algorithms implemented fully in software, running on a farm of commodity CPUs, to provide the final output at a rate of the order of 3 kHz.

Given the challenging conditions of the High-Luminosity LHC, ATLAS will undergo its Phase II upgrade, which encompasses a broad-ranging suite of upgrades to the detectors and also to the trigger. An increase of the event rates at the second stage of the trigger by a factor of 10 is expected. The increase in the number of events to be processed, together with the additional complexity of processing each event given the higher detector occupancy resulting from the high luminosity conditions, means that a greater computational load is placed on the computer farm on which the software trigger will run.

Two obvious courses of action can be undertaken to accommodate this: further optimising existing code whenever possible, so as to better leverage the existing resources, and augmenting the farm with additional CPUs. However, an interesting additional possibility is employing hardware accelerators, which may offer better cost and energy efficiency for certain kinds of operations. Graphical Processing Units, or GPUs, in particular, are designed for massive parallelism, which can be useful for a wide variety of problems, and the feasibility of their use within the ATLAS trigger is under active investigation.

2 The Topo-Automaton Clustering Algorithm

2.1 Calorimeter Reconstruction

As particles pass through the calorimeters of the ATLAS experiment, they generate secondary particle showers through interactions with the medium. The location and shape in which these showers develop can provide relevant insights on the nature and energy of their originating particles, so they are the main focus of calorimeter event reconstruction.

The calorimeters are split into 187652 finite regions of space called *cells*, each of which measures the energy deposition per event. The algorithms employed to recover the showers essentially correspond to clustering algorithms, grouping the cells in which each shower deposited its energy into *clusters*. The cells are organized in up to 28 sampling layers, corresponding to different regions of the detector, and, in general, form an irregular grid.

The algorithms also take into account the noise that affects the energy measurement at each cell, both from the electronic read-out systems and the pile-up. This noise is generally estimated from calibration-derived constants depending on the cell's *gain*, which can vary from event to event in order to optimise the resolution versus the dynamic range of operation. For the particular case of the Tile calorimeter, a more sophisticated *double-Gaussian model* can be applied, which requires calculating the inverse error function of a sum of error functions of the energy of the cell in an event and applying some strict cut-offs based on the results.

In general, an event contains several hundred to a few thousand clusters, depending on the underlying physical process and the event occupancy, and, while the majority of the clusters typically have several tens of cells, some may be significantly larger, up to several thousand.

2.2 Topological Clustering

The main algorithm used for calorimeter reconstruction within ATLAS is Topological Clustering [4], which relies on the signal-to-noise ratio of the energy measurements, acting essentially as a proxy for the relevance of the contribution of each cell to the reconstruction of the underlying physics. The general structure of the Topological Clustering algorithm can be seen in figure 1.

The first stage, **cluster growing**, builds large groups of cells based on their signal-to-noise ratio: *seed cells* will originate clusters, *growing cells* allow clusters to expand to neighbouring cells, *terminal cells* can be added to a cluster, but do not allow it to expand, and *invalid cells* cannot be part of any cluster. A list of seed cells, sorted by their signal-to-noise ratio, is taken as the starting point of the iterative part of the algorithm, which will evaluate the neighbours of the *list of currently evaluated cells*, adding terminal and growing cells to the clusters, and also adding the growing cells to a *list of cells to be evaluated at the next step*. Once all the currently evaluated cells have had their neighbours checked, the next step of the iteration begins, now checking the neighbours of the cells in the second list. This is repeated until the list of cells to evaluate is empty. If, at any point, a growing or seed cell could belong to more than one cluster, all the clusters in question are merged, resulting in a single cluster containing all of their cells.

