SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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

Tesi di Laurea Magistrale in
High Performance Computing Engineering - Ingegneria del calcolo ad alte prestazioni

Author: **Leonardo Beltrame**

Student ID: 10937121
Advisor: Prof. Cristina Silvano
Co-advisors: Davide Gadioli, Felice Pantaleo
Academic Year: 2025-26

Abstract

In this thesis, a general-purpose, efficient, and portable data structure is presented, with direct applications in high-energy physics, particularly within the CMS experiment at CERN. A review of the data structure literature was conducted to enhance the existing Structure of Arrays (SoA) implementation in CMSSW. The new design increases abstraction and introduces new layout representations, including an Array of Structures (AoS) and automatic conversion mechanisms. Further extensions, such as SoA blocks and advanced composition tools, enable greater flexibility and performance portability between CPUs and GPUs through the Alpaka framework. The developed solution provides a user-friendly and flexible interface while maintaining high efficiency and zero overhead. A comparison with existing academic and industrial implementations has been performed, outlining future directions for data structure development within CMSSW and similar high-performance computing environments. The work is part of the Next Generation Trigger project, aiming to improve the computational efficiency of the CMS reconstruction software (CMSSW) across heterogeneous architectures.

Keywords: Data Structures, Structure of Arrays, Performance Portability, Alpaka, CMS Software, High-Performance Computing

Abstract in lingua italiana

In questa tesi viene presentata una struttura dati generica, efficiente e portabile, con applicazioni dirette nell'ambito della fisica delle alte energie, in particolare nell'esperimento CMS al CERN. È stata condotta un'analisi della letteratura sulle strutture dati con l'obiettivo di migliorare l'implementazione esistente della struttura SoA (Structure of Arrays) all'interno di CMSSW. Il nuovo design aumenta il livello di astrazione e introduce nuove rappresentazioni di layout, tra cui l'Array of Structures (AoS) e meccanismi di conversione automatica tra i due modelli. Ulteriori estensioni, come i SoA Blocks e strumenti avanzati di composizione, consentono una maggiore flessibilità e garantiscono portabilità delle prestazioni tra CPU e GPU tramite il framework Alpaka. La soluzione sviluppata fornisce un'interfaccia flessibile e facile da utilizzare, mantenendo al contempo alta efficienza e assenza di overhead. È stata inoltre effettuata una comparazione con implementazioni accademiche e industriali esistenti, delineando le possibili direzioni future per l'evoluzione delle strutture dati all'interno di CMSSW e in altri contesti di calcolo ad alte prestazioni. Il lavoro si inserisce nel progetto Next Generation Trigger, con l'obiettivo di migliorare l'efficienza computazionale del software di ricostruzione di CMS su architetture eterogenee.

Parole chiave: Strutture Dati, Strutture di Array, Portabilità, Alpaka, CMS Software, Calcolo ad alte prestazioni

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 Background **5**

2.1 Foundations of Data Structures . . . 5

2.1.1 Array of Structures . . . 5

2.1.2 Structure of Arrays . . . 8

2.1.3 Hybrids and other solutions . . . 9

2.2 Boost Macros . . . 10

2.3 Performance portability . . . 12

2.4 Context and Motivation at CERN . . . 14

2.5 Data structures in CMS . . . 17

3 State of the Art **19**

3.1 Existing implementations of efficient data structures . . . 19

3.1.1 Template metaprogramming solutions for SoAs . . . 19

3.1.2 Code generation for data structures definition . . . 27

3.2 Reflection-based SoAs . . . 31

3.3 The generic CMSSW SoA . . . 35

3.3.1 Challenges for efficient data structures in CMSSW . . . 35

3.3.2 Definition . . . 36

3.3.3 Template metaprogramming . . . 37

3.3.4 Achieve portability . . . 39

4 Proposed methodology **43**

4.1 User experience and level of abstraction increasing . . . 43
4.1.1 Simplifying SoA backend . . . 43
4.1.2 User-defined methods . . . 45
4.1.3 SoA blocks . . . 48
4.1.4 AoS wrapper . . . 51
4.2 Performance-oriented features . . . 53
4.2.1 Generic SoA composition . . . 53
4.2.2 Heterogeneous data transfer . . . 56
4.2.3 Generic SoABlocks composition . . . 58
4.2.4 Heterogeneous ML inference in CMSSW . . . 60
4.2.5 Data structure conversion . . . 62

5 Results **65**
5.1 Increased the level of abstraction . . . 65
5.1.1 Custom methods for SoA elements and View . . . 65
5.1.2 SoA Blocks . . . 67
5.1.3 AoS wrapper . . . 68
5.1.4 Optimize data access . . . 69
5.2 Analyze performance improvements . . . 71
5.2.1 Generic View . . . 71
5.2.2 Heterogeneoeus data transfer . . . 74
5.2.3 Improvement of ML inference . . . 76
5.2.4 Data structure conversion . . . 77
5.3 Comparison between different SoA implementations . . . 78

6 Conclusions and future developments **83**

Bibliography **87**

List of Figures **95**

Acknowledgements **97**

1 | Introduction

Efficient data structures are a central aspect of high-performance computing, particularly in complex domains where large volumes of data must be processed rapidly and repeatedly. One of the key design choices influencing algorithmic performance is the memory layout, i.e, how data is arranged and accessed in memory. Two of the most famous paradigms are the Array of Structures (AoS) and the Structure of Arrays (SoA).

The AoS layout is an example of object-oriented programming, since each element is represented by a structure containing all of its fields, and is generally straightforward to read and use. However, AoS tends to perform poorly in highly parallel environments, particularly when only a subset of fields must be accessed across many elements. In contrast, the SoA layout stores each field of the data structure in its own contiguous array. This representation increases spatial locality for computations that operate on a single field across many elements and enables coalesced GPU memory accesses, making the code more efficient but the code less readable and intuitive to develop. The difference between the two memory layouts is illustrated in fig. 1.1.

As a result, numerous research efforts and software libraries have attempted to provide data access abstractions over SoA layout. These tools aim to maintain the performance of SoA while offering the high-level access pattern of AoS, improving productivity and portability across different memory representations and architectures.

However, the majority of existing solutions are primarily designed with performance as the main focus, often providing limited abstraction and reduced flexibility in manipulating the data. This typically forces developers to invest significant effort in adapting or rewriting code, and frequently it requires changes to data formats to ensure compatibility between different software modules.

The goal of this thesis is to enhance a pre-existing generic SoA implementation by increasing its level of abstraction while maintaining or improving performance on heterogeneous computing architectures. The work aims to make the SoA easier to use for developers, especially those not deeply familiar with template metaprogramming or GPU programming, without compromising the zero-overhead execution model that is essential

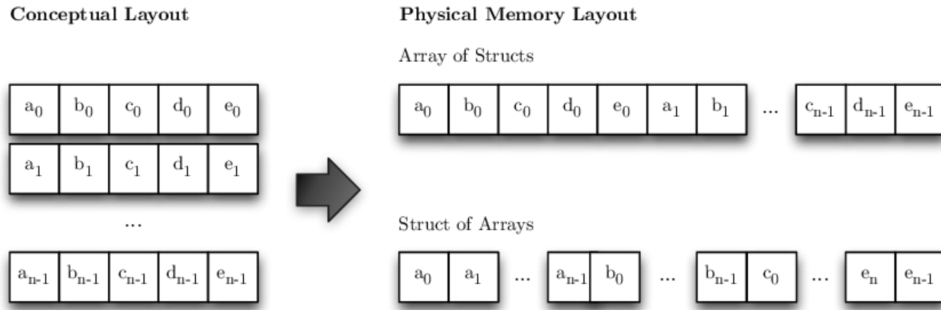

Figure 1.1: A comparison of the Array-of-Structs (AoS) and Struct-of-Arrays (SoA) memory layouts [51]

in high-performance computing. The result consists of:

- A more maintainable and readable backend for future developers;
- Additional features for straightforward data manipulation and aggregation;
- Extended layout support, including an AoS representation and seamless AoS/SoA conversion;
- Deployment within a heterogeneous and portable framework.

A proposed methodology to design and implement a data structure where abstraction and performance are not mutually exclusive will be presented, as well as all the features that can be used within a complex and heterogeneous framework. Furthermore, these extensions enable concrete computational optimizations: custom views, support for different memory layouts, structure composition, and selective column operations, leveraging only the data that is relevant to a given algorithm, improving memory locality, and reducing unnecessary data movement. Without these improvements, the original implementation did not allow flexible data manipulation, making it hard to optimize heterogeneous algorithms and to avoid unnecessary memory transfer.

The evaluation and validation of this work have been done within the CMS experiment at CERN [20], one of the major detectors operating at the Large Hadron Collider [34]. In this detector, proton collisions are recorded and reconstructed to study fundamental physics processes. The reconstruction software must handle increasingly large volumes of data across heterogeneous computing architectures; therefore, efficient, portable data layouts and memory models are essential to ensure scalable, high-performance event processing.

The extended SoA implementation presented in this work is currently deployed in the CMS software framework. The improvements introduced here have been validated in realistic workflows and benchmarked on both CPU and GPU execution backends.

In parallel with the development work, this thesis includes a comparison with state-of-the-art data layout frameworks, based on template metaprogramming and compile-time reflection. This comparison evaluates their usability, flexibility, and performance characteristics, positioning the proposed solution within the broader research landscape.

Furthermore, the improved SoA infrastructure has been integrated into a heterogeneous machine learning inference workflow within the CMS software environment. In particular, a GPU-based model implemented in PyTorch has been adapted to directly operate on the SoA memory buffer by managing strides and columns without performing unnecessary data copies between CPU and GPU, which is usually the bottleneck of hardware-accelerator applications.

In particular, the thesis is structured as follows:

- **Chapter Two:** provides background on fundamentals of data structures and layouts, Boost macro utilities, performance portability frameworks, and introduces the CMS computing environment.
- **Chapter Three:** discuss state-of-the-art data structure design and implementation approaches, highlighting performance and usability trade-offs.
- **Chapter Four:** presents the improved data structure design, including the abstraction mechanisms, layout extension support, and new compositional features.
- **Chapter Five:** evaluates and validates performance and usability through benchmarks and comparisons.
- **Chapter Six:** concludes the thesis and discusses future development directions.

2 Background

This chapter provides the necessary background to understand the work illustrated in this thesis. The main data structure layouts, specifically the Array of Structures (AoS) and the Structure of Arrays (SoA), are presented. Then, key concepts used throughout the thesis are discussed, including the use of preprocessor macros for code generation and the role of performance portability libraries. Finally, the research context in which this work has been carried out is described: the CMS experiment at CERN, its data processing challenges, and the motivations driving the development of portable and efficient data structures for future high-luminosity operations.

2.1. Foundations of Data Structures

The choice of data layout is a fundamental factor that affects memory bandwidth utilization, cache efficiency, and the ability to vectorize or parallelize computations. Two widely used paradigms in this context are the Structure of Arrays (SoA) and Array of Structures (AoS), each with its strengths and trade-offs depending on the target architecture and access patterns. An overview of the main characteristics of these data structures will be presented.

2.1.1. Array of Structures

An AoS definition in C++ is shown in listing 2.1.

Listing 2.1: AoS definition in C++

```
1 struct Vertex {
2     float x;
3     float y;
4     float z;
5 }
6 std::vector<Vertex> vertices(10);
```

For an AoS, the data are accessed in an object-oriented programming approach, as shown

in listing 2.2.

Listing 2.2: Data access syntax for AoS

```
for (int i = 0; i < size; i++) {
    vertices[i].x = 1.f;
    vertices[i].y = 2.f;
    vertices[i].z = 3.f;
}
```

The AoS is often favored for its clarity and object-oriented design principles. By grouping all fields of a single entity, such as a particle with its position and velocity, AoS mirrors the conceptual structure of real-world objects. This makes the code more intuitive and easier to reason about, especially in smaller-scale applications or in early stages of development where clarity and maintainability outweigh raw performance.

The structure of any data is a trade-off between performance, readability, maintainability, future-proofing, extensibility, and reuse [35]. The choice of a particular data layout directly affects how efficiently compilers and hardware can process the data. In this context, auto-vectorization is a compiler optimization that transforms scalar loops into SIMD (Single Instruction, Multiple Data) instructions, enabling the parallel processing of multiple data elements. This technique is essential for exploiting the full capabilities of modern CPUs equipped with wide SIMD registers (e.g., AVX2 provides 256-bit registers, enabling operations on eight 32-bit floating-point numbers per instruction [42]).

For instance, a simple loop such as shown in listing 2.3, can be automatically converted by the compiler into a single AVX instruction operating on 8 elements at a time, greatly improving performance.

Listing 2.3: Simple C/C++ loop

```
for (int i = 0; i < N; ++i)
    a[i] = b[i] + c[i];
```

However, this optimization often fails when using AoS memory layouts, due to how the data is arranged in memory.

Taking listing 2.1 as an example, and assuming to access only one field, such as `x`, with the memory access pattern shown in listing 2.4, the memory access pattern is not contiguous. Although it appears to iterate over a single field, the actual layout in memory is interleaved.

Listing 2.4: Accumulate AoS single element

```
for (int i = 0; i < N; ++i)
    sum += vertices[i].x;
```

`vertices[0].x, vertices[0].y, vertices[0].z, vertices[1].x, ...`

As a result, the `x` values are separated by a stride equal to `sizeof(Vertex)`, which prevents efficient vector loads.

Compilers often refuse to auto-vectorize such loops because:

- The access pattern has a non-unit stride.
- Data alignment cannot be guaranteed.
- SIMD vector loads would require gather operations, which are less efficient than simple contiguous loads.

Figure 2.1: AoS Layout

As you can see in fig. 2.1, in an AoS, data fields from different properties are interleaved in memory. This means that accessing the same property across multiple elements requires skipping over unrelated data, resulting in non-contiguous, strided memory accesses. On CPUs, this layout hinders the efficient use of cache lines and typically prevents the compiler from applying auto-vectorization optimizations. Since the processor has to load more data than needed, often including irrelevant fields, it wastes bandwidth and reduces performance. On GPUs, the problem is even more pronounced: memory accesses lose coherence because threads within a warp end up reading from scattered memory locations rather than from contiguous blocks. This disrupts memory coalescing and significantly impacts throughput. For performance-critical applications, especially those operating over large datasets or in tight loops, the AoS approach can become a bottleneck.

Auto-vectorization is most effective when data is stored contiguously in memory, as in the Structure of Arrays (SoA). In contrast, AoS layouts hinder this optimization due to their interleaved memory structure. Even with aggressive compiler flags (e.g., `-O3`,

`-ftree-vectorize`, `-march=native`), compilers typically cannot vectorize strided access patterns efficiently.

2.1.2. Structure of Arrays

A Structure of Arrays (SoA) in C++ can be defined with C-arrays within a struct, as it can be seen in listing 2.5.

Listing 2.5: SoA definition in C++ with pointers

```
struct Vertices {
    float x[100];
    float y[100];
    float z[100];
};
```

Alternatively, one could use any type representing arrays, as raw pointers, `std::arrays`, or `std::vectors`.

The SoA, as it can be seen in fig. 2.2, is made by contiguous fields in memory and organizes data by separating each field into its own contiguous array. This structure enables highly efficient memory access patterns, particularly for performance-critical applications. When the same property must be accessed across many elements, such as updating the x coordinate for all vertices, SoA allows sequential, cache-friendly access, making it ideal for vectorized operations on CPUs and memory-coalesced accesses on GPUs. Compilers can more easily apply SIMD instructions due to the absence of interleaved data, and processors can avoid loading unnecessary fields, leading to better use of memory bandwidth. This makes SoA especially suited for data-parallel workloads in high-performance computing, numerical simulations, and real-time rendering engines.

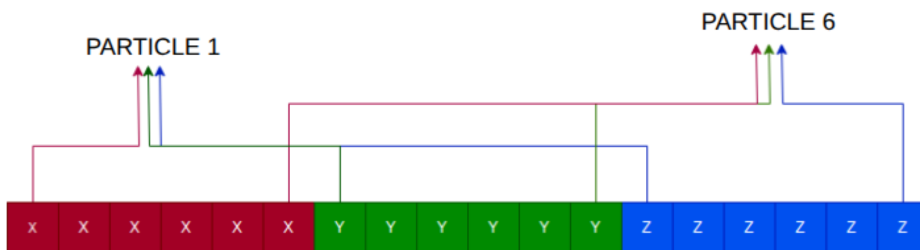

Figure 2.2: SoA Layout

The SoA data access pattern is data-oriented, as shown in listing 2.6.

Listing 2.6: Data access syntax for SoA

```
Vertices vertices;

for (int i = 0; i < size; i++) {
    vertices.x[i] = 1.f;
    vertices.y[i] = 2.f;
    vertices.z[i] = 3.f;
}
```

While it boosts performance, it often reduces code readability, and it is a perfect example of data-oriented design. Developers must manually coordinate multiple arrays representing a single entity, which can introduce complexity and increase the likelihood of maintenance errors.

2.1.3. Hybrids and other solutions

An Array of Structures of Arrays (AoSoA) can be thought of as an AoS where each "particle" is a small SoA. Writing an AoSoA in C++ requires defining two numbers:

1. the size of the fields within the SoAs
2. the size of the AoSoA

An example representing a vector of molecules' trios, each having 3 atoms, is shown in listing 2.7. In this way, it could be possible to define a sequence of three molecules, each having three atoms.

Listing 2.7: AoSoA definition in C++

```
constexpr int atoms_per_molecule = 3;
constexpr int num_molecules = 1000;

struct Molecules {
    std::array<std::string, atoms_per_molecule> A;
    std::array<std::string, atoms_per_molecule> B;
    std::array<std::string, atoms_per_molecule> C;
};

std::vector<Molecules> system(num_molecules);
```

To reconcile the performance benefits of SoA with the readability and structural clarity of AoS, AoSoA, or Structure of Arrays of Structures (SoAoS) have emerged. These approaches divide data into small, fixed-size blocks, where each block stores a chunk of data in either an AoS or SoA form, and they offer a compromise between cache efficiency and

code maintainability. For instance, as can be seen in fig. 2.3, AoSoA groups multiple objects together in a structure-of-arrays format and then arranges those groups in an array, enabling both vectorized access within blocks and improving partial locality of the data.

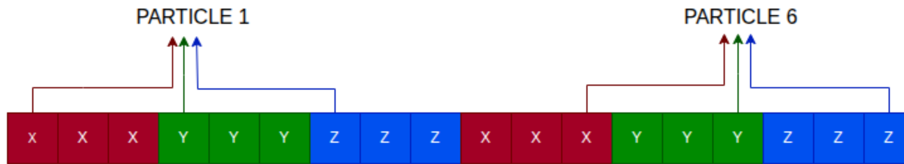

Figure 2.3: AoSoA Layout

Assuming the number of elements per block to be the same, which makes sense both in terms of complexity and memory organization, the access to the data is composed of two indices, one for the block and one for the element within the block, as shown in listing 2.8.

