



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

Efficient Data Structure for Heterogeneous Reconstruction at the CMS Experiment at CERN

LAUREA MAGISTRALE IN HIGH PERFORMANCE COMPUTING ENGINEERING

Author: LEONARDO BELTRAME

Advisor: PROF. CRISTINA SILVANO

Co-advisor: DAVIDE GADIOLI, FELICE PANTALEO

Academic year: 2024-2025

1. Introduction

High-performance computing (HPC) workloads increasingly run across heterogeneous architectures (multi-core CPUs, GPUs, and other accelerators), requiring more attention to data representation and transfer. Supposing that the intention is to store information about 3-dimensional particle positions, a central design choice is the memory layout: *Array of Structures* (AoS) vs *Structure of Arrays* (SoA). In the AoS, each particle's coordinates are placed one after the others (fig. 1), while in the SoA, each field of all objects is stored in a contiguous array (fig. 2). While AoS offers intuitive, object-oriented access, it often limits vectorization and memory coalescing; SoA, instead, stores each field contiguously, improving cache locality, SIMD utilization, and GPU memory coalescing. This makes SoA widely preferred for performance-critical scientific applications. However, SoA typically comes with a usability cost: developers must orchestrate multiple arrays, track indices, and manage conversions, which increases complexity and error proneness, especially considering that the field of HPC is expanding, and many users without a strong C++ development background, such as physicists for particle sim-

ulations, mechanical engineers for fluid dynamics, or biologists for drug discovery, may need to use this data structure. For these reasons, the problem of designing efficient data structures that support different memory layouts and enable developers to write generic portable code across CPUs, GPUs, and FPGAs, is an active and highly relevant research area in high-performance computing.

Different approaches, involving template metaprogramming and code generation, have been explored to provide zero-overhead abstractions that preserve both generality and performance.

One of the earliest successful attempts to provide a generic and zero-overhead abstraction for memory layouts in C++ is the ASX library proposed by Robert Strzodka in 2014 [14]. Libraries such as Kokkos [12] and alpaka [15] focus on performance portability by providing suitable data structures like Cabana [13] and LLAMA [9] [8], with flexible data layout containers that can switch between AoS, SoA, or AoSoA representations at compile time, combining usability with high performance and portability.

The problem is that these data structures are typically designed with performance as the ob-



Figure 1: AoS Layout



Figure 2: SoA Layout

jective, overlooking the abstraction that would allow for more intuitive and flexible utilization. The goal of this thesis is to raise the level of abstraction of an existing SoA-based data structure, providing additional data layouts, improving both flexibility and usability, and preserving zero-overhead performance.

All the introduced features are implemented using alpaka [15], a C++ library that provides an abstraction layer for writing backend-agnostic code to achieve performance portability across different architectures, to manage memory allocation and data transfer. The data structure presented in this work has been tested and compared against other existing implementations in a standalone repository¹. It is suitable for general-purpose use in heterogeneous environments, such as the European Organization for Nuclear Research (CERN), where large amounts of data from particle collisions are processed using CPUs, GPUs, and other hardware accelerators, and where most of the validation and evaluation of this work has been carried out.

2. Background

Within the CMS experiment at CERN, before the introduction of a common library, each group implemented its own ad hoc SoA structures, with different implementations and optimizations, leading to varying memory usage and efficiency. To address this, a general SoA prototype was introduced in early 2023 [6]. Implemented with the CMS software (CMSSW), and using `Boost.Preprocessor` [3], the data structure is generated at preprocessor time: the user specifies the name and type of the columns

¹<https://github.com/cern-nextgen/wp1.7-soa-benchmark>

through macros, and these symbols are then propagated to define the SoA. It is possible to declare three types of columns: scalars, a single value for the whole SoA; columns, represented by a raw pointer and a runtime size; and Eigen [10] columns, i.e., columns that benefit from optimized linear algebra operations.

This design echoes the strategy seen in SoAx [11], but was developed specifically for the CMS software and integrated with alpaka to enable performance portability; in this way, the same code can efficiently run across heterogeneous architectures such as CPUs and GPUs without sacrificing performance. Both host and device memory allocations are managed by the `PortableCollection` class, which provides a constructor for allocating memory, an interface for accessing the data, and support for host/device data transfers. As the level of abstraction increases, performance evaluations are conducted to verify that the added flexibility does not introduce any overhead relative to this original baseline.

3. Proposed methodology

A set of optimizations and extensions is introduced to improve both the level of abstraction and the computational performance of the data structure, enhancing usability, flexibility, and maintainability while preserving portability across different architectures.