These clusters are then split around local maxima of the energy to better distinguish showers generated by multiple particles travelling in a similar direction. This corresponds to the **cluster splitting** stage. Such maxima are identified considering solely the cells within the cluster, with additional requirements in terms of the number of neighbours and their energy. To improve the physical significance of the results, maxima coming from layers with greater radiation length are considered to be *primary maxima*, with other layers giving rise to *secondary maxima*. Whenever secondary maxima overlap with other maxima in the radial direction, they are excluded. The local maxima that remain will be the origin of the final clusters, and the algorithm then proceeds similarly to cluster growing: a *list of currently evaluated cells* is built, in this case from the local maxima sorted by their energy, and the neighbours of these cells are then iteratively checked. Cells within the same post-growing cluster are added to the final clusters and to the *list of cells to be evaluated at the next step*, which will be sorted by the cell energy. Whenever a cell could belong to several final clusters, it will be *shared* between the clusters of the two most energetic neighbours. These shared cells only allow cluster expansion after all reachable non-shared cells have been considered, and in the end a weight representing their contribution to each of the clusters is calculated based on the distance from the cell to the centroid of each cluster and the cluster energies.

Once these final, post-splitting clusters are created, one can proceed with the **cluster moments calculation**. The cluster moments reflect relevant properties of the geometry and shape of the clusters and of the energy signal associated with them, involving the calculation of functions of sums of cell properties such as their energy, gain or coordinates. Certain moments depend on the shower axis, which requires computing the eigenvalues and eigenvectors of a 3×3 matrix. The moments will be used in later stages of event reconstruction, and in particular

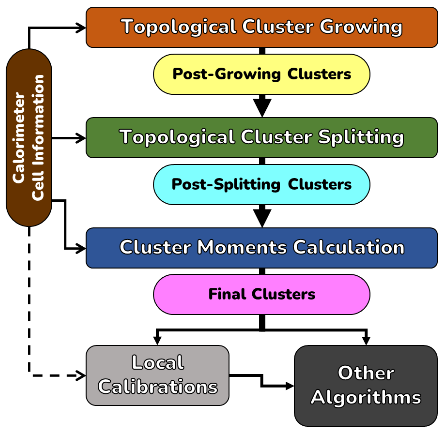

Topological Cluster Growing
Calorimeter Cell Information
Post-Growing Clusters
Topological Cluster Splitting
Post-Splitting Clusters
Cluster Moments Calculation
Final Clusters
Local Calibrations
Other Algorithms

Figure 1: Stages of Topological Clustering.

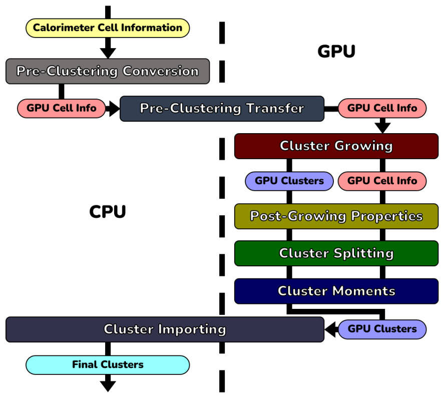


Figure 2: Stages of Topo-Automaton Clustering.

in applying some **local calibrations** to the clusters. The final clusters are then passed on to other trigger algorithms, such as jet, missing transverse energy, electron/photon and tau reconstruction.

2.3 Topo-Automaton Clustering

The Topological Clustering algorithm, as previously described, inherently relies on serial evaluation to provide the correct ordering, once the lists of cells are sorted. Efficiently resizing the lists of cells, as well as the (smaller) lists of cells that are used to represent the clusters, is challenging in a multi-threaded context. For these reasons, a different algorithmic approach will be required: a *tag* is associated with every cell, identifying the cluster(s) to which it has been assigned, and a set of rules and conditions specifies how the tags will change depending on the tags of the neighbouring cells. Since this is formally equivalent to a cellular automaton, we call this the **Topo-Automaton Clustering** algorithm.