Listing 2.8: Data access syntax for AoSoA

```
for (int i = 0; i < num_blocks; i++)
    for (int j = 0; j < elements_per_block; j++) {
        aosoa[i].x[j] = 1;
        aosoa[i].y[j] = 2;
        aosoa[i].z[j] = 3;
    }
```

2.2. Boost Macros

Macros in C and C++ are one of the oldest and most powerful mechanisms available to developers. They operate during the preprocessing stage, before actual compilation. Figure 2.4 is a reminder of how the compilation process works. Indeed, after the preprocessing of the source code, an expanded code is generated, and it can be inspected using the `-E` flag [15]. An example can be found in listing 2.9.

Listing 2.9: Expanding the code with preprocessor flag

```
clang -E main.c -o main.i
```

The preprocessor scans the source code and performs textual substitutions, conditional inclusion, and code transformations based on the defined macro, ranging from simple definitions like `#define N 1000` to more complex examples.

Starting from CLANG 18, it is possible to add the flag `-fkeep-system-includes` which avoids the expansions of all the `#include` directives acting on system libraries [45].

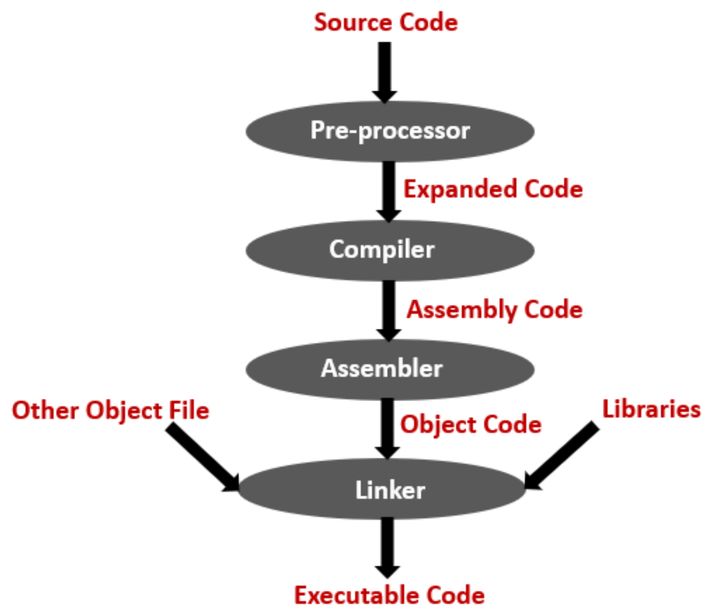

Figure 2.4: Compilation process [29]

Besides simple macros, there exist full libraries to extend the functionalities. For example, Boost Preprocessing is a library of macros that support preprocessor metaprogramming. The library supports both C++ and C compilation. It does not depend on any other Boost libraries and therefore may be used as a standalone library [23].

It originated in the early 2000s; an accessible early introduction appears in the appendix of Abrahams and Gurtovoy's C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond [1]. Once variadic macros became available in mainstream compilers (C99/C++11), the approach became more expressive and ergonomic, enabling idioms such as argument counting, macro overloading, and compile-time iteration with less boilerplate.

As discussed in literature [47], developers often describe a dual relationship with macros: they are extremely useful for portability and code reduction, but at the same time they introduce readability issues, obfuscation, and maintenance challenges. Software development with macros can lead to incomprehensible error messages that are difficult to interpret, and in some cases, even security concerns, since a slight syntax mismatch can lead to undefined behavior. In this sense, macros can be considered a form of "pushed-to-the-limit" static metaprogramming that must be handled with caution.

2.3. Performance portability

Modern scientific computing heavily relies on parallelism to meet the increasing demands of data volume and computational complexity. The advent of massively parallel hardware accelerators, particularly GPUs, has enabled substantial performance gains across a wide range of scientific applications. However, exploiting this hardware efficiently poses some challenges.

Each major vendor provides its own programming framework: NVIDIA supports CUDA C++, AMD promotes HIP/ROCm, and Intel offers OneAPI with DPC++. Despite an evident similarity between, for example, CUDA and HIP, each of these environments is optimized for its respective hardware; they are mutually incompatible, and developers have to write and maintain separate versions of their code for each architecture, or, at least, make large use of macros to enable different pieces of code when certain compilers are involved.

In the context of large-scale, complex software systems, such as those used in high-energy physics, this leads to a significant increase in development and maintenance overhead. Managing multiple software versions not only introduces redundancy but also complicates debugging, testing, and maintainability. On the other hand, if a certain software is architecture-dependent, choosing, for example, CUDA as the programming model, it loses generality and becomes less attractive for research and open-source projects.

This challenge has given rise to the concept of performance portability, i.e., the ability of a software system to run efficiently across a broad range of heterogeneous platforms, maintaining the same performance and without requiring code redundancy. Achieving performance portability is essential to ensure that scientific applications can fully exploit modern and future hardware while minimizing developer effort.

To address this need, several abstraction layers and performance portability libraries have been developed. Some of the most notable are SYCL [66], Kokkos [44], and Alpaka [2]. These frameworks aim to decouple algorithm implementation from hardware-specific execution details, allowing developers to write platform-agnostic code that can be compiled for multiple architectures.

SYCL is a C++-based model chosen as the prime programming model for Intel GPUs, and also implemented by third parties for AMD and NVIDIA GPUs [39]. The SYCL programming model is built around queues and command groups: the programmer explicitly creates a queue for a chosen device (for example, a GPU) and submits kernels via `parallel_for` lambdas. This single-source model means host and device code are in

the same C++ source file, using C++17/20 features like lambda expressions for device kernels. In terms of usability, SYCL is fairly high-level (much cleaner than native CUDA or OpenCL [63]), as shown in listing 2.10.

Listing 2.10: SYCL kernel

```
// Increment each element of an array on a GPU
sycl::queue queue(sycl::gpu_selector{}); // Choose a GPU device
float* data = sycl::malloc_shared<float>(N, queue);
queue.parallel_for(sycl::range<1>(N), [=](sycl::id<1> idx) {
data[idx] += 1; // Increment array element
});
queue.wait(); // Wait for kernel completion
```

Kokkos is a C++ performance portability library originally developed for high-performance computing. It provides abstractions for parallel execution and data management to enable applications to run efficiently on diverse architectures [30].

Originally targeting NVIDIA GPUs (via CUDA) and many-core CPUs (via threads/OpenMP), later Kokkos added an AMD GPU backend (via HIP) and even an experimental SYCL backend for Intel GPUs [44] [39]. The programming model in Kokkos is centered on parallel patterns (like `parallel_for`, `parallel_reduce`) and memory abstractions called *Views*. Developers can write a kernel as a functor or lambda (marked with the `KOKKOS_LAMBDA` macro) and Kokkos dispatches it on the selected execution space, depending on the desired backend. The trade-off is that Kokkos requires a compilation for each target backend (e.g., you must compile with CUDA enabled to run on an NVIDIA GPU), but the source code remains largely unchanged across platforms. currently. It can be compiled and executed for only one back-end at a time, making it not suitable for large software deployments [14]. Despite this, the user syntax for Kokkos is very comfortable, as shown in listing 2.11.

Listing 2.11: Kokkos kernel

```
// Kokkos: Increment each element of an array
Kokkos::View<int*> arr("arr", N); // Allocate array of N ints in device
    memory
Kokkos::parallel_for(N, KOKKOS_LAMBDA(const int i) {
arr(i) += 1; // Increment array element
});
```

Alpaka is a header-only C++17 library supporting CPUs, GPUs, and even FPGAs [71] by exploiting template metaprogramming to define the device and its properties. In contrast to SYCL and Kokkos, Alpaka exposes a lower-level, CUDA-like programming

model: it implements a hierarchical parallelism paradigm with explicit grid, block, and thread indexing [9].

Alpaka's syntax closely resembles native CUDA, which facilitates its integration into existing parallel workflows originally implemented with CUDA, as shown in listing 2.12.

Listing 2.12: Alpaka kernel

```
// Alpaka: Increment each element of an array on a GPU
using Acc = alpaka::acc::AccGpuCudaRt<1, 1>; // 1-dimensional CUDA
    accelerator
auto dev = alpaka::getDevByIdx<Acc>(0); // Select GPU device 0
alpaka::Queue<Acc, alpaka::Blocking> queue(dev); // Create a blocking
    queue on this device
unsigned threads = 256;
unsigned blocks = (N + threads - 1) / threads; // number of blocks given
     N elements
auto workDiv = alpaka::WorkDiv<1>(blocks, threads, 1);
auto kernel = [] ALPAKA_FN_ACC (auto const &acc, int* arr, size_t N) {
uint32_t idx = alpaka::getIdx<alpaka::Grid, alpaka::Threads>(acc)[0];
if (idx < N) arr[idx] += 1; // Increment element at global index
};
alpaka::enqueue(queue, alpaka::createTaskKernel<Acc>(workDiv, kernel,
    arr_dev, N));
```

In Alpaka, kernels are defined as function objects and have several requirements for the definition of the `operator()` [5]:

- it must be templated on the accelerator type
- the accelerator must be passed as the first parameter by const-reference
- it must be const, so no parameters of the struct can be changed
- it must be marked with the `ALPAKA_FN_ACC` macro, to indicate that it will be executed on the accelerator

2.4. Context and Motivation at CERN

The Large Hadron Collider (LHC) [34] is the world's most powerful particle accelerator, located 100 meters underground at the European Organization for Nuclear Research (CERN) near Geneva, Switzerland. It consists of very high-energy particle beams that circulate in a 27-kilometer ring and collide at four main interaction points, called detectors. At each interaction point, a large detector collects data from the collision and

propagates them to make decisions and apply various algorithms.

These detectors are A Toroidal LHC ApparatuS (ATLAS) [19], A Large Ion Collider Experiment (ALICE) [18], Large Hadron Collider beauty (LHCb) [22], and the Compact Muon Solenoid (CMS) [20]. Among them, the CMS is a general-purpose detector designed to study a wide range of physics, and it is the research context of this thesis.

The High Luminosity (HL)-LHC period, also known within CMS as Phase-2, represents a major upgrade of the Large Hadron Collider (LHC). The LHC collides tiny particles of matter (protons) at an energy of up to 14 TeV in order to study the fundamental components of matter and the forces that bind them together. The High-Luminosity LHC will make it possible to study these in more detail by increasing the number of collisions by a factor of 5-7.5 relative to the nominal LHC design [21].

The fig. 2.5 below illustrates the CMS Next Generation Trigger (NGT) demonstrator, which represents the conceptual data-flow model for the upgraded HL-LHC and the related reconstruction system. Starting from the 40 MHz collision rate, the Level-1 Trigger (L1T) performs a first hardware-based data selection, reducing the rate to about 750 kHz. The High-Level Trigger (HLT) then processes these events in software using large farms of CPUs and GPUs, applying advanced reconstruction algorithms to bring the rate down to approximately 10 kHz of events stored for offline analysis. The whole process is performed exploiting both parallel and distributed computing. The component circled in red highlights the focus of this thesis work: efficient data structures.

Since 2016, the CMS experiment has been actively exploring the use of Graphics Processing Units (GPUs) to accelerate its event reconstruction algorithms [7]. In the following years, several algorithms were implemented in CUDA, demonstrating significant performance gains. The success of this integration led to a gradual migration toward performance portability, and in 2023–2024, CMS transitioned to the Alpaka library [71], a C++ abstraction layer enabling heterogeneous execution across CPUs, GPUs, and other accelerators such as AMD or Intel devices.

The increasing heterogeneity and complexity of computing resources make data representation a central aspect of performance. To efficiently execute the same reconstruction algorithms on different hardware, it is essential to adopt data structures that can easily be adapted for different hardware characteristics, minimize memory transfers, maximize cache reuse, and enable coalesced memory access on accelerators and vectorization on the CPU. In particular, the Structure-of-Arrays (SoA) model has emerged as the most suitable approach for high-performance, parallel computing, as it improves vectorization and memory throughput compared to the traditional Array-of-Structures (AoS) layout.

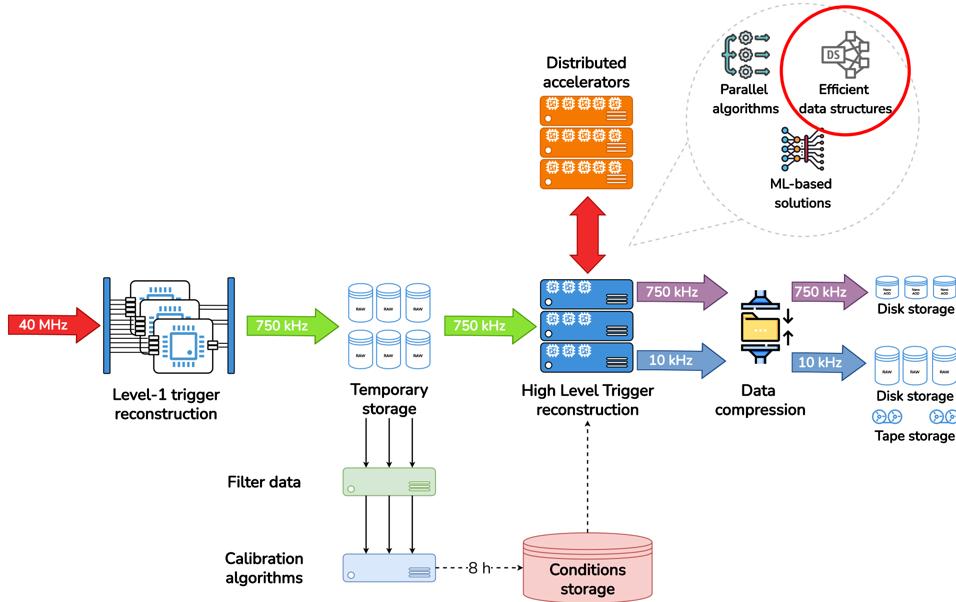

Figure 2.5: Next Generation Trigger for CMS HLT

The motivation for this thesis is thus to provide a modern, efficient, and flexible data structure to CMS, capable of supporting heterogeneous reconstruction algorithms across different architectures, while remaining easy to use and integrate within the CMSSW software ecosystem. This work aims to describe the current SoA used in CMS and to design and implement new features and data layouts that achieve the following goals:

- Support different memory layouts without rewriting algorithms.
- Allow developers to write generic code for accessing the data without committing to a specific memory layout.
- Achieve zero-overhead abstraction to obtain generality without performance loss.
- Ensure portability and efficiency across CPUs, GPUs, and other accelerators.
- Ensure maintainability and readability of the code.
- Provide useful features to obtain seamless integration within the CMS software framework.

CMS has evaluated several performance portability frameworks, considering factors such

as programming experience, ease of development, integration feasibility within the CMS software framework, performance relative to native implementations, CPU fallback behavior, user support, available back-ends, and long-term sustainability.

While SYCL does not yet provide optimal performance and Kokkos limits compilation to a single back-end at a time [44], Alpaka has been chosen because it satisfies all integration requirements and delivers performance comparable to native implementations on both CPUs and GPUs [52] [4] [9].

By modernizing and improving the data structures, this research directly contributes to the Next Generation Trigger (NGT) and Phase-2 CMS computing goals, enabling future algorithms, including machine learning-based ones, to improve flexibility and efficiency while maintaining the same usability and maintainability.

2.5. Data structures in CMS

The CMS software (CMSSW) [16] is organized around two pillars: the *Framework*, and the *Event Data Model* (EDM). Their primary goal is to make the development and deployment of reconstruction and analysis software efficient and modular.

Event processing is performed by a single executable, `cmsRun`, which hosts many plug-in modules (independently compiled C++ components). All domain logic, i.e., calibration, reconstruction, and filtering algorithms, resides in these modules. The same executable is used for both detector and Monte Carlo data.

At runtime, `cmsRun` is configured by a user-provided file that specifies [17]:

- which input data to read,
- which modules to run,
- parameter settings for each module,
- the execution order (*paths*),
- event filters within each path,
- how paths map to output files.

The EDM centers on the *Event*, a C++ object holding all raw and reconstructed data for a single collision. Modules exchange data exclusively through the Event. Event products can be stored in ROOT files, enabling direct browsing, module-level testing, and advanced statistical analysis. These files leverage ROOT's `TTree` structure, storing event products

as individual branches to enable selective I/O and on-demand data access. Auxiliary, non-event data required for processing are provided via the *EventSetup*.

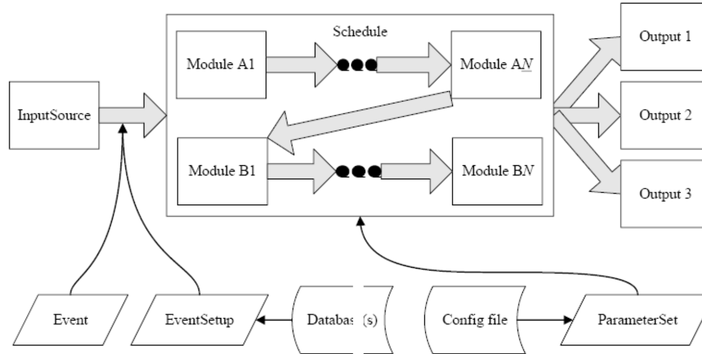

Figure 2.6: CMSSW Framework diagram [17]

To achieve performance portability without losing flexibility and generality, a good implementation strategy requires three layers:

1. Memory Layout: abstract organization of memory: AoS/SoA choice, field ordering, alignment, and the schema visible to ROOT for I/O.
2. Data access patterns (Views and ConstViews): lightweight, non-owning objects that expose read (and, when mutable, write) access to the structure values via stable, typed APIs.
3. Memory allocation and ownership: the objects that manage memory buffers, lifetimes, and transfers in different memory spaces (host or device). These containers must be able to allocate both host and device memory.

Through libraries like Alpaka, it is possible to achieve portable memory allocation and management that is portable across different architectures.

3 | State of the Art

The complexity and heterogeneity of modern computing architectures, ranging from multi-core CPUs to GPUs and specialized accelerators, have made efficient data management a critical concern in high-performance computing (HPC). Democratizing the utilization of efficient data structures in HPC is a challenge, requiring simple mechanisms for defining and switching data layouts, while ensuring portability and high performance across diverse platforms. This chapter reviews the state of the art in data structure organisation and performance portability frameworks that address these challenges. We first focus on data layout options, such as Structure of Arrays (SoA) and Array of Structures (AoS), and how the choice can impact computational throughput, particularly in the context of heterogeneous systems. Then, the main approaches to satisfy usability and efficiency constraints are presented.

3.1. Existing implementations of efficient data structures

In this chapter, some state-of-the-art implementations of efficient and generic data structures are presented, as well as an analysis of performance and usability. Some approaches involving template metaprogramming, code generation, and preprocessing macros will be shown. Finally, a possible future development for data structures' implementation using C++ reflection will be analyzed.

