Marionette: Data Structure Description and Management for Heterogeneous Computing

Nuno dos Santos Fernandes
Instituto Superior Técnico, U. Lisboa, Portugal
LIP, Lisbon, Portugal
The ATLAS Collaboration, CERN, Geneva, Switzerland
nuno.dos.santos.fernandes@cern.ch

Pedro Tomás
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
pedro.z.tomás@tecnico.ulisboa.pt

Nuno Roma
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa, Portugal
Nuno.Roma@inesc-id.pt

Frank Winklmeier
Institute for Fundamental Science,
University of Oregon, United States of America
The ATLAS Collaboration, CERN, Geneva, Switzerland
frank.winklmeier@cern.ch

Patricia Conde Muíño
Instituto Superior Técnico, U. Lisboa, Portugal
LIP, Lisbon, Portugal
The ATLAS Collaboration, CERN, Geneva, Switzerland
patricia.conde.muino@cern.ch

***Abstract*—Adapting large, object-oriented C++ codebases for hardware acceleration might be extremely challenging, particularly when targeting heterogeneous platforms such as GPUs. Marionette is a C++17 library designed to address this by enabling flexible, efficient, and portable data structure definitions. It decouples data layout from the description of the interface, supports multiple memory management strategies, and provides efficient data transfers and conversions across devices, all of this with minimal runtime overhead due to the compile-time nature of its abstractions. By allowing interfaces to be augmented with arbitrary functions, Marionette maintains compatibility with existing code and offers a streamlined interface that supports both straightforward and advanced use cases. This paper outlines its design, usage, and performance, including a CUDA-based case study demonstrating its efficiency and flexibility.**

***Index Terms*—data structures, abstracted memory layout, hardware acceleration, GPU**

I. Introduction

Data structure usage in a context in which heterogeneous accelerators are involved is non-trivial. Adapting a pre-existing code-base that was not written with accelerators in mind can be even more challenging, especially when object-oriented design patterns were adopted and algorithmically relevant code was written to leverage those. Since, in most cases, accelerator-centric code must coexist with some processing being done on the host, there is the additional constraint of being able to interoperate between data structures resident on either. Given that accelerators may favour different memory access patterns, the goal is supporting not only multiple ways of managing memory, but especially different ways in which to lay the data structures out in memory, while maintaining a consistent and convenient interface to the objects contained within them.

Specifically within the context of C++ and its accelerator-related extensions, the most immediate solution would be to write the desired data structures and all functionality to transfer and/or convert between them and the pre-existing host structures by hand. However, this is an onerous, bug-prone process that introduces multiple sources of truth (host data structures, accelerator data structures, host to accelerator conversions, accelerator to host conversions), which can be especially inconvenient if the original data structures themselves may be subject to changes.

Given the grammar of C++ and the (current) limitations in both reflection and programmatic code generation, there is no straightforward way to automatise the writing of all relevant classes and functions from a unified description of the intended design of the data structures. In particular, the ability to ensure that the interface of the automatically generated data structures matches the expectations of pre-existing code, such as in terms of accessors and mutators, is highly non-trivial to provide. In any case, to ensure maintainability, the solution must allow programmers of varying levels of skill and experience to engage with the mechanisms through which data structures and their interface are defined.

As a result of all of these considerations, the Marionette (short for **M**emory **A**bstracted **R**epresentation with **I**nterfaces in **O**bjects **N**ecessitating **E**xtensively **T**emplated **T**ypes **EDM**[1]) library was developed, in an attempt to provide a convenient user experience while employing advanced compile-time programming techniques to ensure the desired behaviour can be achieved at minimal runtime cost, offering a wide variety of customisation points that more advanced users can leverage without compromising the ease of use of simpler functionality.