The definition of the tags can be fine-tuned to optimise the algorithm. In particular, given that floating point numbers following the IEEE-754 standard [5] can be put in a "total ordering" where the bit patterns are ordered in the same way as the underlying floats, the sorting steps can be altogether skipped. Figure 3 shows the general structure of the tags used for cluster growing and cluster splitting. Furthermore, by defining the rules for tag changes in an appropriate way, for instance, using a *reverse propagation counter* to give priority to cells that are closer to the origin of the clusters, one can consider each pair of neighbouring cells independently. This allows for a natural parallelisation of the algorithm, as long as any updates are thread-safe (e. g. atomic). The initial stages of both parts of Topo-Automaton Clustering consist of building the lists of pairs of neighbouring cells that could be relevant for cluster expansion, and the handling of these pairs will be distributed among the GPU threads during the iterative part, which may only change the tag assigned to the first cell of each pair, depending on the rules for tag propagation.

Naturally, some other improvements to this basic idea can be made to achieve greater optimization. A cell to cluster index table allows for a more natural expression of cluster merges during cluster growing and also simplifies the logic for handling shared cells in cluster splitting. If tag propagation uses a separate output array for the tags to be assigned to the cells at the next step, the logic can be simplified, in particular for cluster splitting. The latter optimization also makes each step of the cluster expansion deterministic regardless of the order of evaluation.

A GPU-accelerated implementation of Topo-Automaton Clustering that includes both cluster growing and cluster splitting has been developed using the CUDA programming language. In addition, the calculation of cluster moments has also been implemented on the GPU, leveraging the inherent parallelism to perform the weighted sums of cell properties in a more efficient way. This is also achieved by distributing the work needed to calculate the several moments across a warp rather than a single thread, for greater throughput with minimal compromise to memory locality.

In general, the current implementation follows the structure outlined in figure 2. First, the cell information (corresponding to energies, gains and other associated properties) must be converted to a GPU-friendly format and transferred to the GPU. Then, the stages of the algorithm itself are executed (with the calculation of basic kinematic cluster properties after cluster

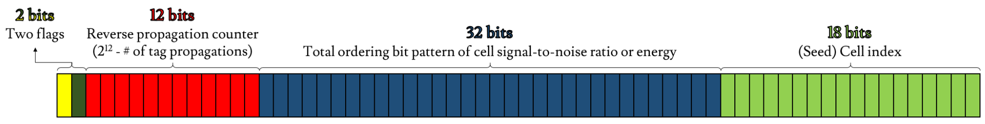

2 bits
Two flags
12 bits
Reverse propagation counter
(2^{12} - # of tag propagations)
32 bits
Total ordering bit pattern of cell signal-to-noise ratio or energy
18 bits
(Seed) Cell index

Figure 3: General structure of the tags used in both stages of Topo-Automaton Clustering.

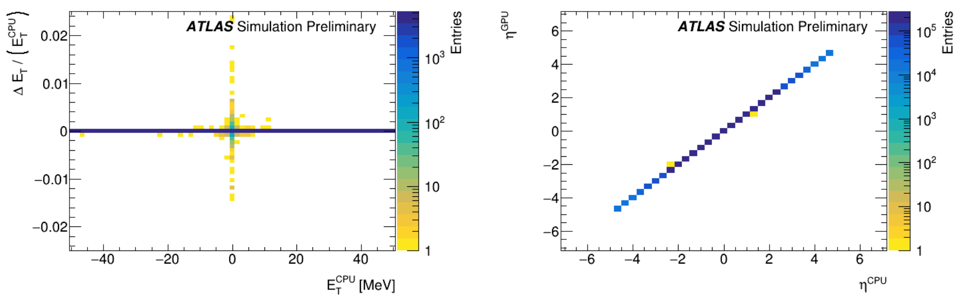
growing being identified as a separate step since its implementation is distinct from cluster growing). In the end, the GPU clusters that result from the algorithm must be transferred back to the CPU and converted to data structures that can be used by the rest of event reconstruction.

3 Validation of the Implementation

To assess the extent of the agreement between the clusters reconstructed by Topo-Automaton Clustering and those coming from the reference CPU implementation, they may be matched based on the cells they have in common. However, certain fundamental differences between the implementations must be taken into account, namely the handling of certain edge cases or indeterminacies. The most apparent is the sorting of cells that have the same signal-to-noise ratio or energy, which is left entirely unspecified by the CPU, while the GPU will use the cell index as tie-breaker. Another possible source of differences is the double-Gaussian noise model, which runs into the limitations of floating point accuracy, potentially leading to cells being classified differently between the CPU and the GPU during cluster growing. If the CPU implementation is modified to behave in the same way as the GPU when it comes to these edge cases and the double-Gaussian model is not used, perfect agreement is found between all the CPU and GPU clusters, down to the very last cell. Nevertheless, even if comparing with the unmodified CPU reference implementation using the double-Gaussian model, on average, no more than 1% of the final clusters show any difference.