3.1.1. Template metaprogramming solutions for SoAs

The problem of efficiently and easily mapping data access performance and syntax to diverse hardware architectures and layouts has been central in HPC for decades. Scientific workloads, such as reconstruction algorithms in high-energy physics, particle simulations, fluid dynamics, and statistics, often involve processing large collections of records with identical fields but different values. The choice of memory layout for these records has a significant impact on performance, particularly in heterogeneous architectures that com-

bine CPUs, GPUs, and accelerators. As said in chapter 2, the choice of the most suitable data structure is crucial for HPC applications. The gain in terms of execution time can be around one order of magnitude in memory-bound workflows, i.e, when memory accesses represent the bottleneck of the algorithms [48] [69] [61] [65] [40] [58] [62]. The SoA layout is generally favored for vectorization and memory coalescing, while AoS offers more intuitive syntax and better spatial locality for some specific access patterns. Hybrid layouts such as AoSoA attempt to balance these trade-offs.

The key research challenges in data structure design and implementation are:

- Support different memory layouts (AoS, SoA, AoSoA, etc.) without rewriting algorithms.
- Allow developers to write generic code for accessing the data without committing to a specific memory layout.
- Achieve zero-overhead abstraction thanks to code inlining to obtain generality without performance loss.
- Ensure portability and efficiency across CPUs, GPUs, and other accelerators.
- Write maintainable and readable code for future developers.

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 [64]. The motivation behind ASX is that scientific codes often start from an AoS design because of its natural syntax, but it is often better to have an SoA memory layout underlying. Consider a simple AoS definition shown in listing 3.1 and suppose we want to achieve the first three bullet points in list 3.1.1, i.e., support also SoA, give the possibility to write generic code for both memory layouts, and without a runtime cost.

Listing 3.1: AoS definition in ASX

```
struct Particle {
    double x, y, z;
};

std::vector<Particle> particles(10);

Particle p = particles[i];
p.x += 1.0;
```

The key concepts that achieve these goals are the template metaprogramming and C++ unions [27].

Consider the piece of code shown in listing 3.2 from [64].

Listing 3.2: Flexible Element definition in ASX

```
template < ASX::ID t_id = ASX::ID_value >
struct FlexibleElement {
typedef ASX::ASAGroup <double, t_id> ASX_ASA;
    union { double x; ASX_ASA dummy1; };
    union { double y; ASX_ASA dummy2; };
    union { double z; ASX_ASA dummy3; };
};
typedef FlexibleElement <> Particle;
```

The definition of a `FlexibleElement` type is the core of the library. Instead of representing each record field as a fixed value, ASX introduces a union that overlays two possible representations:

1. the value representation, which behaves like a standard field in an AoS record;
2. a proxy object `ASAGroup`, which encodes access to the corresponding SoA array at a given index.

Since the unions guarantee that both members live in the same memory address (the size of a union equals the size of its largest member), no extra memory is allocated for the proxy face, and each field can be viewed either as a plain value (AoS face) or as a proxy (SoA face). At compile time, ASX specializes access so that in an AoS container, the compiler generates direct member loads/stores, while in a SoA container, the proxy redirects loads/stores to the corresponding component array at the selected index.

The user then should use the layer combining the union-based structs defined in listing 3.2 with the layout template that defines the type shown in listing 3.3.

Listing 3.3: User syntax with ASX

```
// specify ASX::SOA instead for an SoA container
typedef ASX::Array <Particle, CONT_SIZE, ASX::AOS> Container;

// common access syntax: normal AoS code continues to function
Particle p = {1. , 2. , 3.};
Container container;

p.x = p.z;
container[5].z = 2;
```

Switching the template parameter from `ASX::AOS` to `ASX::SOA`, it is possible to change the

memory layout from AoS to SoA without modifying the user code. The data access syntax remains the same in both cases. This is possible assuming that `ASX::Array` backend provides a layout-aware `operator[]` that does not return a real `Particle&` (which would not exist in SoA), but a lightweight element proxy exposed as `Container::reference`, that a static C++ container can easily represent. That proxy binds the SoA columns (the `ASAGroup` members) to the three component arrays computing the offset through to the selected index.

The zero-overhead abstraction can be achieved using template specializations and aggressive inlining, which lets the compiler generate the same assembly code for both versions with respect to a manual structure.

In this way three out of five bullet points explained in 3.1.1 are achieved, i.e., it is possible to implement a generic class that lets the user to easily switch between AoS and SoA physical layouts, which can be comfortable in HPC applications and benchmarking, and it is also possible to access the data without taking into account the specific underlying layout with zero-overhead abstraction.

However, despite these promising results, the ASX work should nowadays be regarded primarily as a proof-of-concept. While performance portability can, in principle, be extended by allocating memory in device-specific spaces (e.g., CUDA, HIP), the actual usability and maintainability of the abstraction remain limited. The approach demonstrates that one can declare even nested structures or arrays inside records, provided that additional backend code is written, but this flexibility comes at a significant cost: the user must understand and extend the backend themselves [64]. This makes ASX valuable as a conceptual milestone, which shows that a unified AoS/SoA interface with zero-overhead is achievable, but it requires the user to have a good knowledge of C++.

Another work worth mentioning is UltimateSoA, by Vincenzo Innocente [41]. This contribution illustrates a lightweight wrapper around existing object-oriented data structures to expose them in a SoA form. The idea is again to provide an AoS-like data access interface while internally storing data in SoA layout, thus enabling vectorization and memory-friendly access without rewriting the original object hierarchy. However, this is also more a proof-of-concept rather than a production-ready library, since it is very minimal and never tested against over implementations. Its relevance lies in showing how minimal template abstractions can bridge usability and performance in heterogeneous data layouts, and in inspiring later, more comprehensive frameworks.

Another attempt to write a generic SoA data structure to facilitate design and near-seamless use (compared to AOS) of the contained objects has been SOAContainer [59],

developed for CERN LHCb in 2015. Also in this case, despite the common syntax between AoS and SoA being ensured, the effort from the user side is not negligible, the class is not general and flexible enough, and is not portable at all, since it makes use of the C++ standard library and it has the interface of `std::vector` (dynamic memory allocation, incompatible with GPUs).

The next step in this line of research has been the development of more general frameworks that aim at combining the same idea of layout abstraction with greater usability, portability, and long-term maintainability. Among these, two representative projects are presented in this thesis: Cabana [60], and LLAMA [37]. Although they share the common goal of allowing developers to write efficient algorithms with a logical syntax and without referring to a specific memory layout, they differ substantially in terms of generality, usability, and portability.

Cabana is a modern C++ library designed to provide a user-friendly interface for data structures in HPC environments as particle-based simulations, with a particular focus on performance portability. Developed within the U.S. Exascale Computing Project, Cabana builds upon the Kokkos programming model, introduced in section 2.3, and it provides abstractions that allow scientists and engineers to write algorithms once and run them efficiently on CPUs, GPUs, and other emerging architectures. It extends the idea of layout abstraction discussed previously in ASX and UltimateSoA, but generalizes it into a robust and production-ready framework with significant broader applicability.

The user interface looks like the code written in listing 3.4.

Listing 3.4: User syntax with Cabana

```
1  // Declare an array layout.
2  const int elements = 4;
3
4  // Declare an soa type.
5  using member_types = Cabana::MemberTypes<double, int, float, double
       [2][3], unsigned[5], float[3][2][2]>;
6  using soa_type = Cabana::SoA<member_types, elements>;
7
8  // Create an soa.
9  soa_type soa;
10
11 // Set some data with the soa.
12 double v1 = 0.3343;
13 // Accessing the third element of the first column
14 Cabana::get<0>(soa, 3) = v1;
15
```

```
float v2 = 0.992;
Cabana::get<5>(soa, 2, 1, 1, 1) = v2;
```

As can be seen, the size of the SoA is a compile-time variable, and it always exists as a struct like the one shown in listing 3.5 [67].

Listing 3.5: Equivalent struct

```
struct FooData
{
    double _d0[4];
    int _d1[4];
    float _d2[4];
    double _d3[4][2][3];
    unsigned _d4[4][5];
    float _d5[4][3][2][2];
};
```

such that `sizeof(FooData) = sizeof(soa_type)`.

An efficient and user-friendly way to iterate over the fields is shown in listing 3.6, where `Cabana::slice` returns a `Kokkos::View`.

Listing 3.6: Cabana data selection

```
auto pos = Cabana::slice<0>(particles); // access position
auto vel = Cabana::slice<1>(particles); // access velocity
auto time = Cabana::slice<2>(particles); // access time

for (size_t i = 0; i < N; i++) {
    vel(i, 0) = pos(i, 0) / time(i);
    vel(i, 1) = pos(i, 1) / time(i);
    vel(i, 2) = pos(i, 2) / time(i);
}
```

Cabana supports the three main layouts explained in chapter 2, i.e., AoS, SoA, and AoSoA. It offers optimal performance portability thanks to the backend of Kokkos, and achieves zero-overhead abstraction using STL compile-time containers and variadic templates. However, one could notice that it is not possible to iterate over the SoA columns using the fields' names chosen by the user. This can be annoying when developing algorithms involving complex indexing, and it can lead to less readable and maintainable code.

LLAMA (Low-Level Abstraction for Memory Access), introduced by Gruber et al. in 2022 [37], is one of the most promising C++ libraries that exploit template metaprogramming

to support different memory layouts, achieve zero-overhead abstraction and performance portability, i.e., the target objective illustrated in the list 3.1.1. The primary goal of LLAMA is to provide a complete abstraction layer for memory layouts, enabling users to define custom layouts for their data types and seamlessly switch between them. At compile-time, it generates layout-specific access code that can be optimized away by the compiler, yielding performance equivalent to hand-written low-level loops. This design builds on concepts from earlier works such as ASX [64], Cabana [60], and SoAx [40], but formalizes the definition of `Record`, `View`, and `Mapping` more strictly.

LLAMA distinguishes between the array and the record dimensions. The array dimensions are defined at runtime, whereas the record dimension is defined at compile time. This allows to make the problem size itself a runtime value, but leaves the compiler room to optimize the data access [37].

A `Record` in LLAMA can be defined through empty struct definitions and templates as shown in listing 3.7.

Listing 3.7: User syntax with LLAMA

```
struct Id{}; struct Pos{};
struct Mass{}; struct Flags{};

using Particle = llama::Record<
    llama::Field<Id, uint16>,
    llama::Field<Pos, Vec>,
    llama::Field<Mass, double>,
    llama::Field<Flags, bool[3]>
>;
```

It provides a tag and the data type as compile-time information. In C++, the `llama::Record` is equivalent to the code in listing 3.8.

Listing 3.8: Equivalent struct

```
struct ParticleCpp {
    uint16 id;
    Vec pos;
    double mass;
    bool flags[3];
};
```

The data access design represents a crucial point for LLAMA: we call an access to a data structure which returns a memory location and the type stored at this location, a terminal access. A non-terminal access on a data structure does not yet result in

an element of the data structure, but a subpart, an intermediate object [37]. LLAMA uses intermediate object representations, called `VirtualRecord`, to avoid partial address computation and to preserve generality for the specific memory layout choice. The lifetime of a `VirtualRecord` is dependent on the `View` it refers to. An example of utilization can be found listing 3.9.

Listing 3.9: VirtualRecord obejct

```
// VirtualRecord utilization
auto view = llama::allocView(...);
llama::VirtualRecord<...> particle = view(1, 2, 3); // non - terminal
    operation
llama::VirtualRecord<...> pos = particle(Pos{}); // non - terminal
    operation
float& y = pos(Y{}) ; terminal operation
```

The template arguments of `VirtualRecords`, which could be avoided using `auto`, encapsulate information about record dimension, mapping policy, storage, and record coordinate.

On the other hand, it is possible to perform a deepcopy of a `Record` type into a `llama::One<RecordType>` object, as shown in listing 3.10.

Listing 3.10: Function deepcopy

```
// deepcopy using One
llama::One<Particle> p = view(3, 2, 1); // view independent deepcopy
p(Pos{}) (Y{}) = 1.0;
auto p2 = p; // another independent deep - copy
view(1, 2, 3) = p2; // copy into view
```

Lastly, the concept of `Mapping` is crucial in LLAMA, since it converts logical indexing to a physical memory address, according to the chosen layout. LLAMA implements several memory layouts within the library. Apart from the traditional ones, i.e., AoS, SoA, and AoSoA, One (deepcopy of the `Record`), Split (different parts of the same record can be mapped differently), Trace (count memory access for each field to trace memory access pattern), and Heatmap (builds a heatmap of the memory usage, counting memory access for memory units, i.e., bytes) are also available in this library.

The power of LLAMA is its flexibility. The library lets the developer swap among AoS, SoA, AoSoA, profiling mappings like Trace/Heatmap, and custom mappings by changing only the mapping type, and using the same syntax for every memory layout, avoiding code rewriting. Also, iterators on views of any dimension are supported and allow LLAMA to integrate with the C++ standard library [37], offering multiple ways to implement even

complex algorithms.

Zero overhead abstraction is achieved by exploiting the concept of terminal and non-terminal access and through the `VirtualRecord` class.

The `llama::View` is independent of any parallel programming framework, unlike Cabana (Kokkos-based) and other handwritten structures with custom allocators and C++ STL views like `std::span`(C++20) [25] or `std::mdspan`(C++23) [24]. In conclusion, LLAMA achieves portability not only across hardware backends but also across the programming frameworks that provide portability, i.e., Kokkos, SYCL, and Alpaka, since its abstraction layer decouples memory layout from execution. This makes it a more general and flexible solution compared to frameworks tied to a single parallel paradigm. Recent improvements that support compile-time size for the `llama::View` make this trivially copyable when launching GPU kernels [36].

The only caveat of LLAMA can be seen in the user syntax for accessing the data. Despite its flexibility, the way to access the data can make the code less readable when implementing complex code, and thus make the library more adapted to expert C++ developers than average users. The portability approach of LLAMA is an important reference for this thesis; however, the solution presented here focuses on improving usability and flexibility, providing similar portability with fewer layout variants and a more accessible interface.

3.1.2. Code generation for data structures definition

As it has been shown in the previous section, one of the main limitations of many data structures' frameworks is the lack of named field access: while the abstraction allows performance portability across memory layouts, users often lose the ability to access columns using the same field names that would naturally appear in a manually defined data structure. A common solution to this problem has historically been the use of code generation frameworks, like C/C++ preprocessor macros or scripting. With macros, it becomes possible to generate code for data structures, preserving symbolic field names and enabling a more user-friendly syntax. However, this comes at the cost of relying on the compiler's preprocessing step, which means handling symbols rather than true language constructs. As a consequence, macro-based approaches often result in complex and less maintainable backends, even if they provide a familiar user interface.

SoAx [40] represents one of the earliest successful attempts to provide an efficient yet user-friendly abstraction for handling particle data structures in HPC. The main motivation behind this library is indeed to use a SoA layout to store and compute data both for CPUs and GPUs, maintaining a suitable user interface and providing features to cover several

HPC use case scenarios. To achieve this, a combination of template metaprogramming, inheritance, and C++ macros has been used.

The user interface to declare and instantiate the SoA is the one proposed in listing 3.11.

Listing 3.11: User defintion of SoAX

```
// Define particle properties through macro
SOAX ATTRIBUTE(id, 'N'); // identity
SOAX ATTRIBUTE(pos, 'P'); // position
SOAX ATTRIBUTE(vel, 'V'); // velocity
SOAX ATTRIBUTE(mass, 'M') ; // mass
// Specify types and dimensions and
// concatenate attributes using std::tuple
typedef std::tuple<id<int, 1>,
                   pos<double, 3>,
                   vel<double, 3>,
                   mass<float, 1>> ArrayTypes;
// create SoA for 42 particles
Soax<ArrayTypes> soax (42);
```

As can be seen, the first step passes by some C++ macros that generate the definition of the SoA fields and the utilities. The macro here is used to propagate the name of the column to the struct data members and the main data access methods. All the other data structure utilities are implemented through compile-time recursion using variadic templates.

The data access syntax works in the way illustrated in listing 3.12 [40].

Listing 3.12: User syntax with SoAX

```
auto particle = soax.get Element(7);
particle.id() = 42;
particle.pos(0) = 3.14;
soax.push_back(particle);
```

An interesting feature available in `SoAx` is the possibility for the user to define custom methods that act on all the columns of the structure. This can be done by defining a struct containing a fixed-name member function called `doIt`. An example shown in listing 3.13 [40].

Listing 3.13: Custom functions with SoAx

```cpp
struct SetToValue
{
    template <class T, class Type>
    static void doIt (T& t , Type value) {
        for (int i=0; i < t -> size() ; i++)
            t -> operator[](i) = value;
    }
};
```

This feature can connect properties of the columns or even determine a role for them, making the object itself more representative and coherent.

Code generation can be achieved in other ways, and two representative approaches are worth mentioning.

The first one is QIRAL [6], a domain-specific language designed for Lattice Quantum Chromodynamics (QCD). In QIRAL, the user writes the algorithm at a very high level, often using a LaTeX-like mathematical syntax, and the compiler automatically produces high-performance OpenMP C code from these sources. QIRAL is not a tool for data layout optimization, but the data structures are generated according to the specific input of the user (for example, dense or sparse matrices, CSR format, linked graph exc...).

A second and more suitable approach is Python-based code generation, often leveraging templating libraries such as Jinja2. Here, the user describes the structure of the data (fields, types, dimensions) in a simple Python dictionary or YAML/JSON schema, and a script expands this description into a full C++ header implementing the data structures and accessors. For example, considering the `.yaml` file in listing 3.14 and the `.jinja` file in listing 3.15, the code can be simply generated with the Python script shown in listing 3.16.

Listing 3.14: .yaml file describing the fields

```yaml
# fields.yaml
fields = [
    {"name": "x", "type": "float"},
    {"name": "y", "type": "float"},
    {"name": "mass", "type": "double"}
]
```

Listing 3.15: .jinja defining C++ code

```
struct Particle {
    {% for f in fields %}
        {{ f.type }} {{ f.name }};
    {% endfor %}
};
```

Listing 3.16: Script generating the actual code

```
import yaml, jinja2, sys
fields = yaml.safe_load(open("fields.yaml"))["fields"]
tpl = jinja2.Template(open("particle.hpp.jinja").read())
open("particle.hpp","w").write(tpl.render(fields=fields))
```

This process can potentially, with moderate effort, generate all the desired code for the data structure, at the cost of making the development process slower, less readable, and maintainable.

The last approach worth consideration is to act directly on the compiler, expanding some utilities in order to automatically generate temporary SoA views from existing AoS data structures, redirect data accesses inside the annotated regions to temporary SoA buffers, then copy the results back. This allows local performance gains from vectorization and data locality, while keeping the code as simple and readable as the original AoS [55] [56]. An example can be found in listing 3.17.