The code may be found in the following repository: https://gitlab.cern.ch/dossantn/marionette

II. Context and Related Work

Abstracting the description of data structures lies at a challenging intersection between requiring minimal changes to pre-existing code, supporting the broadest possible variety of

This work is financed by Portuguese national funds through the FCT – Fundação para a Ciência e a Tecnologia, I.P., within the scope of the projects PRT/BD/153342/2021 (DOI: 10.54499/PRT/BD/153342/2021), 2024.06584.CERN (DOI: 10.54499/2024.06584.CERN) and UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020).

[1]EDM stands for "Event Data Model" as the project arose from an attempt to accelerate event processing at the ATLAS Experiment [1] with GPUs.

environments, compilers and build systems, offering the best performance possible and minimising syntactical overhead.

Trivially, the solution of writing all the code related to data structures by hand is possible, even if cumbersome. Adding a code generation step, or transpiling to C++ from a different language or dialect, can offer the most intuitive syntax, but it may increase the complexity of the build process and raises concerns in regards to maintainability and support for present and future forms of hardware accelerators. On the other hand, relying on compiler extensions to (partially) automate the generation of the desired code, such as in [2], implies a dependency on a particular subset of compilers, which may not offer support for all accelerators of interest.

Solutions that work solely at the level of C++ source code are only dependent on compiler support for the required C++ standard. Reflection and code generation functionalities are the most general way of approaching this problem. However, the current proposal to add reflection to the C++26 standard [3] only offers limited possibilities for code generation (`define_aggregate`), and compilers and tool-chains for hardware accelerators may not always fully support the latest standards.

The basic grammar of C++ itself offers some possibilities for modifying the contents and interface of a type, through macros, templates and inheritance. Several projects take this approach, such as [4], [5], [6], [7] or [8], with varying support for accelerators and varying trade-offs between performance, generality and user experience. Marionette employs similar strategies, while offering increased flexibility of the defined interfaces and a wide variety of customisation points without loss of performance.

III. Motivating Example

To guide the discussion of the design of Marionette and its functionality, one can consider a motivating example, corresponding to a simplified version of a practical use case.

Drawing from the context of its original design, suppose there is a 2D grid of sensors (of different types) that measure the energy of incoming particles, with a set of calibrated parameters expressing the conversion from raw sensor counts to energy and an estimate of the noise associated with the measurement. The particles are reconstructed based on the energy measurements, including a list of sensors that contributed to the reconstruction and properties that are tracked separately for each type of sensor. Both sensor and particle information will be expressed in a mostly object-oriented way, as shown in listings 1 and 2.

Suppose, then, that part of the code interacting with these structures will be adapted to use hardware acceleration, but, due to more complicated dependencies and/or personpower limitations, a non-negligible portion of the algorithms will have to be ported gradually, if at all. Thus, code referencing the `Sensor` and `Particle` class must remain valid on both host and accelerator. In addition, the ability to experiment with different data layouts may be useful for development efforts and optimization.

Most of the discussion will follow from this example, including the benchmarks presented in section VIII.

```
1  class Sensor {
2    SensorType m_type;
3    uint64_t   m_counts;
4    float      m_energy;
5    class Calibration {
6      bool m_noisy;
7      float m_parameter_A,
8            m_parameter_B;
9      float m_noise_A, m_noise_B;
10     //Depend on the sensor type
11   public:
12     /* getters and setters */
13   } m_calibration_data;
14 public:
15   /* getters and setters */
16   void calibrate_energy();
17   float get_noise() const;
18 };
```

Listing 1. Example `Sensor`.

```
class Particle {
  float    m_energy;
  float    m_x, m_y;
  uint64_t m_origin;
  std::vector<uint64_t>
           m_sensors;
  float    m_x_variance,
           m_y_variance;
  float    m_significance
           [SensorType::Num];
  float    m_E_contribution
           [SensorType::Num];
  uint8_t  m_noisy_count
           [SensorType::Num];
  //Separate for each sensor type
public:
  /* getters and setters */
};
```

Listing 2. Example `Particle`.