One can also study the difference in the calculated cluster properties for the case where the clusters are exactly the same, which essentially reflects the impact that floating point errors may have on such calculations. Figure 4 illustrates both the relative error for the cluster transverse energy, E_T, as a function of the value reconstructed by the reference CPU implementation, and the direct comparison between the calculated cluster pseudo-rapidity, η, in both implementations. These comparisons were produced with 3000 Monte-Carlo simulated $t\bar{t}$ events, which correspond to one of the densest kinds of events in terms of the number and size of the clusters, for which any disagreements would thus be more pronounced than in the average event evaluated by the trigger. For the calculation of the cluster moments, some more significant differences can be found, albeit in less than 1% of the clusters, due to compounded floating point errors, given the several dependencies on previously calculated values.

For further validation purposes, Topo-Automaton Clustering was used in a trigger reprocessing of a standard trigger testing run with real data, taking the place of Topological Clustering in the default trigger configuration (which also means that some differences



(a) Relative error in cluster transverse energy (E_T)

(b) Comparison of cluster pseudo-rapidity (η)

Figure 4: Comparison of calculated cluster properties, based on 3000 Monte-Carlo simulated $t\bar{t}$ events with an average number of collisions per bunch crossing of $\langle\mu\rangle = 80$. Source: [6]

compared to the CPU would be *a priori* expected, since the double-Gaussian model is used). This was the first trigger reprocessing involving GPUs within ATLAS. No significant differences in any triggers depending on the clusters were found except in jet triggers with very low counts, and in all cases the overall trigger selections were not affected. Taking all of this into account, it is thus possible to conclude that Topo-Automaton Clustering accurately reproduces the behaviour of Topological Clustering for triggering purposes.

4 Topo-Automaton Clustering Benchmarks

The following benchmarks were obtained on a system with a Tesla P100 GPU and a Xeon E5-2695 v4 CPU, based on two sets of Monte-Carlo simulated events: 3000 $t\bar{t}$ events with an average pile-up of $\langle\mu\rangle = 80$ (somewhat higher than currently achieved at the LHC) and 10000 di-jet events at a pile-up of $\langle\mu\rangle = 200$ (corresponding to HL-LHC conditions).

Using a per-thread clock, time measurements were taken for the CPU and GPU implementations, for different numbers of CPU threads (corresponding to different numbers of events being processed in parallel). The speed-up factor calculated from the ratio between the CPU and GPU execution times is shown in figure 5. The difference between $t\bar{t}$ and di-jet events is explained mostly by how the CPU implementation scales with the complexity of the event, especially in terms of the average number of clusters per event (~ 1500 for $t\bar{t}$ *versus* ~ 900 for di-jet).

In order to determine potential future targets for further optimisation, one can also measure the time taken by the several steps of the GPU implementation, for a single CPU thread. This is shown in table 1. The main bottleneck lies in the post-clustering data transfers and (especially) conversions back to the CPU data structures. This is, however, somewhat non-trivial to address

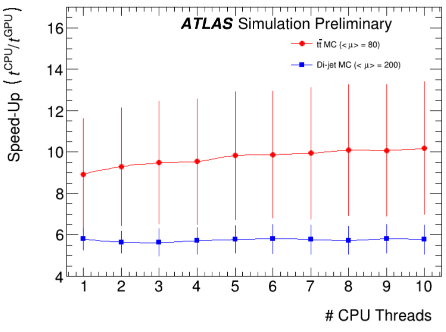


Figure 5: Speed-up from GPU acceleration, defined as the per-event ratio between the CPU and GPU processing time. The error bars show the standard deviation of the distribution. Source: [6]

Table 1: Breakdown of the GPU execution times for every step of Topo-Automaton Clustering.