Listing 3.17: Compiler extension for code manipulation

```
struct Particle {
    double pos[3];
    double vel[3];
    bool updated;
};

void drift(Particle* particles, int size) {
    [[clang::soa_conversion_target(particles)]]
    [[clang::soa_conversion_size(size)]]
    [[clang::soa_conversion_inputs(pos, vel)]]
    [[clang::soa_conversion_outputs(pos, updated)]]
    for (int i = 0; i < size; ++i) {
        auto& p = particles[i];
        p.pos[0] += p.vel[0] * dt;
        p.updated = true;
    }
}
```

In the next section, the focus will be on C++ reflection as a promising direction to further advance these ideas.

3.2. Reflection-based SoAs

The most promising approach to developing efficient data structures for HPC applications is a work-in-progress C++ feature called `reflection`. Reflection is a much-studied concept in programming literature, and from the born of scientific computing and HPC, there have been attempts to provide fully templated code to achieve compile-time (de-)serialization of classes and methods [46] and to motivate the introduction of a C++ library for runtime reflection [31], emulating languages like Java and Python.

Since template metaprogramming techniques in C++ have evolved significantly during the last decade with utilities like `std::tuple`, `if constexpr` statement, and `concepts`, a proposal to ISO C++ has been opened in 2024 to integrate this feature starting from C++26 [28].

The library would introduce some new concepts to give access to class and function metadata:

- `std::meta::info`: an opaque constant-expression type that represents program entities (variables, functions, classes, namespaces, etc.) at compile time.
- the reflection operator `^`, which, when applied to a variable, produces a std::meta::info object representing the metadata properties of the instance.
- a set of `consteval` metafunctions and splicers (written as `[: ... :]`), allowing to query meta-objects (e.g. name, type, members) and generate the corresponding code elements back into the program during compilation.

Once you have a meta-object, you can use a standard set of consteval metafunctions to query its properties (like name, type, scope, members) or even derive new meta-objects. Together with splicers, this makes it possible to reconstruct grammatical elements from metadata, effectively turning reflection values into real code during compilation.

An easy example of reflection can be found in listing 3.18.

Listing 3.18: Example of reflection

```
constexpr auto r = ^int;
typename[:r:] x = 42;        // Same as: int x = 42;
typename[:^char:] c = '*';   // Same as: char c = '*';
```

An example of type aliasing based on compile-time conditions can be seen in listing 3.19. In this example, a compile-time comparison between byte sizes is performed and, based on that, the type of `MyType` is determined.

Listing 3.19: Compile-time conditional alias with reflection

```
using MyType = [:sizeof(int)<sizeof(long)? ^long : ^int:];  // Implicit
    "typename" prefix.
```

This library is especially useful for enabling seamless flexibility in switching between different data layouts. It is possible to define consteval functions to define custom data layouts. An example showing how to implement an SoA with the reflection and how to define an instance of this type [28] is illustrated in listing 3.20. A `consteval` function takes a C++ struct (e.g., `Point`) and automatically build either a SoA or an AoS by injecting `std::arrays` named after the original members, each with the requested size. It is possible by introspection of member names and types at compile time, so the layout is generated without runtime reflection or manual boilerplate (for example using a dictionary).

Listing 3.20: SoA implementation with reflection

```
#include <meta>
#include <array>

template <typename T, std::size_t N>
struct struct_of_arrays_impl;

// function evaluated compile-time building the SoA struct
consteval auto make_struct_of_arrays(std::meta::info type,
                                     std::meta::info N) -> std::meta::
                                        info {
  std::vector<std::meta::info> old_members = nonstatic_data_members_of(
      type);
  std::vector<std::meta::info> new_members = {};
  for (std::meta::info member : old_members) {
    auto type_array = substitute(^std::array, {type_of(member), N });
    auto mem_descr = data_member_spec(type_array, {.name = name_of(
        member)});
    new_members.push_back(mem_descr);
  }
  return std::meta::define_class(
    substitute(^struct_of_arrays_impl, {type, N}),
    new_members);
}

template <typename T, size_t N>
```

```
using struct_of_arrays = [: make_struct_of_arrays(^T, ^N) :];

// Defining the SoA
struct Point {
  float x;
  float y;
  float z;
};

using points = struct_of_arrays<Point, 30>;
// equivalent to:
// struct points {
//   std::array<float, 30> x;
//   std::array<float, 30> y;
//   std::array<float, 30> z;
// };
```

With this approach should be possible to switch the memory layout just by changing a compilation flag with minimal macro-conditional code (1 line of code), as shown in listing 3.21.

Listing 3.21: Switching data layout

```
#if defined(SOA)
    using Points = struct_of_arrays<Point, 30>;
#elif defined(AOS)
    using Points = array_of_structs<Point, 30>;
#else
    #error "Define one of: SOA, AOS"
#endif
```

A first academic paper about flexible and efficient data structures with reflection is a work in progress by Jolly Chen [13]. The effort required by the user in this implementation can be seen in listing 3.22.

Listing 3.22: SoA definition using reflection

```
struct Vec {
   double &x, &y, &z;
};

struct S {
   int &id;
   Vec v;
};
```

```cpp

using SoA = rmpp::AoS2SoA<S>;
SoA s;
```

Since the backend can directly implement name and type introspection, the injection of suitable data members and accessors is enough to allow the syntax shown in listing 3.23.

Listing 3.23: Data access with reflection

```cpp
// AoS Style Indexing
for (int i = 0; i < s.size(); i++) {
    std::cout << s[i] << std::endl;
}

// SoA Style Indexing
for (int i = 0; i < s.id.size(); i++) {
    std::cout << s.id[i] << std::endl;
}
for (int i = 0; i < s.v.x.size(); i++) {
    std::cout << s.v.x[i] << std::endl;
}
```

This library introduces a cutting-edge technique to achieve the goals in list 3.1.1, which should be the best approach when C++26 and maybe C++29 will be fully adopted. By enabling direct access to metadata such as names, types, and methods, it allows developers to write generic code that was previously impossible without cumbersome boilerplate. At the same time, the current proposals inevitably expose a rather verbose and heavy syntax, which might hinder everyday adoption. Further studies and community discussion will therefore be essential to introduce syntactic sugar and higher-level abstractions, making reflection not only a technically groundbreaking feature, but also a practical and elegant tool for real-world development.

Finally, it is worth noting that macros have also been explored as a tool for implementing reflection mechanisms in C++. For instance, some months ago, Wu [70] discussed their use in combination with templates to provide C++ reflection described in section 3.2. This highlights how macros, despite their limitations, continue to play a central role in areas where the language still lacks native support.

What has been done in CMSSW basically means implementing C++ reflection by hand, with obvious limitations due to the (still) static approach.

3.3. The generic CMSSW SoA

The CMSSW SoA, the topic of my thesis, has been part of the CMS reconstruction framework since July 2022 [10]. The main purpose of the package is to provide an efficient, flexible, and user-friendly data structure that can be further extended in the future. The key ingredients of this implementation are three: C++ macros to propagate column types and names, template metaprogramming to achieve zero-overhead abstraction, and Alpaka for performance portability. Details on the motivations behind the design and implementation of the data structure, which constitutes the starting point of the work proposed in this thesis, is presented in the following.

3.3.1. Challenges for efficient data structures in CMSSW

Designing suitable data structures for CMSSW is a complex task that intersects multiple areas, including high-performance computing, heterogeneous workflows, modern C++ standards, and usability for both physicists and developers. As the demands on data processing pipelines grow, particularly with the integration of machine learning (ML) and the increasing diversity of computing architectures, the requirements for data structures become more stringent. The key challenges that have been faced are:

1. **Efficient and Portable Columnar Data Structures**
   Supporting columnar layouts is essential for SIMD/vectorized operations and ML workloads. Achieving this efficiently across different hardware platforms (e.g., CPUs, GPUs) requires careful design and cross-language compatibility (CUDA, HIP, SYCL).

2. **Dynamic Allocation**
   Many physics workflows require dynamic sizing of data structures at runtime. Supporting dynamic allocation without incurring high performance costs, especially on GPUs, is a significant challenge.

3. **User-Friendly Interface**
   Despite internal complexity, the interface must remain accessible to non-expert users. A balance must be struck between performance and ease of use.

4. **Efficient Serialization with ROOT**
   Integration with ROOT I/O is mandatory. Columnar structures must serialize efficiently and without unnecessary duplication, even in the absence of native C++ reflection.

3.3.2. Definition

The strategy adopted in CMSSW to implement the SoAs leverages the library illustrated in section 2.2 to auto-generate data members, type-safe accessors, and auxiliary subclasses/traits. In practice, it uses a similar approach to SoAx, preserving a similar user-facing syntax, but introducing an AoS-like interface via `operator[]` for element-wise access and broadening the design to allow greater flexibility in choosing column types at definition time. In particular, the implementation supports integration with Eigen, a widely used C++ template library for linear algebra, allowing programmers to easily manipulate vector and matrix objects [38]. Based on this, the user can choose between three types of columns: scalars, columns, and Eigen columns. The first one is a single value shared for all the SoA, like a tag, the second represents a column of values, with the size specified at runtime, while the third one gives the possibility to define a column of Eigen objects, which can be thought of as an AoS-column within the SoA, with indexing access.

An SoA in CMSSW can be defined by writing code similar to the one shown in listing 3.24.

Listing 3.24: Basic SoA definition in CMSSW

```
GENERATE_SOA_LAYOUT(SoAPositionTemplate,
                    SOA_COLUMN(double, x),
                    SOA_COLUMN(double, y),
                    SOA_COLUMN(double, z),
                    SOA_EIGEN_COLUMN(Eigen::Vector3d, covariance_matrix)
                    SOA_SCALAR(int, id))
```

The generated SoA will have the data members written in listing 3.25.

Listing 3.25: Generated data members

```
private:
    std::byte* mem_;
    size_type elements_;
    size_type const scalar_ = 1;
    byte_size_type byteSize_;
    double* x_ = nullptr;
    double* y_ = nullptr;
    double* z_ = nullptr;
    size_type covariance_matrixElementsWithPadding_ = 0;
    Eigen::Vector3d::Scalar* covariance_matrix_ = nullptr;
    size_type covariance_matrixStride_ = 0;
    int* id_ = nullptr;
```

In particular, other than the raw pointers, which represent the columns, each data member

has a specific mean:

- `mem_` represents the byte in which the SoA allocation will take place;
- `elements_` is the number of elements;
- `scalar_` is a const-variable equal to 1 to distinguish scalar variables' size for ROOT serialization [57];
- `byteSize_` is the total number of bytes;
- `x_`, `y_`, `z_`, and `covariance_matrix_` are the four double pointers to the columns;
- `id_` is a pointer to the integer scalar value;
- `covariance_matrixElementsWithPadding_` and `covariance_matrixStride_` are helpers to handle alignment and padding for the Eigen columns.

The goal of this design is to provide a minimal and portable SoA representation, based on raw pointers and size information, while supporting three categories of columns: standard contiguous arrays, scalar fields shared across all elements, and Eigen-based vector or matrix columns enabling advanced numerical operations.

The implementation details are not presented in this thesis. The generation process exploits variadic and conditional macros with functions included in `boost/preprocessor.hpp`.

The data accessors, their const-version, the AoS-element proxies, and the metadata functions are generated in the same way.

3.3.3. Template metaprogramming

Template metaprogramming in CMSSW's SoAs can be seen as a "sugar layer" on top of the macro-based code generation provided by the Boost Preprocessor library. While macros handle repetitive structure and boilerplate, templates add type safety, physical memory configuration, and zero-cost abstractions that the compiler can optimize away.

When defining a SoA, the user specializes instances with type aliases that configure the layout and its associated view. These template parameters determine, for example, the alignment strategy, whether runtime checks are performed, and how pointers are qualified.

Consider again the SoA defined in listing 3.24. To define a valid instance of `SoAPosition`, it is necessary to specialize the class, as shown in listing 3.26.

Listing 3.26: Template specialization

```
using SoAPosition = SoAPositionTemplate<ALIGNMENT, ALIGN_CHECK>;
using SoAPositionView = SoAPosition::ViewTemplateFreeParams<
    OTHER_ALIGNMENT, OTHER_ALIGN_CHECK, RESTRICT_QUALIFY, RANGE_CHECK>;
using SoAPositionConstView = SoAPosition::ConstViewTemplateFreeParams<
    OTHER_ALIGNMENT, OTHER_ALIGN_CHECK, RESTRICT_QUALIFY, RANGE_CHECK>;
```

The template parameters involved in the example above are:

- ALIGNMENT: fixed byte size of a memory chunk, which will introduce padding at the end of each column. Usually 64 for CPU and 128 for GPU. Default value: 128.
- ALIGN_CHECK: boolean to enable a runtime check for the columns to be aligned during allocation. Default value: false.
- OTHER_ALIGNMENT, OTHER_ALIGN_CHECK: different parameters for the View.
- RESTRICT_QUALIFY: boolean to enable `__restrict__` attribute declaration for the pointer type when accessing memory address within a specific index. Default value: true.
- RANGE_CHECK: boolean to enable runtime range-checking when accessing elements through the View. Default value: true.

The default parameters allow the user to write clear aliasing, like the portion of code in listing 3.27.

Listing 3.27: Default template parameters

```
using SoAPosition = SoAPositionTemplate<>;
using SoAPositionView = SoAPosition::View;
using SoAPositionConstView = SoAPosition::ConstView;
```

The `View` or `ConstView` is built from the `Layout`, and it can be accessed in the way shown in listing 3.28.

Listing 3.28: Data access syntax

```
// number of elements
const std::size_t elements = 10;
// size in bytes
const std::size_t bufferSize = SoAPosition::computeDataSize(elements);
// memory buffer aligned according to the layout requirements
std::unique_ptr<std::byte, decltype(std::free) *> buffer{
```

```
    reinterpret_cast<std::byte *>(aligned_alloc(SoAPosition::alignment,
       bufferSize)), std::free};
// SoA layout
SoAPosition soa{buffer.get(), elements};

SoAPositionView view{soa};
SoAPositionConstView const_view{soa};
auto elem = view[0];
elem = {7, 11, 13, 17};
// Fill up
for (SoAPosition::View::size_type i = 1; i < view.metadata().size(); ++i
   ) {
    auto elemi = view[i];
    // Copy the const view
    elemi = const_view[i - 1];
    auto elemix = elemi.x();
    elemi.x() += elemi.y();
    elemi.y() += elemi.z();
    elemi.z() += 1.;
    elemi.covariance_matrix() += {elemix, elemix + 1., elemix + 2.};
}
view.id() = 42;
```

3.3.4. Achieve portability

Portability for CMSSW's SoA is achieved by introducing the concept of `PortableCollection`, implemented within a different Alpaka-dependent library. They act as thin, type-safe wrappers around SoA data layout, providing a uniform interface for data access and manipulation across heterogeneous platforms.

The `PortableCollections` perform a layout-agnostic memory allocation, based on the SoA Layout type and on the `alpaka::Device`. Memory can be allocated as pinned memory to optimize data transfer between the host and the device. In this case, an `alpaka::Queue` is needed to coordinate execution and synchronization. They provide `Views` for data access that can be passed to device kernels as lightweight Alpaka-compatible objects.

Since the environment is heterogeneous, we have different types of collections:

- `PortableHostCollection<T>`: host-only memory buffer, it allocates both pageable and pinned memory
- `PortableDeviceCollection<T, Device>`: device-only memory buffer, it allocates

both pageable and pinned memory for a specific `alpaka::Device`, taken as template argument

- `PortableCollection<T, Device>`: host-device memory buffer. It is an alias that represents either host or device collection, depending on the type of the `alpaka::Device` template argument.

An example of `PortableCollection` instantiation is shown in listing 3.29.

Listing 3.29: PortableCollection instantiation

```
// Run on the first available device
auto const& device = cms::alpakatools::devices<Platform>()[0];
Queue queue{device};

// Number of elements
const std::size_t elems = 100;

// Portable Collection
PortableCollection<SoAPosition, Device> positionCollection(elems, queue)
    ;

// Portable Collection View
SoAPositionView& positionCollectionView = positionCollection.view();

// Portable Collection ConstView
const SoAPositionConstView& positionCollectionConstView =
    positionCollection.const_view();
```

A kernel can be called simply by passing the `View` instantiated in listing 3.29, as shown in listing 3.30.

Listing 3.30: Kernel call

```
// fill up
auto blockSize = 256;
auto numberOfBlocks = cms::alpakatools::divide_up_by(elems, blockSize);

const auto workDiv = cms::alpakatools::make_workdiv<Acc1D>(
    numberOfBlocks, blockSize);

alpaka::exec<Acc1D>(queue, workDiv, FillSoA{}, positionCollectionView);

alpaka::wait(queue);
```

This approach enabled heterogeneous code in CMSSW, and it has brought significant improvements in reconstruction algorithms [3] [8].

4 | Proposed methodology

In this section, we present and discuss the evolution of SoAs in CMSSW over the past year. The progress spans both user experience, i.e., clearer APIs, custom methods, and new abstractions, and performance-focused features such as heterogeneous data movement, zero-copy data manipulation, and AoS/SoA transposition. Together, these changes improve flexibility and readability while delivering state-of-the-art efficiency across CPUs and GPUs. The data structure has been modernized, porting some components to C++20, and the user interface has been improved.

4.1. User experience and level of abstraction increasing

The majority of the changes presented in the section focus on introducing new improvements in maintainability, which represents the main limitation of the framework, and provide new features to simplify the usage of CMSSW SoA.

4.1.1. Simplifying SoA backend

The first versions of the backend, i.e., the low-level layer responsible for memory layout and column access presented in section 3.3.2, relied heavily on conditional macros built on top of the `Boost::PP` library to specialize behavior for different column kinds (e.g. `scalar`, `column`, `eigen`). While this worked, it came with familiar drawbacks: macro-heavy code is opaque to tooling, difficult to read, and brittle with respect to ordering and expansion context. Error messages also tend to be noisy and detached from the intent of the code.

Conditional code-generation with macros has been replaced with small, explicit C++ traits that encode the type of column with compile-time decisions via template specializations. The result is both clearer and easier to maintain: the rules live in a handful of header-only traits that can be unit-tested and reused, and call sites read like normal C++ instead of macros.

A typical macro-based implementation for computing per-column memory sizes looked like the one shown in listing 4.1. In the example, column-based conditional code generation is performed. Based on `VALUE_TYPE`, which is a symbol representing the type of the column (scalar, column, eigen), different code is generated. The C++ type and the name of the column are propagated as well in the form of symbols. The same concept can be implemented using template metaprogramming, as shown in listing 4.2. The padding is taken into account with a suitable helper function `alignSize`. The Eigen library itself manages the stride between Eigen internal columns. An appropriate function performing the same operation as the one shown in listing 4.1 has been specialized for different template parameters representing the various column types. With this approach, the memory offset is computed with the code written in listing 4.3.