IV. Main Requirements

As suggested in the previous sections, Marionette was developed to offer a general solution for abstracting the layout of data structures in an accelerator-compatible way. The range of supported heterogeneous computing platforms should be as broad as possible. Given current trends, GPUs would be the most natural use case, but the design should not exclude support for FPGAs and other forms of co-processor. The set of requirements for the design of Marionette may be summarized thus:

- Allow the description of data structures with unified interface across multiple heterogeneous computing platforms:
  - Handle data structure generation at compile time, avoiding unnecessary run-time overhead and polymorphism;
  - Provide an intuitive object-oriented interface on top of the underlying data structures (array-of-structures-like);
  - Support user-provided data storage strategies;
  - Provide basic data storage strategies that can be readily adapted to new heterogeneous computing devices by specifying a minimal set of memory-related operations;
- Implement efficient data transfers between data structures that are laid out according to different storage strategies:
  - Allow the end user to specify additional data transfers, including from types outside the library;
- Allow the addition of arbitrary functions to the final interface of the data structures;
- Support several kinds of properties within the data structures, with different semantics, which should be arbitrarily nestable;
- Impose no further requirements than the ability to compile C++17 for the target platform/device, but support later features when available (e. g. the spaceship operator <=>);
- Achieve all of this with minimal run-time impact compared to the equivalent handwritten solution.

V. The Basic Design of Marionette

The Marionette library is designed around two main classes: `Object` and `Collection`. These represent, respectively, a single entity with the specified contents and interface and a list (with an `std::vector`-like interface) of such structures. The three template parameters of these classes provide the core of the flexibility offered by the library.

The first template parameter corresponds to the *layout*, which determines the way in which the data will be stored in memory. For objects, the layout is used internally to support both owning objects and proxies into collections (which provide the object-oriented interface of the data structures). For collections, it is specified by the user according to their needs. For example, allocating memory in blocks of a given size, as opposed to a pure structure-of-arrays where each array is contiguous, or on a hardware accelerator as opposed to the

host. Sub-section VII-B provides more details on the layouts offered by Marionette and on how to define a new layout.

The second template parameter corresponds to a compile-time list of *properties* to be held by the collection. This allows generating the data structures according to the layout, with the desired interface. A property, in this sense, is a generalisation of a member variable to be associated with each object. Defining properties will correspond to the majority of end-user interactions with the Marionette framework, as explained in the following section.

The third template parameter holds *meta-information* that provides a more convenient interface to certain kinds of properties. As this is managed internally by the library, the end user should simply rely on the default value of the template parameter when declaring variables (i. e. not provide it explicitly).

VI. Implementing Data Structures in Marionette

Properties are represented by a *property description class*, which expresses compile-time information about the desired semantics of the property and any other relevant information (e. g. the type of a variable). The property description class also specifies functions that will be added (via inheritance) to a collection or object that contains the property in question. These would be implemented in `CollectionFunctions` and `ObjectFunctions` member types, which are templated on the type of the overall collection or object and the underlying layout. Within those functions, casting the `this` pointer to a pointer to the final class allows accessing the full interface of the `Collection` or `Object`, including other functions associated with the properties. Of note are `get_collection`, `get_object` and `get`, which access the storage corresponding to a property description class.

The most common kind of property would be a single native type that is associated with every object in a collection: a ***per-item property***. For this, one must specify the type to be stored, as well as any functions (e. g. accessors and mutators) that provide the desired interface. To give a concrete example, listing 3 implements the `energy` property of a `Sensor`.

Given that this is a very common pattern, a more convenient `MARIONETTE_DECLARE_PER_ITEM_PROPERTY` macro is provided, taking as arguments the uncapitalized and capitalized forms of the property name, with the latter being used as the name of the property description class, and the type to be stored. Several ways are offered to customise the generation of the accessor and mutator names. Similar property declaration macros exist for most other kinds of properties, all of the form `MARIONETTE_DECLARE_*` and taking the names as the first two arguments.