3.1. Increasing level of abstraction

In many complex workflows, it is important to make the data structure more representative of the object being processed. Enhancing the semantic coherence of the structure can improve readability and ease of use without sacrificing efficiency. For this reason, several extensions have been developed. In particular, a new abstraction that allows representing more complex objects using multiple SoAs and custom methods, while remaining portable across heterogeneous architectures, will be introduced. Finally, following the state of the art, an AoS layout is added to the existing memory layouts.

3.1.1 SoABlocks

the SoABlocks structure allows grouping multiple columns of different sizes within a single

contiguous memory blob, allocated through alpaka to ensure portability across heterogeneous architectures. Moreover, by enabling the generation of user-defined methods using dedicated macros, either operating on the whole structure or on single elements (for instance, providing an appropriate `operator[]` for the `SoABlocks`), it becomes possible to represent complex objects while maintaining the performance of the underlying SoAs. Two examples that are going to be used within the reconstruction in CMSSW are the `GraphSoA` (fig. 3), where two SoA blocks represent node and edge features, and the association map, with two columns, one for the offsets and one for the actual content to retrieve. Of course, the column of offsets is smaller than the other one, and, in general, the number of nodes in a graph is always less than or equal to the number of edges.

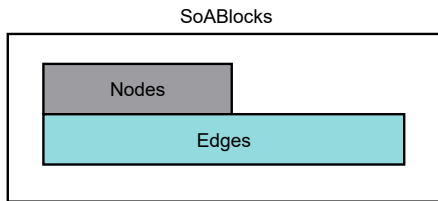


Figure 3: `SoABlocks` representing a graph

3.1.2 AoS layout

To cover the use cases where data is accessed element-by-element or in an irregular way, an AoS memory layout has also been implemented and integrated in the code. The design and implementation follow the same strategy of the SoA, i.e., exploiting code generation with macros, and ensure that switching between SoA and AoS requires only minimal changes to the user code, confirming that the data-access syntax remains general and layout-agnostic, in line with the state-of-the-art data structures that support multiple memory layouts. Memory allocation is handled through `PortableCollections`, ensuring that the AoS layout remains portable across architectures. Furthermore, a backend-agnostic kernel using alpaka for a straightforward conversion from SoA to AoS and viceversa has been implemented.

3.2. Performance optimizations

Certain reconstruction algorithms operate only on a subset of the available data, and accessing unnecessary columns or performing copies may introduce overhead. To address this, the concept of data *View* has been extended so that it can refer not only to a single and contiguous SoA layout, but also to columns belonging to different SoAs. The column heterogeneity must be preserved to ensure portability across different backends.

3.2.1 Generic View

A lightweight object that can wrap references to the required columns, regardless of their physical location in memory, is introduced. As an example, one may consider a SoA storing the two-dimensional positions of a set of particles, and another SoA storing their velocities. It is possible to wrap the references to the x-direction columns and access the data using an object without performing copies (fig. 4).



Figure 4: Building a generic View from x-direction columns for position and velocity

It is then possible to consolidate these columns and create a new contiguous SoA layout through a `deepCopy()` function (fig. 5). Since it uses alpaka for copying data, the function is portable, and the buffers can be located on any device supported by alpaka.

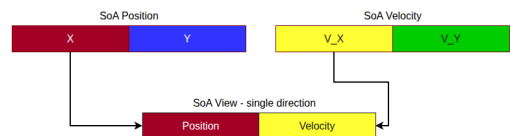


Figure 5: `DeepCopy` the x-direction columns for position and velocity

3.2.2 Generic View for `SoABlocks`

The same mechanism can be applied for `SoABlocks`, wrapping existing SoAs into a single `SoABlocks` data view, i.e., with SoAs playing

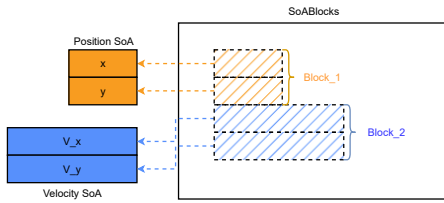


Figure 6: The View of a SoABlocks points to two different SoAs

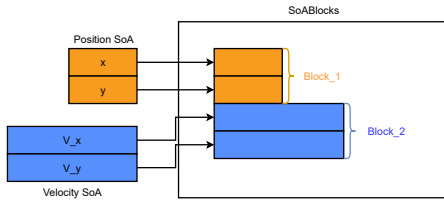


Figure 7: Building the SoABlocks by copying the data from different SoAs

the role of the columns, and eventually copying the data (fig. 6 and fig. 7).

Furthermore, the interface for the described generic View provides access to metadata such as alignment and stride, which were previously not exposed. This has enabled the CMS SoA to be used as a valid data format for heterogeneous ML inference with PyTorch, without requiring host-device data conversions, by directly manipulating its metadata to ensure correct data access.