Listing 4.1: Macro-based conditional statement

```
/**
 * Computation of the column or scalar size for SoA size computation
 */
// clang-format off
#define _ACCUMULATE_SOA_ELEMENT_IMPL(VALUE_TYPE, CPP_TYPE, NAME)                          \
  _SWITCH_ON_TYPE(VALUE_TYPE,                                                             \
      /* Scalar */                                                                        \
      _soa_impl_ret += cms::soa::alignSize(sizeof(CPP_TYPE), alignment);                  \
      ,                                                                                   \
      /* Column */                                                                        \
      _soa_impl_ret += cms::soa::alignSize(elements * sizeof(CPP_TYPE), alignment);       \
      ,                                                                                   \
      /* Eigen column */                                                                  \
      _soa_impl_ret += cms::soa::alignSize(elements * sizeof(CPP_TYPE::Scalar), alignment)  \
             * CPP_TYPE::RowsAtCompileTime * CPP_TYPE::ColsAtCompileTime;                 \
  )
// clang-format on

#define _ACCUMULATE_SOA_ELEMENT(R, DATA, TYPE_NAME) _ACCUMULATE_SOA_ELEMENT_IMPL TYPE_NAME
```

Listing 4.2: Template specializations

```
// Helper functions for accumulating column elements
template <typename ColumnType>
struct AccumulateColumnByteSizes;

template <typename T>
struct AccumulateColumnByteSizes<cms::soa::SoAParametersImpl<cms::soa
    ::SoAColumnType::scalar, T>> {
  cms::soa::byte_size_type operator()(cms::soa::size_type, cms::soa::
      byte_size_type alignment) const {
    return cms::soa::alignSize(sizeof(T), alignment);
  }
};
```

```cpp

template <typename T>
struct AccumulateColumnByteSizes<cms::soa::SoAParametersImpl<cms::soa
    ::SoAColumnType::column, T>> {
  cms::soa::byte_size_type operator()(cms::soa::size_type elements,
      cms::soa::byte_size_type alignment) const {
    return cms::soa::alignSize(elements * sizeof(T), alignment);
  }
};

template <typename T>
struct AccumulateColumnByteSizes<cms::soa::SoAParametersImpl<cms::soa
    ::SoAColumnType::eigen, T>> {
  cms::soa::byte_size_type operator()(cms::soa::size_type elements,
      cms::soa::byte_size_type alignment) const {
    return cms::soa::alignSize(elements * sizeof(typename T::Scalar),
        alignment) * T::RowsAtCompileTime *
            T::ColsAtCompileTime;
  }
};
```

Listing 4.3: Computation of offset through templated function

```cpp
#define _ACCUMULATE_SOA_ELEMENT_IMPL(VALUE_TYPE, CPP_TYPE, NAME, ARGS)
                                                              \
  _soa_impl_ret += cms::soa::detail::AccumulateColumnByteSizes<
    BOOST_PP_CAT(typename Metadata::ParametersTypeOf_, NAME)>{}(
    elements, alignment);
```

This template-based approach improves:

- **Readability:** Intent is expressed with ordinary C++ names and specializations;
- **Maintainability:** Adding a new column kind means adding one specialization, not threading macro statements through many call sites;
- **Debugging:** Compilers understand traits and produce precise errors.

4.1.2. User-defined methods

A key feature for making SoA structures practical is the ability to attach user-defined methods that operate on elements and columns. Existing solutions such as SoAx[40] offer this only with heavy metaprogramming and the rigid requirement that methods be named `doIt`, which limits flexibility and readability.

In this section, useful features for high-level usability with data structures are presented. In particular, the possibility to define custom methods acting directly on the data structure, without a deep knowledge of the underlying backend.

As an example, we chose to have a SoA representing particles through position and velocity in 3D. In this context, we are often interested in computing the Euclidean norm of both physical quantities: the position vector $\mathbf{p} = \{x, y, z\}$ and the velocity vector $\mathbf{v} = \{v_x, v_y, v_z\}$. By defining custom element methods directly inside the SoA, each element can provide a simple interface to evaluate $||\mathbf{p}||$ and $||\mathbf{v}||$ without exposing low-level implementation details. Once these norms are available, further operations such as normalizing the position and velocity columns with respect to their magnitudes will be performed.

In our approach, users can declare entire code blocks via macros and generate them directly either to the `element` struct or the `const_element`. In the example is shown in listing 4.4, a method that normalizes the position and velocity vectors has been implemented, making use of const methods those compute the norm of such vectors. These methods are directly injected into the code. Due to the simplicity of the operation, it is possible to declare the methods `constexpr` making the code portable.

Listing 4.4: Custom-defined methods for elements

```
GENERATE_SOA_LAYOUT(
        SoAPointCloud,
        SOA_COLUMN(float, x),
        SOA_COLUMN(float, y),
        SOA_COLUMN(float, z),
        SOA_COLUMN(float, v_x),
        SOA_COLUMN(float, v_y),
        SOA_COLUMN(float, v_z),

        SOA_ELEMENT_METHODS(
            constexpr void normalise() {
              float norm_position = square_norm_position();
              if (norm_position > 0.0f) {
                x() /= norm_position;
                y() /= norm_position;
                z() /= norm_position;
              };
              float norm_velocity = square_norm_velocity();
              if (norm_velocity > 0.0f) {
                v_x() /= norm_velocity;
                v_y() /= norm_velocity;
```

```
                v_z() /= norm_velocity;
              };
            }),

        SOA_CONST_ELEMENT_METHODS(
            constexpr float square_norm_position() const {
                return x() * x() + y() * y() + z() * z(); };

            constexpr float square_norm_velocity() const {
                return v_x() * v_x() + v_y() * v_y() + v_z() * v_z();
                   };),

        SOA_EIGEN_COLUMN(Eigen::Matrix<double, 6, 6>, covarianceMatrix),
        SOA_SCALAR(int, detectorType))
```

For instance, after preparing and building the object instances as described in listing 3.27, the methods to perform the squared norm of the vectors can be called simply as shown in listing 4.5. In the example, two arrays representing position and velocity are filled with the new values of normalized vectors.

Listing 4.5: Custom methods utilization

```
// arrays of norms
std::array<float, elements> position_norms;
std::array<float, elements> velocity_norms;

// Apply square norm functions
for (size_t i = 0; i < elements; i++) {
    position_norms[i] = const_view[i].square_norm_position();
    velocity_norms[i] = const_view[i].square_norm_velocity();
}
```

This bridges the gap between the natural AoS interface, where methods belong to objects, and the SoA backend, where elements are normally not accessible.

It simplifies syntax, enables semantically meaningful method names, and allows intuitive, high-level operations while retaining the performance benefits of SoA layouts.

In the same way, it can be convenient to define methods acting on the whole structures, performing mutable and const operations involving entire columns. For example, starting from listing 4.4, one would want to add a method that sorts all the particles with respect to their Euclidean norm, or compute the centroid of the system with respect to positions. An example is shown in listing 4.6, where, for simplicity, the whole SoA definition of listing 4.4 and the actual implementation have been avoided.

Listing 4.6: Custom-defined methods for columns

```
SOA_METHODS(
      void sort() {
         // sort the columns
      }
),
SOA_CONST_METHODS(
       std::array<float, 3> centroid() const {
          // return the centroid
       }
)
)
```

In fact, something important to notice is that the implementation strategy of these methods depends on the intent of the programmer. Since the CMSSW SoA's main purpose is to run on heterogeneous frameworks, the implementation should make use of the `alpaka` library [71], avoiding standard library functions and structures and enabling performance portability on different devices.

4.1.3. SoA blocks

The traditional definition of SoA says that each column has the same dimension. This restriction can make some problems hard or even impossible to implement in complex and heterogeneous frameworks like the CMSSW one. In particular, for applications such as association maps or graph-based data structures.

To achieve this, a new abstraction, `SoABlocks`, has been introduced to allow the definition of SoAs with columns of different sizes. The implementation relies on the Boost::PP macro library, mirroring the approach used for the traditional SoA layout. SoABlocks integrate seamlessly with existing `PortableCollections`, while preserving performance with traditional SoAs. Users can access blocks through their assigned names and define custom methods as shown in listing 4.6 to extend functionality, for example, implementing a suitable `operator[]` to retrieve information elegantly.

As a use case example, let's consider the problem of representing a graph using a SoABlocks. A graph $G = (V, E)$ is a mathematical structure defined by a set of nodes (or vertices) V and a set of edges $E \subseteq V \times V$, where $|V| \leq |E|$ in the general case. Each edge represents a connection between two nodes, and depending on the context, the graph can be directed or undirected. Within the SoA framework, such a structure can be naturally represented using SoABlocks, where one layout describes the nodes and another layout

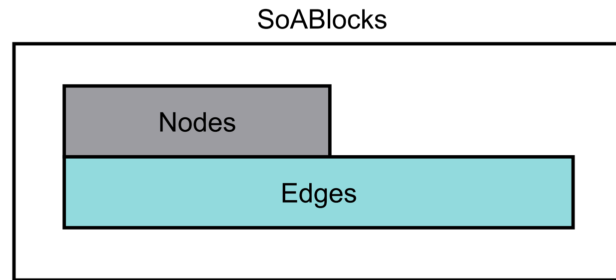

Figure 4.1: Graph represented as a Structure of Arrays. Nodes and edges are organized in separate blocks.

describes the edges as shown in listing 4.7 and fig. 4.1. This separation allows each block to store specific attributes, for example, weight and state for the nodes, or direction and flux for the edges, while maintaining different sizes for the corresponding columns. For example, the node block may contain identifiers and properties of vertices, while the edge block stores pairs of indices and associated weights. By combining these blocks into a single SoABlocks object, it becomes possible to represent and manipulate complex graph data structures efficiently in heterogeneous computing environments.

Listing 4.7: SoA Graph

```
GENERATE_SOA_LAYOUT(SoAEdges, SOA_COLUMN(Edge, edges))
GENERATE_SOA_LAYOUT(SoANodes, SOA_COLUMN(Node, nodes))

GENERATE_SOA_BLOCKS(SoAGraphTemplate,
  SOA_BLOCK(edges, SoAEdges),
  SOA_BLOCK(nodes, SoANodes),
  SOA_METHODS(
       EdgeSet operator[](Node node) {
        EdgeSet edgeSet;
        int size = edges().edges().metadata().size;
        for (std::size_t i = 0; i < size; i++) {
            Edge edge = edges().edges()[i];
            if (edge.incident(node))
                edgeSet.add(edge);
        }
        return edgeSet;
    }
  ))
```

The association map is a data structure that, given an offset, returns a set of elements. More specifically, the span of elements from that offset to the next one. Indeed, it is

possible to have a minimal 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.

Using this new feature, it is possible to obtain such a map with a macro composition, as shown in listing 4.8.

Listing 4.8: SoA Association Map

```
GENERATE_SOA_LAYOUT(ContentBuffersLayout, SOA_COLUMN(uint32_t, indices))
GENERATE_SOA_LAYOUT(OffsetBufferLayout, SOA_COLUMN(uint32_t, offsets))

GENERATE_SOA_BLOCKS(SoAMap,
                    SOA_BLOCK(content, ContentBuffersLayout),
                    SOA_BLOCK(offsets, OffsetBufferLayout),
                    SOA_METHODS(std::span<std::size_t> operator[](std::
                       size_t i) {
                    std::size_t start = offsets()[i].offsets();
                    std::size_t end = (i + 1 < static_cast<std::size_t>(
                       offsets().metadata().size()))
                                        ? offsets()[i + 1].offsets()
                                        : indices().metadata().size();
                  return {indices().indices().data() + start, indices().
                     indices().data() + end};
                }),
```

In this way, we can have a more complex structure within a single SoA, and with custom attributes and properties between the data layouts.

The SoABlocks can be instantiated passing as many integers as the number of blocks in the structure, and data can be comfortably accessed using the names the user decided to assign to each column. An example following listing 4.8 can be seen in listing 4.9. Note that `bufferSize` is guaranteed to satisfy `bufferSize % alignment == 0`, as required by `aligned_alloc` and by the SoA layout construction, as shown in listing 4.2.

Listing 4.9: Data access syntax for SoABlocks

```
// array of elements
const std::array<std::size_t, SoABlocksMap::blocksNumber> sizes{{
   size_content, size_offsets }};
// size in bytes
const std::size_t bufferSize = SoABlocksMap::computeDataSize(sizes);
// memory buffer aligned according to the layout requirements
std::unique_ptr<std::byte, decltype(std::free) *> buffer{
```

```
7     reinterpret_cast<std::byte *>(aligned_alloc(SoABlocksMap::alignment,
          bufferSize)), std::free};
8 // SoABlocks layout
9 SoABlocksMap blocks{buffer.get(), sizes};
10
11 SoABlocksMapView blocksView{blocks};
12 SoABlocksMapConstView blocksConstView{blocks};
13
14 // Fill the blocks with some data
15 for (int i = 0; i < blocksView.content().metadata().size(); ++i) {
16     blocksView.content()[i].indices() = i;
17 }
18
19 for (int i = 0; i < blocksView.offsets().metadata().size(); ++i) {
20     blocksView.offsets()[i].offsets() = i * 5;
21 }
```

4.1.4. AoS wrapper

Many use cases show that an AoS can perform very well when combinatorial algorithms are involved. For instance, consider an algorithm that needs to evaluate relationships between elements, such as finding pairs or triplets that satisfy a geometric or logical relation. In a SoA, each attribute required for the comparison must be retrieved from a separate column, leading to unefficient memory access and waste of registers. This can introduce overhead in tight nested loops, reduce cache efficiency, and prevent vectorization. In contrast, an AoS layout provides all attributes of an element next to each other in memory, improving locality. For this reason, it is important to provide an AoS layout to the CMSSW data structure.

A new nested class of the layout, `AoSWrapper`, has been implemented. This object contains an array of elements as data members, and it mirrors the implementation and, in particular, the data access pattern of the SoA. Maintaining the AoS very similar to the SoA, it is possible to use the same Alpaka wrappers and, in general, the same syntax for both structures.

Since the AoS is a nested class of the SoA, the definition is the same as listing 3.24, and it is enough to specify the aliases for the `AoSWrapper` type, and, if needed, the ones for the `AoSWrapper::View`, as it is possible to see in listing 4.10.

Listing 4.10: Basic AoS definition in CMSSW

```
GENERATE_SOA_LAYOUT(SoAPositionTemplate,
                    SOA_COLUMN(double, x),
                    SOA_COLUMN(double, y),
                    SOA_COLUMN(double, z),
                    SOA_EIGEN_COLUMN(Eigen::Vector3d, covariance_matrix)
                    SOA_SCALAR(int, id))

using SoAPosition = SoAPositionTemplate<>;
using AoSPosition = SoAPosition::AoSWrapper;
using AoSPositionView = AoSPosition::View;
using AoSPositionConstView = AoSPosition::ConstView;
```

The allocation and usage are exactly the same as the SoA, and the design of the AoS has been thought to make that compatible with `PortableCollections`. As it is possible to notice in listing 4.11, the syntax is exactly the same as listing 3.29 and listing 3.30, except for the template parameter representing the data structure layout.

Listing 4.11: PortableCollection with AoS

```
// Run on the first available device
auto const& device = cms::alpakatools::devices<Platform>()[0];
Queue queue{device};

// Number of elements
const std::size_t elems = 100;

// Portable Collection
PortableCollection<AoSPosition, Device> positionCollection(elems, queue)
    ;

// Portable Collection View
AoSPositionView& positionCollectionView = positionCollection.view();

// Portable Collection ConstView
const AoSPositionConstView& positionCollectionConstView =
    positionCollection.const_view();

// fill up
auto blockSize = 256;
auto numberOfBlocks = cms::alpakatools::divide_up_by(elems, blockSize);

const auto workDiv = cms::alpakatools::make_workdiv<Acc1D>(
    numberOfBlocks, blockSize);
```

```
alpaka::exec<Acc1D>(queue, workDiv, FillAoS{}, positionCollectionView);

alpaka::wait(queue);
```

In this way, it is possible to switch memory layout, for example, for performance analysis, with zero code rewriting.

4.2. Performance-oriented features

The features presented in this section can potentially improve SoA performance within CMSSW. In particular, these developments focus on improving data composition, heterogeneous memory operations, integration with machine learning frameworks, and layout conversion between SoA and AoS.

4.2.1. Generic SoA composition

Given a certain data flow, it can be useful to select specific columns between different SoAs and wrap them without copying the data. This requires extending the concept of `View` introduced in section 2.5, deleting the assumption that the accessors to data must point to a unique memory blob.

Suppose you have the following SoAs, one representing position and one representing statistical information, as shown in listing 4.12.

Listing 4.12: Two layouts to combine

```
GENERATE_SOA_LAYOUT(SoAPositionTemplate,
                    SOA_COLUMN(double, x),
                    SOA_COLUMN(double, y),
                    SOA_COLUMN(double, z),
                    SOA_SCALAR(int, detectorType))

using SoAPosition = SoAPositionTemplate<>;
using SoAPositionView = SoAPosition::View;
using SoAPositionConstView = SoAPosition::ConstView;

GENERATE_SOA_LAYOUT(SoAPCATemplate,
                    SOA_COLUMN(float, eigenvector_1),
                    SOA_COLUMN(float, eigenvector_2),
                    SOA_COLUMN(float, eigenvector_3),
                    SOA_EIGEN_COLUMN(Eigen::Vector3d, direction))

```

```
using SoAPCA = SoAPCATemplate<>;
using SoAPCAView = SoAPCA::View;
using SoAPCAConstView = SoAPCA::ConstView;
```

Supposing that some calculations using these data structures have been performed in some previous CMSSW modules, for example filling `SoAPosition` and retrieving statistical information from the coordinates, the user would consider a third SoA and build its `View` from any of the `SoAPosition` and `SoAPCA` columns. Then, a `deepCopy` function is provided to consolidate the `View` in a physical layout.

The first step is to declare the desired SoA Layout struct, as shown in listing 4.13.

Listing 4.13: Generic SoA

```
GENERATE_SOA_LAYOUT(GenericSoATemplate,
                    SOA_COLUMN(double, x),
                    SOA_COLUMN(double, y),
                    SOA_COLUMN(double, z),
                    SOA_EIGEN_COLUMN(Eigen::Vector3d, direction))

using GenericSoA = GenericSoATemplate<>;
using GenericSoAView = GenericSoA::View;
using GenericSoAConstView = GenericSoA::ConstView;
```

The objective is to implement a function for the SoA backend, whose full expansion in C++ would have the form illustrated in listing 4.14.