A ***no-property*** **property** allows extending the interface of collections or objects without any associated storage. The property description class must inherit from `NoProperty` and define the `ObjectFunctions` and `CollectionFunctions` classes. This would be used to implement the `calibrate_energy` and `get_noise` member functions of a `Sensor`.

To reproduce the semantics given in the example for `calibration_data`, one would employ a ***sub-group property***. The property description class inherits from `SubGroup` and defines a `Properties` member type corresponding to a compile-time list of other property descriptions to be

```
struct Energy : Marionette::InterfaceDescription::PerItemProperty {
 using Type = float;
 template <class F, class L> struct ObjectFunctions {
  const float & energy() const {
   return static_cast<const F*>(this)->template get<Energy>(); }
  float & energy() {
   return static_cast<F*>(this)->template get<Energy>(); }
  void set_energy(const float & e) {
   static_cast<F*>(this)->template get<Energy>() = e; }
 };
 template <class F, class L> struct CollectionFunctions {
  decltype(auto) energy() const {
   return static_cast<const F*>(this)->template get_collection<Energy>(); }
  decltype(auto) energy() {
   return static_cast<F*>(this)->template get_collection<Energy>(); }
  template <class T> void set_energy(T && t) {
   static_cast<F*>(this)->energy() = std::forward<T>(t); }
  decltype(auto) energy(const typename Layout::size_type i) const {
   return static_cast<const F*>(this)->template get_object<Energy>(i); }
  decltype(auto) energy(const typename Layout::size_type i) {
   return static_cast<F*>(this)->template get_object<Energy>(i); }
  void set_energy(const typename Layout::size_type i, const float & e) {
   static_cast<F*>(this)->energy(i) = e; }
}; };
```

Listing 3. Implementing the `energy` of a `Sensor` explicitly.

```
using Marionette::InterfaceDescription::PropertyList;
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(type, Type, SensorType);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(counts, Counts, uint64_t);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(energy, Energy, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(noisy, Noisy, bool);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(parameter_A, ParameterA, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(parameter_B, ParameterB, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(noise_A, NoiseA, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(noise_B, NoiseB, float);
MARIONETTE_DECLARE_SUBGROUP_PROPERTY(calibration_data, CalibrationData,
                        Noisy, ParameterA, ParameterB, NoiseA, NoiseB);
struct SensorFuncs : Marionette::InterfaceDescription::NoProperty;
//Defines appropriate calibrate_energy and get_noise object functions
using Sensor↩
        = PropertyList<Type, Counts, Energy, CalibrationData, SensorFuncs>;
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(x, X, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(y, Y, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(origin, Origin, uint64_t);
MARIONETTE_DECLARE_SIMPLE_JAGGED_VECTOR(sensors, Sensors, int, uint64_t);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(x_variance, XVariance, float);
MARIONETTE_DECLARE_PER_ITEM_PROPERTY(y_variance, YVariance, float);
MARIONETTE_DECLARE_SIMPLE_ARRAY_PROPERTY(significance, Significance,
                                         SensorType::Num, float);
MARIONETTE_DECLARE_SIMPLE_ARRAY_PROPERTY(E_contribution, EContribution,
                                         SensorType::Num, float);
MARIONETTE_DECLARE_SIMPLE_ARRAY_PROPERTY(noisy_count, NoisyCount,
                                         SensorType::Num, uint8_t);
using Particle = PropertyList<Energy, X, Y, Origin, Sensors,
                              XVariance, YVariance, Significance,
                              EContribution, NoisyCount>;
```

Listing 4. Implementing `Sensor` and `Particle` in Marionette.

contained within the sub-object. These will be treated as separate properties in the same way as if they were declared at the top level (e. g. being stored as separate arrays in a structure-of-arrays layout). The property declaration macro will take the list of other properties starting from its third argument.