4. Validation and results

The impact of the introduced extensions and performance optimizations has been evaluated within the CMSSW environment. The experimental setup is designed to represent a realistic compute-bound application, increasing the problem size to study scaling behavior. In particular, first, each thread reads some values from the structures. Then, it processes a sequence of fused multiply-add (FMA) and trigonometric functions, and performs writing operations to update the structures. Benchmarks were executed on an NVIDIA L4 GPU, keeping the number of threads fixed (512) and varying the number of blocks based on the input size. For every experiment, measurements are averaged over 20 independent runs, after discarding 2 warm-up executions.

The performance of SoABlocks was compared to

that obtained using multiple SoAs storing the same fields. The results show that SoABlocks exhibits the same performance as the independent SoAs (fig. 8). This confirms that grouping SoAs having different sizes into a single portable memory blob via SoABlocks introduces no performance penalty, while improving the expressiveness and coherence of the data representation.

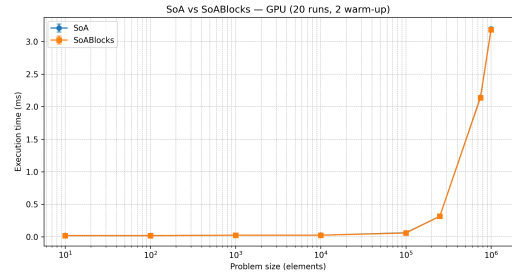


Figure 8: SoABlocks vs traditional SoAs in GPU

To validate the AoS support, the conversion function between SoA and AoS layouts was also benchmarked. A structure composed of six fields (one scalar, four columns, and one Eigen column) was used, and the time required to convert from AoS to SoA and from SoA to AoS was measured. The results show a slight performance advantage (2x speedup) when converting from AoS to SoA (fig. 9). This is likely due to the kernel performing coalesced reads and non-coalesced writes in the AoS to SoA conversion, while the opposite requires irregular reads, which are generally more expensive due to cache inefficiency.

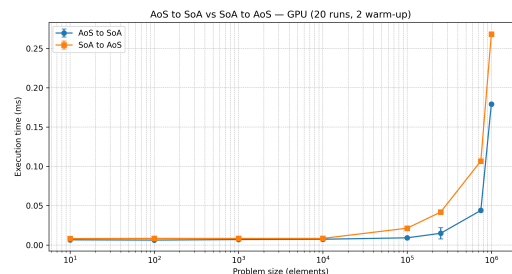


Figure 9: Data structure conversion in GPU

To evaluate the performance improvement of referencing columns from different SoAs through the new View interface, the same benchmark was performed in two configurations: one in which the selected columns were first copied into a new compact SoA layout, and one where they

were accessed through the lightweight wrapper without copying. The results show a clear performance improvement when using the new approach (fig. 10). This is expected, since memory transfer is usually the major bottleneck in GPU applications.

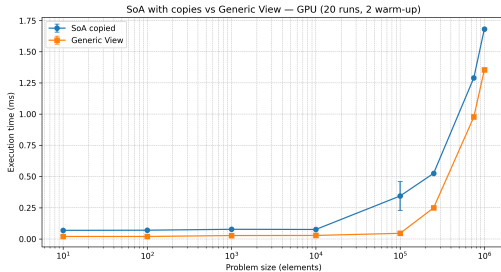


Figure 10: Copy the data vs Generic View in GPU

The direct use of CMS SoA layout as input and output tensor for heterogeneous ML inference in PyTorch, without format conversion or intermediate data transfer, has finally been evaluated within CMSSW. The tested model is a Graph Neural Network (GNN) used for particle reconstruction, where individual detector hits are represented as nodes, and possible physical connections between them are represented as edges. The performance improvement is evident, reaching about 25x speedup for an input size of 700 nodes and about 30k edges (fig. 11).

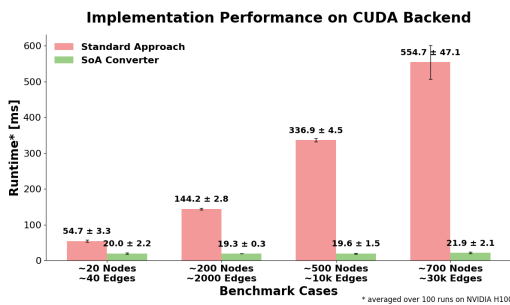


Figure 11: SoA converter vs Standard approach

In addition to these evaluations, the CMS data structure has also been tested within the context of the Next Generation Trigger (NGT) [5] project at CERN, where it was compared against two other data structures: one based on template metaprogramming, and one exploiting the experimental C++ compile-time reflection [4] [1]. The comparisons were performed in a CPU-only environment due to the still limited portability of the two considered implementa-

tions. Two of the benchmark kernels used in this study were the *N-body* and the *invariant mass* algorithms, representative examples commonly used in numerical simulations. The results show identical performances for such cases (fig. 12 and fig. 13); a larger suite of benchmarks is available in the repository referenced in section 1.