Listing 4.14: Generic SoA

```
struct FlexibleSoA {
    int elements;
    double *x_flex , *y_flex , *z_flex;
    Eigen::Vector3d :: Scalar *covariance_matrix_flex;

    FlexibleSoA (int n) : elements{n} {
        // allocation of the arrays one after the other
    }

    void attach_to (Position& pos, PCA& pca) {
        elements = pos.elements;
        if (elements != pca.elements)
        throw std::runtime_error ("Mismatch: Position and PCA have
            different number of elements");
        x_flex = pos.x;
        y_flex = pos.y;
```

```
17          z_flex = pos.z;
18          covariance_matrix_flex = pca.covariance_matrix;
19      }
20  };
```

A logic approach to implement this feature is to add a constructor to the `View`. Since a static code generation technique is used (see section 2.2), the feature has to respect the following rules:

- the SoA that will play the role of being "flexible" must have as data members the exact number and C++ types that are needed for the role (the constructor will fill all the data members).
- the number of elements of the involved `Views` must be the same. If that is not the case, a run-time exception will be thrown.
- the stride for Eigen columns has to be the same. If that is not the case, a run-time exception will be thrown. This scenario has to be taken into account because the involved SoAs can have the same number of elements, but different alignments.
- the constructor parameters have to represent the size of each column and the pointers to link.

To perform runtime size checks for every column, a new subclass of the `View` has been introduced, called `Metarecords`. This class has the same data members as the `View`, but the accessors to data return a tuple composed of a templated object representing the column, the corresponding size, and, for Eigen columns, the stride. For this reason, `Metarecords` should not have other purposes if not this.

The user syntax to build a `View` from different SoA's columns passes from accessing the a `Metarecords` nested class through the `records()` function, as shown in listing 4.15.

Listing 4.15: Generic SoA View building

```
1 auto positionRecords = position_view.records();
2 auto pcaRecords = pca_view.records();
3 // Building the View from different memory blobs
4 GenericSoA::View genericView(positionRecords.x(),
5                              positionRecords.y(),
6                              positionRecords.z(),
7                              pcaRecords.direction());
```

The same operation can be performed for `ConstViews`.

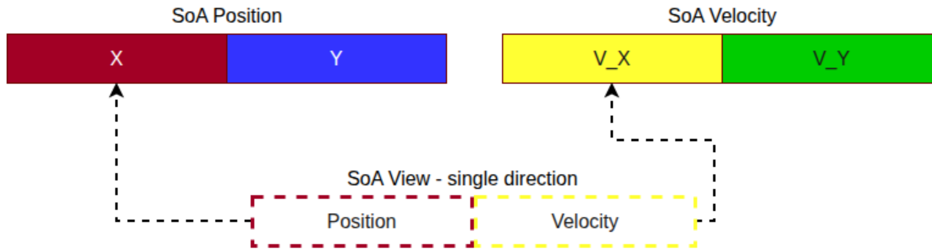

Figure 4.2: Building a generic View from x-direction columns for position and velocity

It can be useful now to consolidate this `View` in a physical layout. To do so, it is necessary to copy the data. In the example written in listing 4.16, the `deepCopy()` function acts on a `PortableCollection` previously built on top of the layout shown in listing 4.13, in a similar way of listing 3.29.

Listing 4.16: Generic SoA composition

```
genericSoA.deepCopy(genericView);
```

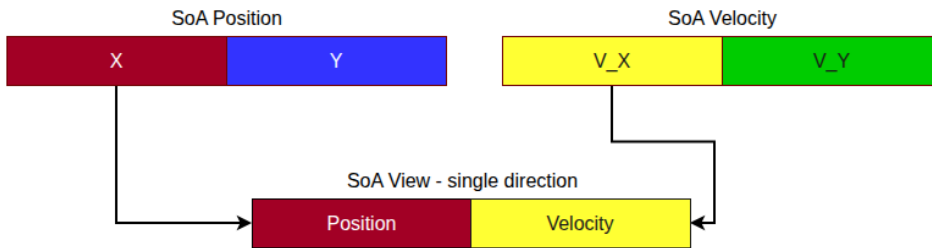

Figure 4.3: deepCopy the x-direction columns for position and velocity

4.2.2. Heterogeneous data transfer

Since the feature described in section 4.2.1 builds a non-owning `View` referring to multiple memory areas, it is necessary to provide an efficient way to copy the non-contiguous columns into a unique blob. As said in chapter 3, the actual user interface for SoAs to achieve performance portability is represented by the `PortableCollection`, so the copies will be performed with Alpaka. This leads to the same concept illustrated in fig. 4.3, but the behavior is backend-agnostic, i.e., the memory can be located in all possible devices (CPUs, GPUs etc.).

Assuming all the considered SoAs to be allocated in the same device, building the generic `View` can be done with the same syntax as shown in listing 4.15.

On the other hand, the `deepCopy` method has to make use of the `alpaka::memcpy` function [2], with the associated `queue`.

Since the design choice of CMSSW's SoAs is to separate the macro-generated SoA backend from Alpaka, and let the allocation to the `PortableCollections`, a template-based interface to loop over the SoA columns without using names is needed.

Following the idea implemented in Cabana [60], as shown in listing 3.4, the concept of `Descriptor` has been introduced, together with its const version. This object has to describe the properties of the structure that are needed to perform the heterogeneous `deepCopy`. In particular, it has to provide:

- the number of columns;
- the pointers to the data;
- the size of each column;
- a container that supports compile-time loop over the columns.

These challenges have been achieved through using a tuple of spans (available in the C++ STL as `std::tuple<std::span<T>...` [26]), and a static variable representing the number of columns, easily obtained during code generation.

Then, a compile-time loop using recursion and variadic templates has been implemented and illustrated in listing 4.17.

Listing 4.17: Heterogeneous deepCopy

```
1    // Copy column by column heterogeneously for host/device to host/
       device data transfer.
2    template <typename TQueue>
3    void deepCopy(ConstView const& view, TQueue& queue) {
4      ConstDescriptor desc{view};
5      Descriptor desc_{view_};
6      _deepCopy<0>(desc_, desc, queue);
7    }
8
9  private:
10   // Helper function implementing the recursive deep copy
11   template <int I, typename TQueue>
12   void _deepCopy(Descriptor& dest, ConstDescriptor const& src, TQueue&
       queue) {
13     if constexpr (I < ConstDescriptor::num_cols) {
14       assert(std::get<I>(dest.buff).size_bytes() == std::get<I>(src.buff
           ).size_bytes());
15       alpaka::memcpy(
16           queue,
```

```
            alpaka::createView(alpaka::getDev(queue), std::get<I>(dest.
                buff).data(), std::get<I>(dest.buff).size()),
            alpaka::createView(alpaka::getDev(queue), std::get<I>(src.buff
                ).data(), std::get<I>(src.buff).size()));
        _deepCopy<I + 1>(dest, src, queue);
    }
}
```

The `deepCopy` method covers every type of data transfer, thanks to the power of Alpaka, as already said in section 2.3.

The user syntax is very similar to the one for host-only data, but it needs the `alpaka::Queue`, as shown in section 4.2.2.

```
genericSoA.deepCopy(genericView, queue);
```

4.2.3. Generic SoABlocks composition

The concept of generic composition can be extended from individual SoAs to SoABlocks. As described in section section 4.2.1, a generic `View` can be built from existing memory regions belonging to different SoAs, provided that their metadata (sizes, strides, and alignments) are consistent. In a similar spirit, one can construct a SoABlocks object starting from existing SoAs, where each block represents a separate layout.

This feature, exactly as the one described in section 4.2.1, can potentially avoid inefficient data copies and make the SoABlocks coherent with the SoAs in terms of possible operations that can be performed.

Assuming to have three different SoAs, one representing particles' positions, and one representing particles' velocity, as shown in listing 4.18, we want to build the View of a SoABlocks from these existing SoAs. The first step is to define the interface that will reference the existing Views, i.e., a SoABlocks with the suitable types, as shown in listing 4.19. Just to notice, it is important to pass to the blocks the fully templated type of the Layout. In this way, it is possible to force the alignment to be the same for all the internal SoAs representing blocks since, as explained in section 3.3.3, the first template parameter contains the alignment. Default value, as for the traditional SoAs, is 128 bytes.

Listing 4.18: Two SoAs' examples

```
GENERATE_SOA_LAYOUT(PositionSoATemplate,
                    SOA_COLUMN(double, x),
                    SOA_COLUMN(double, y))

```

```
using PositionSoA = PositionSoATemplate<>;
using PositionSoAView = PositionSoA::View;

GENERATE_SOA_LAYOUT(VelocitySoATemplate,
                    SOA_COLUMN(double, v_x),
                    SOA_COLUMN(double, v_y))

using VelocitySoA = VelocitySoATemplate<>;
using VelocitySoAView = VelocitySoA::View;
```

Listing 4.19: SoABlocks

```
GENERATE_SOA_BLOCKS(SoAGenericBlocksTemplate,
                    SOA_BLOCK(position, PositionSoATemplate),
                    SOA_BLOCK(velocity, VelocitySoATemplate))
```

Composing a SoABlocks from existing SoAs mirrors the process of building a SoA from existing columns. The new constructor for the `BlocksView` does not include any particular constraint, but the number of elements for each SoA must be the same, and the constructor's number of parameters has to match the total number of blocks. Indeed, it must not be possible to leave any empty block in the struct. As one can see in fig. 4.4, two pre-existing SoAs are wrapped within a SoABlocks structure, which takes the references to those.

The syntax is simply the one shown in listing 4.20.

Listing 4.20: SoABlocks composition by different SoAs

```
    SoAGenericBlocksView view{positionView, velocityView};
```

Then, as it is possible to do with traditional SoAs, it is possible to copy the data using Alpaka and create a new memory blob that contains all the data. Consolidating the data can be useful for several reasons:

- It is possible to ensure the contiguity of the data;
- It is possible to send to a kernel a single `View` instead of multiple ones;
- One single heterogeneous class can represent many different things thanks to the custom methods, as explained in section 4.1.3.

The syntax is the same as shown in section 4.2.2; passing the `View` built so far in listing 4.20 to the `deepCopy` method.

The fig. 4.5 illustrates this concept.

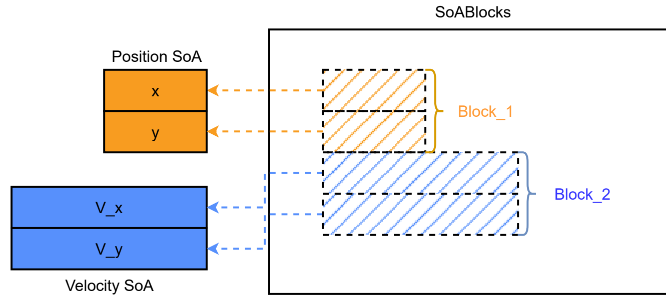

Figure 4.4: The View of a SoABlocks points to two different SoA Views.

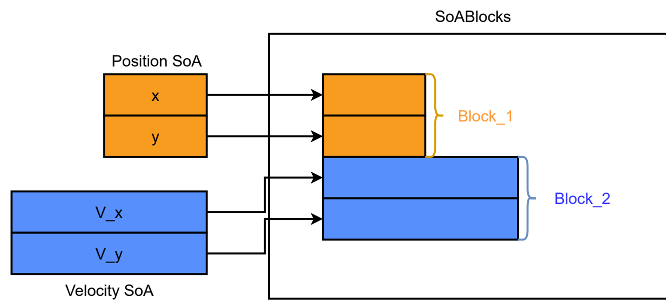

Figure 4.5: The SoABlocks is built by copying different SoAs.

4.2.4. Heterogeneous ML inference in CMSSW

Efficient machine learning in GPU-based workflows requires overcoming the bottleneck of unnecessary data movement between GPU memory and tensor formats. To address that, an interface in CMSSW that converts SoA data directly into PyTorch tensors has been implemented, eliminating explicit data transfers. This interface leverages the `Metarecords` struct described in section 4.2.1, to compute the correct shapes and strides for each column, and the `torch::from_blob()` function [54] to obtain zero-copy views in GPU memory. The core problem is that Python ML inference frameworks (e.g., PyTorch) require tensors with precise data types (e.g., `float32`, `int64`), well-defined memory layouts (contiguous or explicitly strided), and specific data readout. By using `Metarecords` to expose per-column metadata, we map SoA data format directly into tensors in place, and write model outputs back without staging. Since the view created with `torch::from_blob()` is non-owning, the lifetime of the underlying buffer must be guaranteed for the entire duration of the tensor use; on GPU, this further requires that the buffer resides on the same device and that the tensor type matches the column type to avoid undefined behaviour. In fig. 4.6, the logic flow is shown, highlighting the key concept: the ML model is now portable within the CMS software framework, and the two expensive device-to-host and

host-to-device data transfers are avoided through seamless data format conversion.

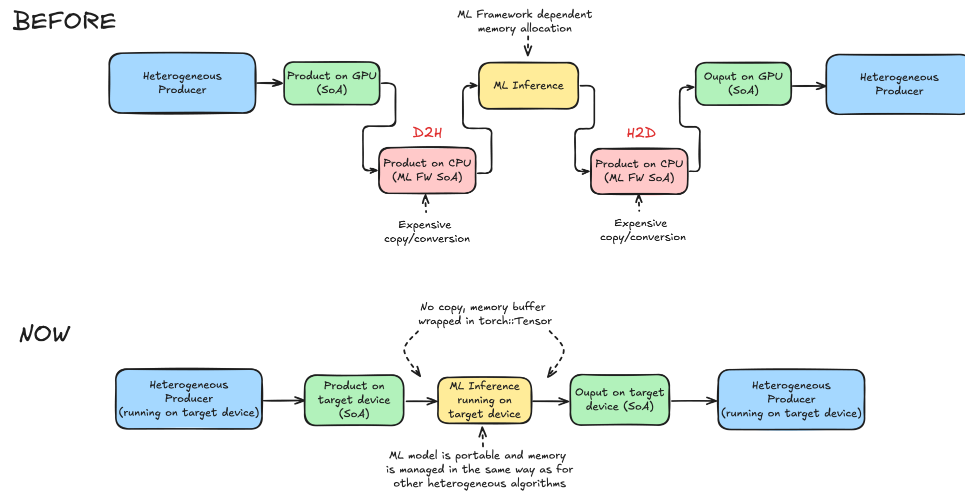

Figure 4.6: Machine Learning framework before and after the heterogeneous inference using SoAs

The interface can handle single columns with their metadata, distinguishing between scalars, columns, and eigen columns at compile-time. As we can see in listing 4.21, both ahead-of-time (AOT) and just-in-time (JIT) compilation are supported via a compile-time policy, e.g. `Model<CompilationType::kAheadOfTime>` or `Model<CompilationType::kJustInTime>` while the SoA to Tensor conversion, zero-copy interface remains unchanged. With AOT, the model is exported as an artifact and loaded as a shared library during execution [53], with fixed data types, no warm-up, yielding low latency, and strong reproducibility at the cost of reduced flexibility. JIT compiles on the first invocation, adapting to runtime variables, but introduces an initial compilation overhead, typical of the Python programming language.

Listing 4.21: SoA-Tensor converter

```
using namespace cms::torch::alpaka;

// structs
PortableCollection<SoAInputs, Device> inputs_device(batch_size,
    alpaka_device);
PortableCollection<SoAOutputs, Device> outputs_device(batch_size,
    alpaka_device);

// instantiate model
auto model = Model<CompilationType::kJustInTime>(m_path);
model.to(queue);

```

```
// metadata for automatic tensor conversion
auto input_records = inputs_device.view().records();
auto output_records = outputs_device.view().records();
SoAMetadata<SoAInputs> inputs_metadata(batch_size);
inputs_metadata.append_block("features", input_records.x(),
    input_records.y(), input_records.z());
SoAMetadata<SoAOutputs> outputs_metadata(batch_size);
outputs_metadata.append_block("preds", output_records.m(),
    output_records.n());
ModelMetadata<SoAInputs, SoAOutputs> metadata(inputs_metadata,
    outputs_metadata);

// inference
model.forward(metadata);
```

4.2.5. Data structure conversion

An important feature of the framework is the ability to convert data from SoA to AoS format or vice versa. This capability is particularly useful in heterogeneous workflows, where different stages of computation may require different data layouts for optimal performance. For instance, a dataset may initially be organized as an SoA, but before executing a clustering algorithm, where data access is often irregular or not coalesced, it may be beneficial to convert it to an AoS.

The design of this feature consists of a host function called by the Layout-dependent `PortableCollection`, called `transpose()`, that takes as arguments the `PortableCollection` wrapping the opposite layout, and an `alpaka::Queue` to schedule the operation.

The conversion function is performed via a kernel, in which each thread assigns one AoS element to the corresponding SoA value (or viceversa). This mechanism can be applied in heterogeneous environments, providing flexibility and performance portability across various computational scenarios, in addition to the speedup due to the GPU implementation.

The syntax is shown in listing 4.22. The accelerator type is passed as a template argument, and how and in which hardware the operations will be performed.

Listing 4.22: Conversion function

```
    // Transpose the data from SoA to AoS
    aosCollection.transpose<Acc1D>(soaCollection, queue);
    alpaka::wait(queue);
```

The policy to handle scalars within the data structures is to put them at the end of the memory buffer.

5 | Results

In this chapter, some practical and experimental results from the work presented in this thesis are shown. The features involving usability and new abstraction integrations are presented, showing that zero-overhead is preserved while the level of abstraction increases. The performance-oriented extensions are evaluated, highlighting the average reduction in execution time.

5.1. Increased the level of abstraction

The level of abstraction refers to how far the implementation details of a data structure are hidden from the user, allowing them to interact with it through a simpler and more intuitive interface. Increasing the level of abstraction is crucial because it improves usability, reduces the need for low-level knowledge of the backend, and enables developers to focus on the logic of their applications rather than technical details.

Experimental results, both for CPU and GPU, demonstrate that there is no additional overhead in using these methods, implementing them with appropriate free functions, or placing complex code directly in the kernels.

5.1.1. Custom methods for SoA elements and View

Two new macros were introduced to enable the direct injection of user-defined functions within the SoA backend, allowing them to act on individual elements. Both host and device methods are supported, including templates and const variants, making this feature highly suitable for heterogeneous computing environments.

As an experiment, the syntax shown in listing 4.4 for the functions computing the position and velocity norms is compared with two free functions doing the same thing, as shown in listing 5.1.

Listing 5.1: Free function computing norm

```
float norm_position(SoAPointCloud::ConstView::const_element point) const
    {
    return sqrt(point.x() * point.x() + point.y() * point.y() + point.z
        () * point.z());
}

double norm_velocity(SoAPointCloud::ConstView::const_element point)
    const {
    return sqrt(point.v_x() * point.v_x() + point.v_y() * point.v_y() +
        point.v_z() * point.v_z());
}
```

Then, instead of calling the functions as shown in listing 4.5, the syntax would be the one in listing 5.2.

Listing 5.2: Free function utilization for elements

```
// arrays of norms
std::array<float, elements> position_norms;
std::array<double, elements> velocity_norms;

// Apply square norm functions
for (size_t i = 0; i < elements; i++) {
    position_norms[i] = square_norm_position(const_view[i]));
    velocity_norms[i] = square_norm_velocity(const_view[i]));
}
```

It is immediately noticeable that encapsulating the functionality as a class method makes the syntax clean, since the function conceptually operates on the inside of an object [49]. Therefore, in the case of listing 5.1, the user must know the name of the nested class `const_element`, which increases the complexity of writing such methods.