The members of the example `Particle` that are tracked by sensor type could benefit from being stored in separate arrays for each type, while still providing the interface of an array within each object. This would be an ***array property***, which in essence has both a "vector of arrays" and an "array of vectors" interface. The property description class will contain the `Properties` to be held in the array and the compile-time *extent* of the array. Besides the expected property declaration macro, which takes the extent and then the list of properties starting from its fourth argument, a `*_SIMPLE_*` version will take just the extent and a type to cater to the common case where a single per-item property is held.

A ***jagged vector property*** allows associating a dynamic number of values to each object. The set of all the values for all the objects in a collection is accessible as if it were a single, continuous vector. To represent this, the prefix sum of the sizes of each individual vector is stored in a *global property*, which is not added to the interface of individual objects. In addition, the

concept of a *size tag* must be introduced so that differently sized arrays may coexist within a collection, as the set of all values for all the objects has a size that is independent from the overall number of objects. This handled in a way that is transparent to the end user. The property description class specifies the type to be used to represent the prefix sum of the sizes of the jagged vectors (which may be smaller than the `size_type` of the collection) and the list of properties to be held. Similarly to array properties, a `*_SIMPLE_*` version of the macro handles the case where there is a single per-item property within the vectors, as in the `sensors` of the `Particle`.

A possible implementation of the `Sensor` and `Particle` classes within Marionette is shown in listing 4.

VII. Advanced Usage

A. Support for Heterogeneous Computing

Marionette employs two main abstractions to support hardware accelerators: *execution contexts*, which abstract where the code is being compiled for (e. g. "CPU", "GPU with CUDA", "GPU with HIP" or "GPU with OpenCL"), and *memory contexts*, which encapsulate a way of managing memory. While support for new execution contexts may require some changes to the internals, users are free to define new memory contexts.

Each memory context has an associated `ContextInfo` type to hold runtime information related to each individual allocation. Memory contexts specify how to `allocate`, `deallocate` and `memset` memory, according to the provided context information. By design, every collection will have an associated memory context, determined by the collection's layout, and store the corresponding context information. An `update_memory_context_info` member function of `Collection` is offered to allow modifications of the context information after a collection has been constructed. This can even mean allocating memory using the new information, copying from the old memory and deallocating it.

Transferring memory between different contexts is made possible through `memcopy_with_context` and several variants of it that support partially overlapping arrays for easier insertion and deletion. These are defined as free functions, in order to support several combinations of contexts. Depending on the memory contexts in question, additional parameters may be provided to control some aspects of the data transfer.

In order to ensure all the relevant functions will be callable from all execution contexts, a set of `MARIONETTE_PORTABILITY_ANNOTATIONS` is defined and used to decorate class and function declarations throughout the internals of Marionette. These should expand to the appropriate annotations (e. g. `__host__ __device__`), with the user being able to extend or override their definition. User-defined functions and classes may employ them as well, for maximum portability.

B. Defining Layouts

The most powerful feature of Marionette is the ability to define different ways of storing the data by writing a *layout class*. The actual storage is described in the `layout_holder` member type, which is templated on three parameters: the compile-time list of properties to be stored and two multiplicative factors to the extent of the properties and size tags, which are needed to fully support properties being nested within array properties.

The layout class must specify the `size_type` and `difference_type` to be used for indexing within the collection, the memory context associated with the collection and `interface_properties` to specify what can be done with its contents (which may vary with the execution context). Other optional member functions may also be defined in order to enable optimizations and cater to specific use cases.

Implementing the `layout_holder` requires a certain amount of technical complexity since any combination of user properties must be properly supported. The required interface corresponds to a set of simple size-changing operations (`resize`, `reserve`, `clear`, `shrink_to_fit`, `insert` and `erase`), which will act on either individual "arrays" or the entirety of the properties under a given size tag, as well as access to these same "arrays" and size tags. The "arrays" need not be contiguous, only a mapping from an index to a variable of the required type.