Step	$t\bar{t}$ Events		Jet Events	
	Time (ms)	Fraction of Total Time	Time (ms)	Fraction of Total Time
Pre-Clustering Conversion	1.4 ± 0.2	$9\pm2\%$	1.1 ± 0.1	$13\pm1\%$
Pre-Clustering Transfer	0.27 ± 0.01	$1.7\pm0.2\%$	0.25 ± 0.02	$2.9\pm0.3\%$
Cluster Growing	0.40 ± 0.05	$2.5\pm0.4\%$	0.28 ± 0.02	$3.2\pm0.2\%$
Post-Growing Property Calculation	$(70\pm20)\times10^{-3}$	$0.5\pm0.1\%$	$(55\pm2)\times10^{-3}$	$0.7\pm0.1\%$
Cluster Splitting	1.42 ± 0.13	$9\pm1\%$	0.85 ± 0.07	$10\pm1\%$
Cluster Moments Calculation	1.4 ± 0.1	$9.1\pm0.6\%$	0.9 ± 0.1	$10.4\pm0.5\%$
Post-Clustering Transfer + Conversion	10.7 ± 1.4	$68\pm3\%$	5.1 ± 0.7	$59\pm2\%$
Total	16 ± 2	—	9 ± 1	—

given the need to interface with the rest of the CPU-based trigger algorithms. The latter impose the requirement that the final output of the algorithm is expressed in the data structures used elsewhere in the code, which, for several technical reasons, are unimplementable in an efficient way on the GPU.

5 Event Data Model Developments: Marionette

As laid out in the previous section, the main bottleneck in the current implementation is the data structure conversion step. While a more directed approach that could improve this for the particular case of the data structures used within Topo-Automaton Clustering would be possible, we opted for the development of a more general Event Data Model (EDM) framework that can handle arbitrary data structures. This framework can automatically provide transfers and conversions between CPU and GPU (or vice-versa) and accommodate a pre-existing interface, providing enough customization points to cater for the majority of possible use cases. This more general solution may be useful for other GPU-accelerated projects within and beyond the ATLAS trigger.

The Marionette (**M**emory **A**bstracted **R**epresentation with **I**nterfaces in **O**bjects **N**ecessitating **E**xtensively **T**emplated **T**ypes **EDM**) library[1] has thus been developed over the past year to provide the desired functionality. It is a header-only, C++17 library that relies on template meta-programming and a rather complex set of compile-time abstractions to build data structures with the desired properties and interface.

The basic idea is two-fold: firstly, by decoupling *what is stored* from *how to store it*, one can easily define data structures with a single source of truth for their constituents; secondly, by allowing the description of the data structures to also contain functions that will be added to the interface of the final classes, the behaviour of pre-existing data structures can be reproduced, greatly reducing the effort of porting code to use Marionette-provided data structures. From the point of view of the end user, one simply needs to provide a list of compile-time classes describing the desired properties (e. g. a single value, a vector of values, a set of values to represent a sub-object), and to specify a given *layout*, which will implement the desired storage strategy. While the library implements some useful strategies by itself, it provides the freedom for users to describe their own layouts, with behaviour tailored for the intended use case (for example, if additional bookkeeping beyond what a simple allocator can provide is needed).

Marionette provides two main classes, `Object` and `Collection`, to represent, respectively, an object-oriented class holding the desired properties (e. g. a calorimeter cluster) and a set of such classes (e. g. the list of all clusters created during an event). These classes are templated based on three parameters: `Layout`, which represents the data storage strategy (which, for the case of `Object`, corresponds to either being a proxy into a collection, or holding the individual values, while `Collection` offers full customisability); `Properties`, the compile-time list of properties that the data structures should contain; and `MetaInformation`, which is used internally by the library to express certain constructs, such as views into a subset of the properties of the collection, and thus for most cases should not be directly specified by the end user as the default parameter has the desired behaviour. A `Collection` provides the same interface as `std::vector` when it comes to dealing with the individual objects, even if the underlying storage may hold each property as a separate array (in other words, an *array-of-structures* interface is provided even though the underlying layout is *structure-of-arrays*). Sensible default behaviours for transferring data between different layouts are provided (thus allowing e. g. CPU to GPU transfers). In addition, it is possible for implementers and end users to override these behaviours if desired, with all of this also being handled fully at compilation time.