The same concept is valid for writing custom methods for the `View`, as shown in listing 4.6. Indeed, in the case of listing 4.7, an `operator []` has been defined to return, given a certain node, the edges associated with that node. That will help make the graph-SoA a complete object with its own attributes. Assuming to have the same SoABlocks definition of listing 4.7, and a free function called `getEdges(Node node)` that does the same as the `operator []`, the difference in how you perform the same operation is illustrated in listing 5.3. The advantage in readability and usability of the `operator []` is evident.

Listing 5.3: Free function utilization for the view

```
// free function
auto nodes = graph.view().nodes_view();
EdgeSet associated_edges = getEdges(nodes[2]);

// operator []
auto graphView = graph.view();
EdgeSet associated_edges = graphView[2];
```

5.1.2. SoA Blocks

To evaluate the impact on efficiency of using the SoABlocks instead of multiple traditional SoAs, two applications with identical instructions are implemented. Both traverse three views (`positionView`, `velocityView`, `algebraView`) and perform the same sequence of *read*/*write* operations, fused multiply–adds (FMAs), and trigonometric functions. Any performance difference, therefore, stems from memory management and accessor overhead in the underlying SoABlocks rather than from divergent operations.

Work is distributed uniformly over the element range $[0, N)$, where N is the size of the smallest SoA (`algebraView`). For each element index i, the routine executes a fixed per-thread workflow. First, there are three *reads* from `positionView` to load $p_x = x(i)$, $p_y = y(i)$, and $p_z = z(i)$. Second, three velocity components are seeded from these positions by simple multiplications, $v_x = 0.5\,p_x$, $v_y = 0.5\,p_y$, $v_z = 0.5\,p_z$. Third, three small accelerations are initialized as affine functions of the positions, $a_x = 10^{-3} + 10^{-6}p_x$, $a_y = 2 \cdot 10^{-3} + 10^{-6}p_y$, $a_z = 3 \cdot 10^{-3} + 10^{-6}p_z$. With constants `iterations` $= 100$, `dt` $= 10^{-3}$, and `interval` $= 10^{-2}$, a loop of fixed length follows. At each iteration, the routine performs

- three FMAs updating the velocities, $v \leftarrow a \cdot \mathtt{dt} + v$ (one FMA per component);
- trigonometric and elementary functions to introduce non-linearity, computing $s = \sin(v_x) + \cos(v_y)$ and $r = \sqrt{v_z^2 + 1}$ (one sin, one cos, one sqrt);
- three FMAs updating the accelerations, $a_x \leftarrow s \cdot 10^{-3} + a_x$, $a_y \leftarrow r \cdot 10^{-3} + a_y$, $a_z \leftarrow (s + r) \cdot 10^{-3} + a_z$.

The instruction count per element is thus deterministic and identical across the two implementations: 6 FMAs per iteration (600 FMAs per element with `iterations` $= 100$), one additional multiplication for v_z^2, three additions to form s, $v_z^2 + 1$, and $s + r$, and three meth functions evaluations per iteration.

After the loop, the routine performs structured *writes* to the output. The three velocity components (v_x, v_y, v_z) are written back to `velocityView`. The `algebraView` receives three scalars computed by division, $e_1 = v_x/$`interval`, $e_2 = v_y/$`interval`, $e_3 = v_z/$`interval`, and the candidate direction is stored component-wise as (a_x, a_y, a_z). This results, per element, in three *reads* (positions) and nine *writes* (three velocity components, three eigenvector components, three direction components). Assuming 32-bit `float`, the per-element data movement is approximately 12 B read and 36 B written ($\sim$ 48 B total), while arithmetic work includes 600 FMAs ($\sim$ 1200 floating-point operations if counted as one multiplication plus one addition per FMA), 100 evaluations each of sin, cos, and sqrt, plus the small number of standalone multiplications and additions mentioned above.

The access pattern is intentionally regular: contiguous *reads* of x, y, z followed by coalesced *writes* to velocity and algebra fields, with no cross-element dependencies and a fixed loop trip count. Observed differences in timing or scaling can therefore be attributed to backend choices such as address arithmetic, alignment policy, padding of scalar tails, and the cost of accessors.

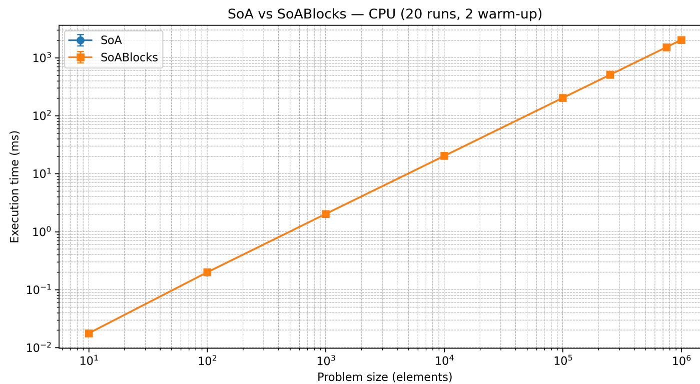

Figure 5.1: SoABlocks vs traditional SoAs in CPU

Two different plots, fig. 5.1 and fig. 5.2, show the comparison of performance between SoABlocks and traditional SoA, respectively, on CPU and NVIDIA L4 GPU.

As shown, performances are exactly the same, proving the correctness of the result and encouraging the use of SoABlocks for writing more flexible, coherent, and portable code.

5.1.3. AoS wrapper

A performance overview of the AoS data structures currently developed for CMSSW, measured in isolation to highlight their intrinsic memory access characteristics, is presented

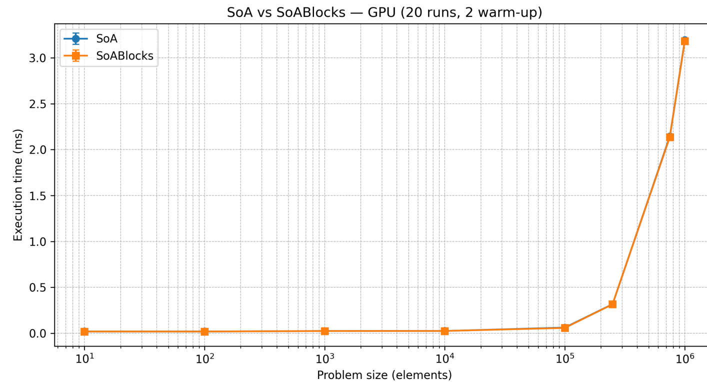

Figure 5.2: SoABlocks vs traditional SoAs in GPU

in this section.

The performance overview indicates that the newly introduced CMSSW AoS data structure performs well. Indeed, when compared against the CMSSW SoA implementation, which is already fully optimized and known to match the efficiency of a carefully hand-written SoA, results show that AoS remains competitive across an evaluated compute-bound scenario (the same presented in section 5.1.2), as shown in fig. 5.3 and fig. 5.4. This confirms that the CMSSW AoS representation is a robust choice when algorithms primarily operate locally on individual objects, without stressing memory bandwidth or requiring large-scale columnar access.

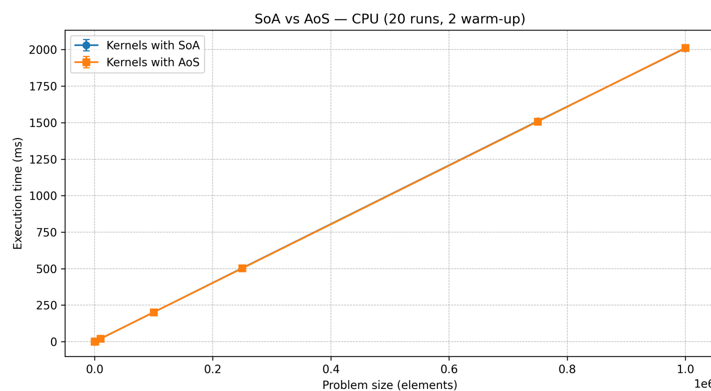

Figure 5.3: AoS performance in CPU

5.1.4. Optimize data access

As related work, in this section, an update to a modern data access pattern is presented. In particular, when someone wants to send single columns to a GPU kernel, these were expressed as raw pointers. As a consequence, the runtime check for out-of-bounds memory

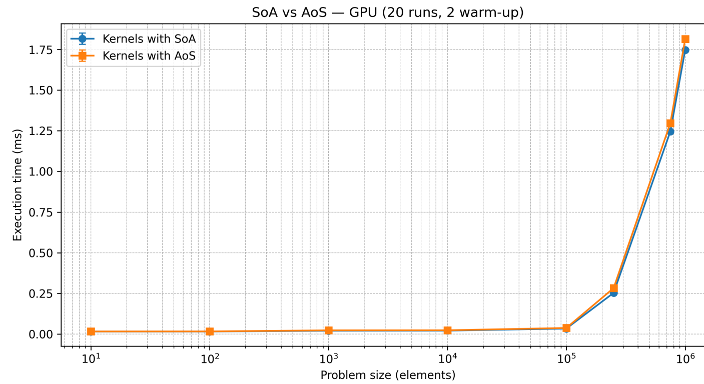

Figure 5.4: AoS performance in GPU

access must be implemented manually in the data structure backend, which may introduce performance issues and error proneness.

Starting from C++20, the concept of `std::span`, consisting of a pointer and a size, performs automatic range-checking [25]. To be more specific, the column accessor shown in listing 5.4 can result in either a raw pointer or a `std::span`. Since `std::span` is nothing but a pointer and a size, it is compatible with GPU programming and suitable for portable frameworks.

Listing 5.4: SoA column accessor

```
soaView.x();
```

Using the same computational benchmark of section 5.1.2, zero-overhead in using `std::span` over raw pointers for both CPU and GPU has been proven, as shown in fig. 5.5 and fig. 5.6. This behavior was not obvious, since `std::span` brings a pointer and a size for each column, which could cause overuse of registers when passing single columns to a kernel.

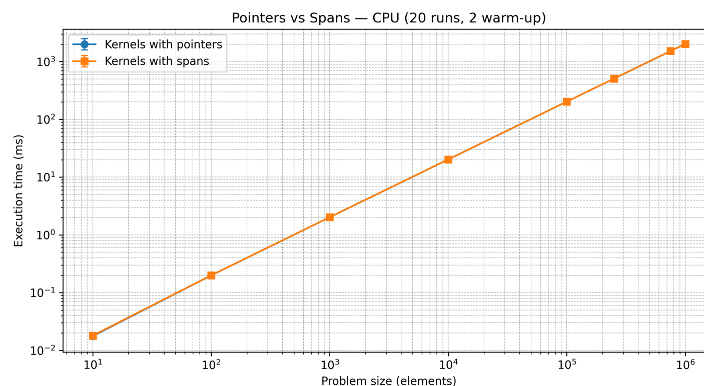

Figure 5.5: Pointers vs Spans in CPU

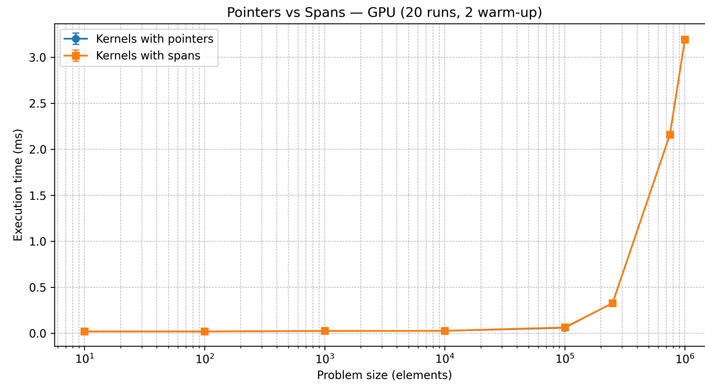

Figure 5.6: Pointers vs Spans in GPU

5.2. Analyze performance improvements

This section presents some benchmarks designed to evaluate the performance impact of the new features introduced in the data structure framework. Each test focuses on a specific aspect, from zero-copy data access and heterogeneous transfers to machine learning inference and layout conversion, to validate that the new abstractions preserve or even enhance performance across both CPU and GPU backends.

5.2.1. Generic View

The possibility of selecting useful columns between different SoAs, without performing copies, and with an easy-to-use API, can open up new and important options when porting algorithms to GPUs, where data transfer is usually a bottleneck.

To quantify the potential advantages of Generic Views, two different experiments have been carried out.

In the baseline approach, three independent SoAs are passed to the kernel described in section 5.1.2, together with their full `Views`. The kernel then accesses only a small subset of the fields that are actually needed for the computation.

The same benchmark was then run using a Generic View that wraps only the columns of interest and is passed as a single argument to the kernel. Since the Generic View is essentially a bundle of pointers to existing device memory (with zero copying or reshuffling of data), the execution time is expected to match the baseline.

As shown in fig. 5.7 and fig. 5.8, the results confirm this expectation: no measurable execution overhead is introduced by using the Generic View instead of the original SoAs.

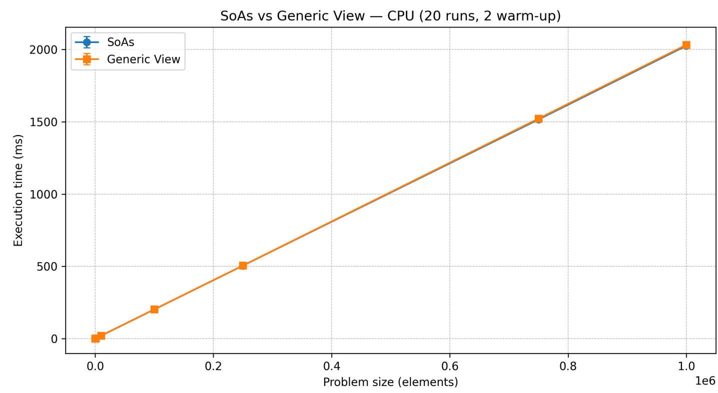

Figure 5.7: Three SoAs vs Generic View in CPU

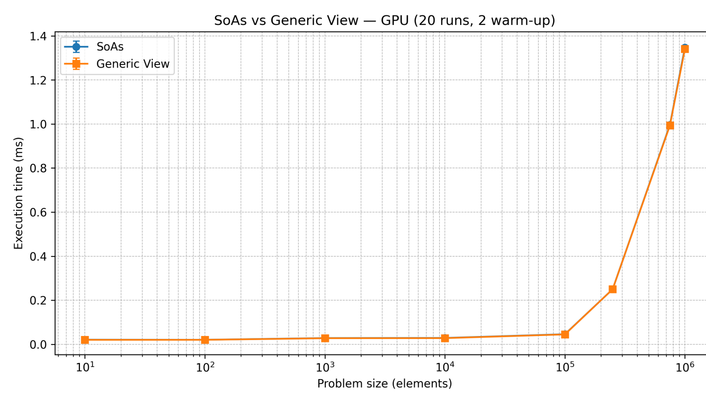

Figure 5.8: Three SoAs vs Generic View in GPU

This means the Generic View has no impact on performance compared to a traditional View. Reducing the number of parameters and eliminating unused fields simplifies the kernel signature, making the code more maintainable and more readable. In more complex kernels with larger parameter packs, this can even improve register occupation on GPUs.

A second experiment considers the alternative scenario in which the user explicitly copies the relevant columns into a new SoA and passes that new structure for the kernels.

Such a strategy requires copying the data column by column into a new `PortableCollection`. On GPUs, this is extremely expensive: every column copy introduces additional memory transfers across the PCI bus and launches separate device-side copy operations.

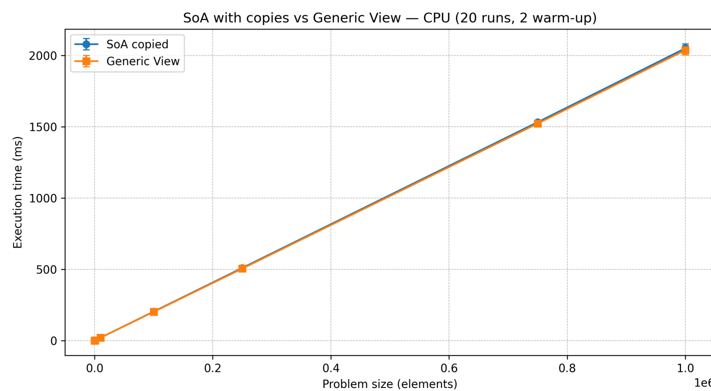

Figure 5.9: Copy the data vs Generic View in CPU

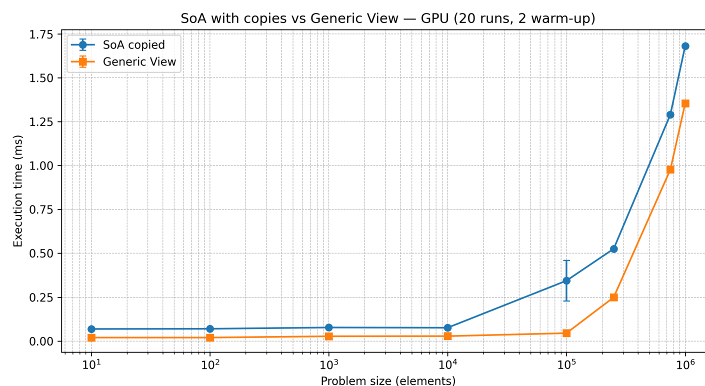

Figure 5.10: Copy the data vs Generic View in GPU

In this case, the Generic View shows a clear advantage: since it avoids data copying, the additional transfer cost is completely removed. This makes the Generic View a preferable solution whenever the selected columns are already on the device, as is typically the case in large workflows. Performance results in fig. 5.9 and fig. 5.10 show that using Generic View instead of copying the data in the GPU can achieve an execution time reduction

of about 50% when considering a million elements in the SoAs. In CPU, the difference is almost nonexistent, since serial execution makes the computation very expensive, and the data transfer is negligible with respect to that.

5.2.2. Heterogeneoeus data transfer

While the Generic View enables zero-copy access to selected data across different SoAs, there are situations in which a physically contiguous and self-contained memory layout is required. Typical examples include serializing the data for persistence, sharing structures across heterogeneous frameworks, or for compatibility with certain algorithms. For these use cases, a `deepCopy()` API is provided, which allows transferring the currently referenced columns into a new `PortableCollection`, filling its device with the columns of the desired View.

To validate this design, a comparison was made between the `deepCopy()` operation and a manually written sequence of `alpaka::memcpy` operations. The benchmarks in fig. 5.11 and fig. 5.12 show that there is no measurable performance difference between the two strategies, confirming no significant overhead for the function calling. This is an expected result, since `deepCopy()` internally translates into the same low-level copy operations that a user would write by hand.