Marionette provides out-of-the-box support for use cases where a vector-like container should be associated with each per-item property (the `VectorLikePerProperty` layout) and where a single block of memory, corresponding to a maximum size that may be specified for each property, is allocated using an allocator-like class (the `DynamicStruct` layout). Some template parameters provide additional flexibility. Accelerators are immediately supported by using appropriate containers or allocators, such as the provided `ContextAwareAllocator` and the `ContextAwareVector` classes, which rely on the functions defined in the memory context. This means that supporting new accelerators simply requires having an appropriate memory context to represent allocations on the accelerator.

Transferring data between layouts is exposed both through normal copy/move assignment and `copy` and `move` functions, which may receive the same additional arguments as a `memcopy_with_context` between the contexts of the two collections. A comprehensive set of defaults that copy the arrays corresponding to each property one by one are provided, but users may provide more optimal implementations or specify transfers from pre-existing data structures defined outside of Marionette through appropriate specialisations of the `TransferSpecification` class. This is templated on a `TransferPriority` enumerate that allows gracefully falling back to more general implementations.

VIII. Case Studies

The first prototypes obtained by applying Marionette within its original use case of accelerating the processing of events at the ATLAS Experiment with GPUs show that the generated PTX code matches the handwritten solution. The same happens for some of the tests included in the Marionette repository, such as `GPU_{nth}_test.cu`. In other words, the abstractions offered by Marionette have no runtime impact in these cases.

A more detailed study can nonetheless be carried out based on the example described in section III, whose source may be found in the `realistic_example.cu` test in the repository. This represents a realistic scenario where the data structures may be used: raw counts are used to calculate the energy of each sensor, particles are detected based on the 5×5 neighbourhood around a sufficiently energetic sensor,

and the remaining properties are calculated from the sensors that contributed in that neighbourhood. The benchmarks focus on demonstrating that the usage of Marionette has no impact compared to the equivalent hand-written solution for using a structure-of-arrays, both on the host and using CUDA for GPU acceleration. They were obtained on a system with an Intel Xeon Gold 5220 and an NVidia RTX A6000. The time measurements come from the average of the ten fastest times out of 50 executions of 10 different events.

Figure 1 shows the time taken to fill the data structures with the raw sensor information, transfer it to the GPU (if applicable) and calculate the sensor energy, as a function of the number of sensors in the grid. The overheads associated with GPU acceleration outweigh any gains for a grid smaller than 100×100, after which the difference between CPU and GPU execution times is fixed, suggesting that the data transfers and conversions are dominant. There also seems to be no difference between array-of-structures and structure-of-arrays on the CPU, likely due to the fact that the majority of the information in the `Sensor` class is needed for the computation. In any case, for both CPU and GPU operation, Marionette and the equivalent handwritten solution display exactly the same performance.

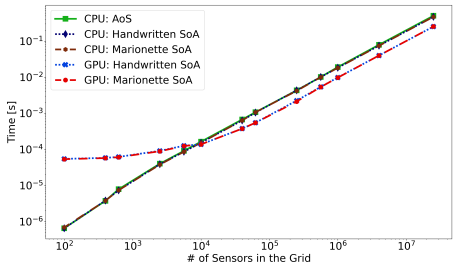

Figure 1. Execution time for sensor-related computations in the realistic test.

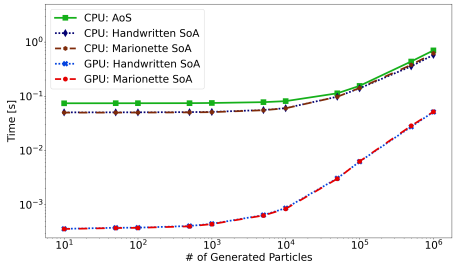

Figure 2. Execution time for particle-related computations in the realistic test.