In short, Marionette offers a wealth of functionalities and is designed to offer as much customisability as possible to ensure the final classes are performant and offer an intuitive

[1] https://gitlab.cern.ch/dossantn/edm-overhaul

interface. In its current state, several of the examples included in its repository show that using Marionette data structures can result in exactly the same assembly as using the equivalent hand-written data structures, in particular within a CUDA kernel, thereby proving that the provided abstractions have no runtime impact. Further developments are ongoing to improve compilation times, which, by the nature of the library, can become somewhat significant for sufficiently complex data structures, and to further extend some of its functionalities, such as supporting objects holding arbitrarily nested vectors or partial views into collections (subspans).

6 Conclusions and Future Work

In order to provide a possible solution to alleviate the greater computation demands placed upon the trigger of the ATLAS experiment by the High-Luminosity LHC and the associated ATLAS Phase II upgrade, the Topo-Automaton Clustering has been successfully developed and implemented, using CUDA for GPU acceleration. The current implementation encompasses cluster growing, cluster splitting and the calculation of cluster moments, with configurability on a par with the reference implementation of Topological Clustering on the CPU, allowing it to act essentially as a drop-in replacement of the pre-existing code. Under appropriate conditions, perfect agreement in cell assignment can be found between Topo-Automaton and Topological Clustering, and in all other cases the reasons for any remaining differences are well understood and have been shown to have no impact on the trigger operation.

In terms of speed-up, one could quote a value of 9–10 for $t\bar{t}$ and 6 for di-jet events. Given that a significant portion ($60 \sim 70\%$) of the GPU event processing time is spent in data conversions (and, to a lesser extent, data transfers), it is clear that GPU acceleration for the algorithm itself has the potential for even greater speed-up.

Efforts are currently underway to improve the bottleneck represented by the data structure conversions. This is a complex issue, in particular due to the well-established nature of the CPU data structures that the rest of the code relies on, but Marionette is being developed as an attempt to provide a general solution to this class of problems that may be used in further GPU acceleration efforts. The validation tests suggest that the provided abstractions will come at no runtime performance penalty compared to the equivalent hand-written code. The natural next steps, once the data structures of the current implementation have been ported to Marionette, will be to benchmark Topo-Automaton Clustering once more to ensure this is indeed the case, and then to leverage the functionalities of Marionette to provide an optimised implementation of the data structure conversions.

References

[1] L. Evans, P. Bryant, LHC Machine, Journal of Instrumentation **3**, S08001 (2008). 10.1088/1748-0221/3/08/s08001

[2] G. Apollinari, I. Béjar Alonso, O. Brüning, P. Fessia, M. Lamont, L. Rossi, L. Tavian, High-Luminosity Large Hadron Collider (HL-LHC): Technical Design Report V. 0.1 (2017). 10.23731/CYRM-2017-004

[3] The ATLAS Collaboration, The ATLAS Experiment at the CERN Large Hadron Collider, JINST **3**, S08003. 437 p (2008). 10.1088/1748-0221/3/08/S08003

[4] The ATLAS Collaboration, Topological Cell Clustering in the ATLAS Calorimeters and its Performance in LHC Run 1, The European Physical Journal C **77** (2017). 10.1140/epjc/s10052-017-5004-5

[5] IEEE Standard for Floating-Point Arithmetic, IEEE STD 754-2019 (Revision of IEEE 754-2008) (2019). 10.1109/IEEESTD.2019.8766229

[6] The ATLAS Collaboration, Event Filter – Calorimeter Public Results, https://twiki.cern.ch/twiki/bin/view/AtlasPublic/EFCaloPublicResults