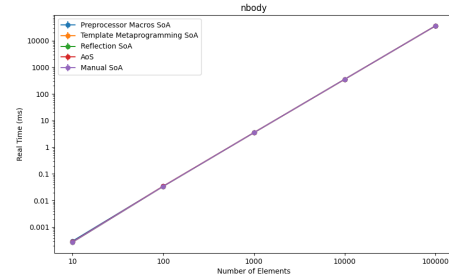


Figure 12: N-body comparison in CPU

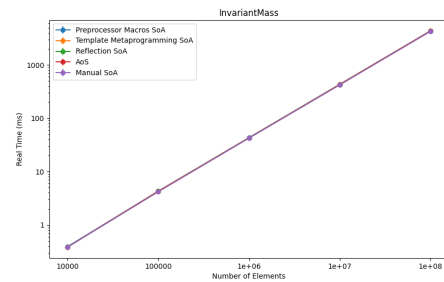


Figure 13: Invariant mass comparison in CPU

As a consequence, the focus shifts from performance to readability, portability, maintainability, and usability within large-scale frameworks such as CMSSW.

5. Conclusion and future works

This thesis addresses the design and implementation of a portable and efficient data structure for heterogeneous computing workflows, demonstrating that a high level of abstraction is achievable without sacrificing efficiency.

It is now possible to wrap columns from different data structures into a lightweight portable object and to represent complex entities through the SoABlocks abstraction. Thanks to this flexibility, significant performance improvements have been observed, for example, in heterogeneous ML inference.

Future developments will focus on reducing implementation complexity and improving readability and maintainability. In the future, the

introduction of compile-time reflection, starting from C++26, will make it possible to replace macro usage with code injection and manipulation using standard library utilities, simplifying implementation and improving maintainability. Additional work will aim to improve coherence between features, for example, introducing AoS-Blocks to mirror SoABlocks and provide the same conversion function as the one existing for the classic layouts, AoS and SoA.

For the layout benchmark suite for structure comparison, the benchmarks will be ported to GPU to evaluate performance portability and to show potential differences in efficiency across architectures and implementations.

These developments will consolidate the CMS SoA as a mature, general-purpose data model that combines portability, performance, and usability.

This work has been evaluated and validated within the context of the Compact Muon Solenoid (CMS) experiment [2], one of the four detectors of the Large Hadron Collider (LHC) [7] at European Council for Nuclear Research (CERN). In particular, the implementation and performance studies were carried out within the CMS reconstruction software framework, CMSSW, where SoAs are extensively used to efficiently process collision data for physics analysis in a heterogeneous environment.

References

- [1] Jolly Chen, Ana Lucia Varbanescu, and Axel Naumann. Optimizing Memory Access Patterns through Automatic Data Layout Transformation (Work in Progress Paper). In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering, ICPE '25*, page 47–53. ACM, 5 2025.
- [2] CMS Collaboration. The cms experiment at the cern lhc. *Journal of Instrumentation*, 3(08):S08002, 2008.
- [3] Boost/C++ community. Reference Boost Preprocessor Library for C/C++.
- [4] C++ community. C++ reflection proposal.
- [5] CERN community. Next Generation Trigger website.
- [6] 2023 E. Cano for the CMS Collaboration. Implementation of generic soa data structure in the cms software. 2023.
- [7] Lyndon Evans and Philip Bryant. LHC Machine. *Journal of Instrumentation*, 3(08):S08001, aug 2008.
- [8] Bernhard Manfred Gruber. Updates on the Low-Level Abstraction of Memory Access. 2 2023.
- [9] Bernhard Manfred Gruber, Guilherme Amadio, Jakob Blomer, Alexander Matthes, René Widera, and Michael Bussmann. LLAMA: The Low-Level Abstraction for Memory Access. 2 2022.
- [10] G. Guennebaud, B. Jacob, et al. Eigen v3. <https://eigen.tuxfamily.org>, 2010. A C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
- [11] Holger Homann and Francois Laenen. SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes. *Proceedings of 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2018.
- [12] Kokkos contributors. Kokkos Reference Documentation.
- [13] Stuart Slattery, Samuel Temple Reeve, Christoph Junghans, Damien Lebrun-Grandié, and Robert Bird. Cabana: A Performance Portable Library for Particle-Based Simulations. *The Journal of Open Source Software*, 5 2022.
- [14] Robert Strzodka. Abstraction for AoS and SoA Layout in C++. *GPU Computing Gems Jade Edition*, 12 2012.
- [15] Erik Zenker, Benjamin Worpitz, Rene Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang Nagel, and Michael Bussmann. Alpaka - an abstraction library for parallel kernel acceleration. 02 2016.