On the other hand, figure 2 shows the time taken to reconstruct the particles, transfer back to the CPU (if applicable) and fill back the original array-of-structures, as a function of the number of particles that were generated, for a grid of 5000×5000 sensors. In this case, there is a clear speed-up from GPU acceleration, though the overheads of data transfers and conversions become increasingly relevant as the number of particles goes above 10^4. This is particularly evident in the CPU case, for which the performance advantage of a structure-of-arrays layout becomes less significant. Once again, there are no differences in performance between Marionette and the equivalent handwritten solution, both on the CPU or the GPU, confirming the zero-cost nature of the library's abstractions.

Data structures with a much more complex interface can be implemented within Marionette. Given the strictly compile-time nature of the mechanisms employed to generate the data structures, the expectation is that, in every case, the resulting code will be optimised to be equivalent to the (much more cumbersome) handwritten solution. In general, the additional possibility of sidestepping unnecessary conversions (in this case, the final step of filling the pre-existing data structures) can bring even more benefits, together with the possibility of fine-tuning the layout to the needs of the problem without loss of generality or ergonomy of the final code.

IX. Summary and Conclusions

The Marionette library offers a comprehensive set of abstractions over the description of data structures that allows supporting multiple memory allocation strategies while maintaining the same interface. The ability to specify arbitrary functions as part of the interface of the data structures or each individual object contained within them allows replicating the behaviour of pre-existing code, without incurring any run-time costs since everything is resolved during compilation. The many possibilities for customisation of the behaviour allow advanced users to fulfil the particular requirements of their use case, without compromising ease of use for more common cases.

Applying Marionette to some concrete use cases demonstrates the zero-cost nature of the abstractions it provides, as code using Marionette data structures will compile to the same operations as the equivalent hand-written data structures. Work is ongoing to employ Marionette within the more complex use case that motivated its design, with preliminary results suggesting that the zero-cost nature of the abstractions will remain.

Several other improvements to the functionality of the library are foreseen, including new types of properties, new layouts and support for more heterogeneous computing platforms.

References

[1] The ATLAS Collaboration, "The ATLAS Experiment at the CERN Large Hadron Collider," *JINST*, vol. 3, p. S08003. 437 p, 2008. [Online]. Available: https://cds.cern.ch/record/1129811

[2] P. K. Radtke and T. Weinzierl, "Compiler Support for Semi-manual AoS-to-SoA Conversions with Data Views," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, E. Deelman, and K. Karczewski, Eds. Cham: Springer Nature Switzerland, 2025, pp. 301–314.

[3] W. Childers, P. Dimov, D. Katz, B. Revzin, A. Sutton, F. Vali, and D. Vandevoorde, "Reflection for C++26," published 2025-05-17, retrieved 2025-06-25, programming Language C++, P2996R12. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2996r12.html

[4] H. Homann and F. Laenen, "SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes," *Computer Physics Communications*, vol. 224, pp. 325–332, 2018. [Online]. Available: https://doi.org/10.1016/j.cpc.2017.11.015

[5] M. Springer and H. Masuhara, "Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout," in *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3178433.3178439

[6] S. Jubertie, I. Masliah, and J. Falcou, "Data Layout and SIMD Abstraction Layers: Decoupling Interfaces from Implementations," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*, 2018, pp. 531–538. [Online]. Available: https://doi.org/10.1109/HPCS.2018.00089

[7] P. Incardona, A. Gupta, S. Yaskovets, and I. F. Sbalzarini, "A portable C++ library for memory and compute abstraction on multi-core CPUs and GPUs," *Concurrency and Computation: Practice and Experience*, vol. 35, no. 25, p. e7870, 2023. [Online]. Available: https://doi.org/10.1002/cpe.7870

[8] B. M. Gruber, G. Amadio, J. Blomer, A. Matthes, R. Widera, and M. Bussmann, "LLAMA: The Low-Level Abstraction for Memory Access," *Software: Practice and Experience*, vol. 53, no. 1, pp. 115–141, 2023. [Online]. Available: https://doi.org/10.1002/spe.3077