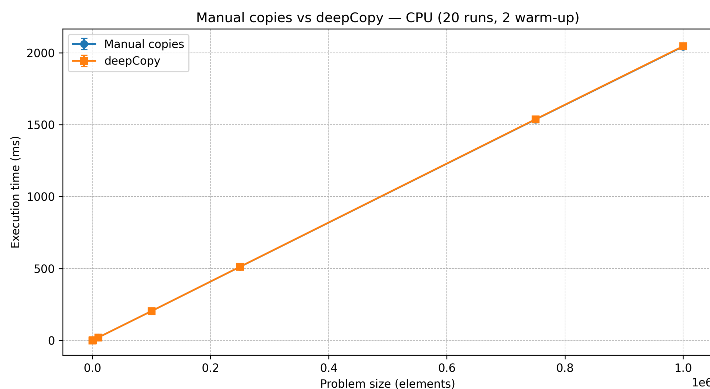

Figure 5.11: Manual copies vs deepCopy in CPU

However, the manual approach has two evident disadvantages. First, the code required to perform multiple column-wise copies becomes significantly longer and more error-prone, particularly when struct layouts evolve. Second, the user must be familiar with Alpaka view creation, data extents, and device semantics, making the approach less accessible to non-experts and harder to maintain. It is possible to see an example with 5 columns in listing 5.5, and it is possible to compare with the single line shown in section 4.2.2. The `deepCopy()` abstraction, in contrast, provides a concise and reliable interface that

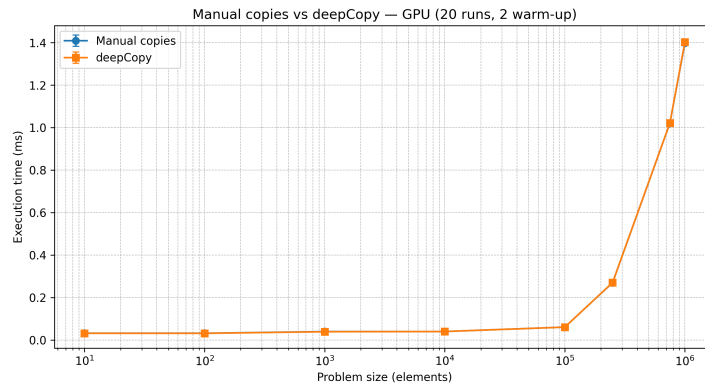

Figure 5.12: Manual copies vs deepCopy in GPU

remains valid even if internal implementations change.

Listing 5.5: Manual data copies

```
alpaka::memcpy(queue,
    alpaka::createView(device, genericCollectionView.x().data(), size),
    alpaka::createView(device, positionCollectionView.x().data(), size))
    ;
alpaka::memcpy(queue,
    alpaka::createView(device, genericCollectionView.y().data(), size),
    alpaka::createView(device, positionCollectionView.y().data(), size))
    ;
alpaka::memcpy(queue,
    alpaka::createView(device, genericCollectionView.vx().data(), size),
    alpaka::createView(device, velocityCollectionView.vx().data(), size)
    );
alpaka::memcpy(queue,
    alpaka::createView(device, genericCollectionView.vy().data(), size),
    alpaka::createView(device, velocityCollectionView.vy().data(), size)
    );
alpaka::memcpy(queue,
    alpaka::createView(device, genericCollectionView.candidateDirection
    ().data(), cms::soa::alignSize(size * sizeof(double), SoA::
    alignment) * 3 / sizeof(double)),
    alpaka::createView(device, pcaCollectionView.candidateDirection().
    data(), cms::soa::alignSize(size * sizeof(double), SoA::alignment
    ) * 3 / sizeof(double)));
```

5.2.3. Improvement of ML inference

As said in section 4.2.4, the standard approach, consisting of running a heterogeneous model inference copying the data back to the host, has been replaced with a seamless wrapping using `torch::from_blob()`.

An experiment to show the improvements in terms of execution time has been implemented and tested. 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. Each node and edge carries a set of numerical features describing local properties (such as energy, position, or shape). The network processes the graph through several convolutional layers, where each node updates its state by exchanging information with its connected neighbours. Finally, a link predictor estimates whether pairs of nodes belong to the same particle, effectively merging fragments into reconstructed objects [43].

The fig. 5.13 shows the typical improvement achieved when useless data transfer between host and device is avoided. In particular, a 25x speedup is observed for 700 particles.

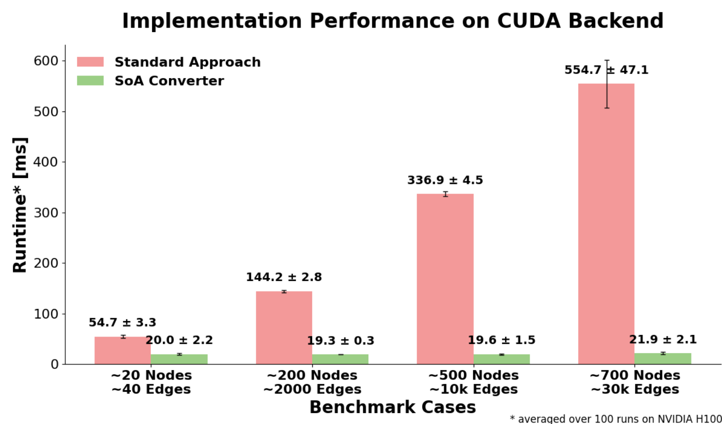

Figure 5.13: SoA converter vs Standard approach

AoT, where optimized kernels are generated ahead of execution, is preferable for production deployments with stable input schemas. In contrast, JIT is better suited for research and development where model or input evolution is expected.

AoT avoids dynamic allocations and extra memory traffic. Since data types are well-known, the compiler can enable proper optimizations (tiling, unrolling, ideal layouts, and inlining). And with near-zero runtime overhead, AoT keeps per-inference latency low even after warm-up, so the advantage persists as models get larger, as shown in fig. 5.14.

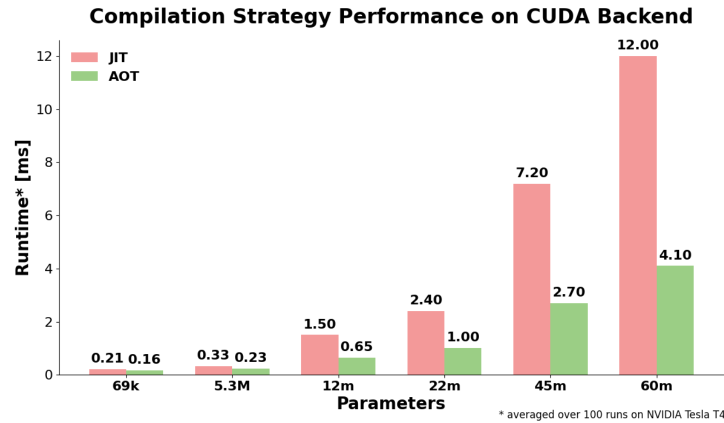

Figure 5.14: JIT vs AOT

5.2.4. Data structure conversion

In this section, the cost of converting between AoS and SoA layouts (and viceversa) is evaluated and presented using the conversion function developed in this thesis.

A structure composed of six fields (one scalar, four columns, and one Eigen column) is used, and the time required to convert from AoS to SoA and from SoA to AoS is measured.

The fig. 5.15 and fig. 5.16 report execution time for converting from AoS to SoA and from SoA to AoS for both CPU and GPU.

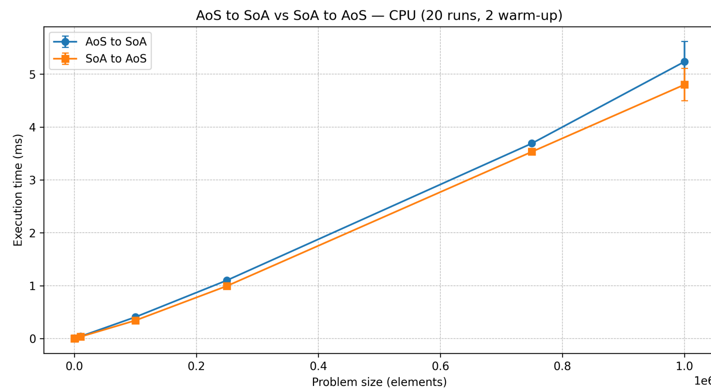

Figure 5.15: Data structure conversion in CPU

The conversion scales linearly with the number of elements, confirming that the implementation does not introduce overheads or synchronization bottlenecks. On the CPU, the two directions exhibit very similar performance, with SoA to AoS being slightly faster. This is consistent with the expected memory-access pattern: converting into an AoS means writing contiguous structs in memory, which is favorable for the CPU store pipeline.

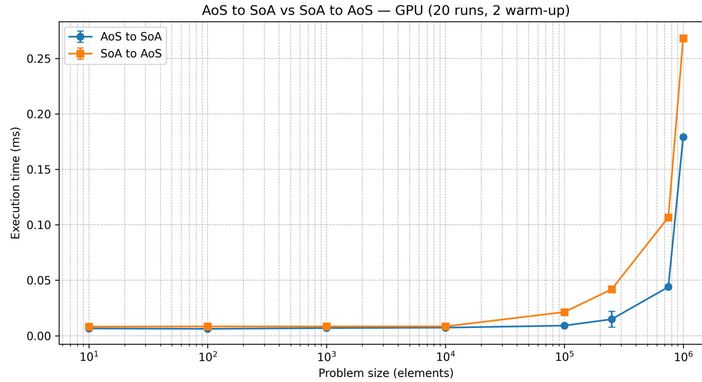

Figure 5.16: Data structure conversion in GPU

On the GPU, the AoS to SoA conversion is significantly faster than the inverse operation, especially at larger sizes. This can be explained by the fact that AoS to SoA writes follow a perfectly coalesced pattern: each thread writes its own index into consecutive elements of each column array, which maps optimally to the GPU memory subsystem. In contrast, SoA to AoS requires each thread to write a full struct, resulting in strided and partially non-coalesced stores.

5.3. Comparison between different SoA implementations

In parallel to the development of the SoA framework presented in this thesis, a dedicated benchmarking repository has been created to study and compare the performance and usability of different data structures implementation strategies [11]. This repository has been designed as a common research environment to serve as a reference platform for ongoing studies in the HPC community for data structure design and implementation. An overview of the project was presented at the ACAT 2025 conference in Hamburg [12]. The repository is publicly available and continuously evolving, with continuous integration (CI) testing to ensure stability and to detect regressions in performance or functionality as new features are introduced.

The main objective of this work is to provide a unified framework to test different SoA techniques under identical conditions. To do this, a generic and homogeneous data access pattern has been implemented through a set of macros and helper abstractions that guarantee consistency among all the approaches. In this way, the performance results reflect purely the differences in the data structure implementations, and not variations in the usage model. Therefore, it is easy to integrate other implementations into the

framework and test them against the existing ones.

Four distinct SoA strategies have been evaluated:

- **Manual SoA and AoS implementation**: baseline, ad-hoc and manual written data structures.
- **Reflection-based SoA**: a structure leveraging the new experimental C++ feature (partially available starting from C++26).
- **Template Metaprogramming-based SoA**: a type-safe design where columns are encoded in template parameters and accessed through `constexpr` metadata.
- **CMSSW SoA**: the solution developed in this thesis, built upon the `PortableCollection` for performance portability, exploiting `Boost::PP` macro library for code generation and inlining techniques.

A concrete representation of how these four strategies respectively declare a pretty complex SoA (10 columns and 5 different C++ types) is shown in listing 5.6.

Listing 5.6: Different layout declarations

```
1  // Template metaprogramming SoA
2  template <template <class> class F>
3  struct S10 {
4      template <template <class> class F_new>
5      operator S10<F_new>() { return {x0, x1, x2, x3, x4, x5, x6, x7, x8,
         x9}; }
6      F<float> x0, x1;
7      F<double> x2, x3;
8      F<int> x4, x5;
9      F<Eigen::Vector3d> x6, x7;
10     F<Eigen::Matrix3d> x8, x9;
11 };
12
13 using SoAWrapper = wrapper::wrapper<S10, std::span, wrapper::layout::soa
     >;
14 using AoSWrapper = wrapper::wrapper<S10, std::span, wrapper::layout::aos
     >;
15
16 // Reflection SoA
17 struct S10 {
18     float &x0, &x1;
19     double &x2, &x3;
20     int &x4, &x5;
21     Eigen::Vector3d &x6, &x7;
```

```
22     Eigen::Matrix3d &x8, &x9;
23 };
24 using SoA = rmpp::SoA<S10>;
25 using AoS = rmpp::ApS<S10>;
26 
27 // CMSSW SoA
28 GENERATE_SOA_LAYOUT(CMSSWSoA,
29     SOA_COLUMN(float, x0),
30     SOA_COLUMN(float, x1),
31     SOA_COLUMN(double, x2),
32     SOA_COLUMN(double, x3),
33     SOA_COLUMN(int, x4),
34     SOA_COLUMN(int, x5),
35     SOA_EIGEN_COLUMN(Eigen::Vector3d, x6),
36     SOA_EIGEN_COLUMN(Eigen::Vector3d, x7),
37     SOA_EIGEN_COLUMN(Eigen::Matrix3d, x8),
38     SOA_EIGEN_COLUMN(Eigen::Matrix3d, x9))
39 
40 using CMSSWSoA = CMSSWSoA<>;
41 using CMSSWAoS = CMSSWSoA::AoSWrapper;
```

In addition to qualitative aspects such as readability, maintainability, and implementation complexity, an extensive performance benchmarking campaign has been conducted. A unified micro-benchmark structure has been defined, organizing tests into four representative categories:

1. **General benchmarks**, including:
   - Soft read/write
   - Soft compute
   - Real-case read/write (mixed access)
   - Strided CPU access
   - Hard read/write (worst cache behavior)
2. **N-Body algorithm**: memory-bound operations with irregular access.
3. **Stencil algorithm**: computation with nearest-neighbor operations.
4. **Invariant mass calculation**: typical HEP physics computation involving multiple columns.

The plots shown in fig. 5.17 and fig. 5.18 show performance results obtained on a CPU

backend. Both the N-Body and the Invariant Mass benchmarks (fig. 5.17 and fig. 5.18) exhibit almost the same performance for all tested variants (reflection-based implementation has not been included in the benchmarks since the compiler is only available for experimental trials, and it is not yet trustworthy). Since the workloads are dominated by arithmetic operations and irregular access patterns across multiple fields, both AoS and SoA formats achieve similar efficiency, and the overhead from abstraction is completely optimized away by the compiler, resulting in no statistically significant difference in execution time.

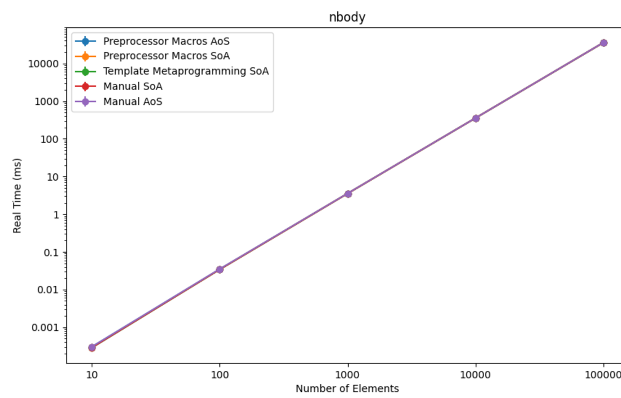

Figure 5.17: N-body comparison in CPU

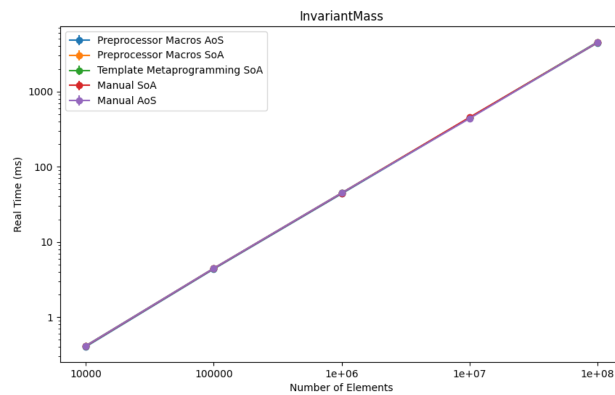

Figure 5.18: Invariant Mass comparison in CPU

6 | Conclusions and future developments

Efficient data layout design plays a central role in high–performance computing, particularly in complex heterogeneous environments where memory access patterns directly influence scalability. The distinction between Array of Structures (AoS) and Structure of Arrays (SoA) layouts is well known: SoA enables higher throughput on parallel architectures at the cost of reduced usability. A key objective of the present work has therefore been to provide a library that preserves the zero-overhead abstractions while offering users the best possible syntax.

In this thesis, a study on modern data structures for HPC has been conducted, with particular attention to SoAs. Beginning with an explanation of the main theoretical concepts of data structures and a survey of existing approaches to design and implementation, the work has presented the CMS Software framework SoA. The implementation addresses the needs of heterogeneous workflows in particle reconstruction. Afterward, the CMSSW SoA has been extensively improved in both functionality and internal design. Combining the preprocessing techniques with modern C++ template metaprogramming, the structure now supports a rich spectrum of features: a new API for ML heterogeneous inference, a new constructor for the View taking columns belonging to different SoAs, user-defined custom methods, direct AoS–SoA conversion, composition of multiple SoAs, and heterogeneous data transfer enabled by Alpaka-based backend integration. All the new features have been validated and evaluated within the CMSSW framework, using both simple but representative benchmarks and real CMS data.

Nevertheless, like almost every research-oriented work, it is essential to focus on future steps and further developments for the data structure.

The most relevant directions for future work are the following:

- **Reduce the dependency on macros using template metaprogramming:** as said in section 2.2, code generation using macros can be very powerful but also error-prone and not less readable and maintainable than modern C++ metaprogramming.

The goal is to reduce the use of macros to propagate only the column names, giving template objects the responsibility to deduce the column and the C++ type. The starting point to address this should be to replace the data members of the layout shown in listing 3.25 with objects wrapping this information.

- **Direct member access without operator():** instead of using the syntax shown in listing 6.1, it could be worth it to try to obtain the syntax in listing 6.2, i.e., avoiding the `operator ()` to access the column value.

Listing 6.1: Current data access syntax

```
// AoS-like syntax
soa[i].x();
// SoA-like syntax
soa.x()[i];
```

Listing 6.2: Desired data access syntax

```
// AoS-like syntax
soa[i].x;
// SoA-like syntax
soa.x[i];
```

This could improve code readability and make the data structure more intuitive to use.

- **Macro expansion tooling in CMSSW:** since the data structure backend is not easy to read because of the huge macro dependency, it should be useful to add a command to the building environment, so that a user could easily expand the pre-processor symbols and obtain a readable header. CMSSW relies on SCRAM [68] to configure, build, and manage its software environment in a consistent and reproducible way across different platforms. An idea could be to add an easy-to-use command like `scram preprocess <file_name>`.

- **ROOT serialization with RNTuple:** future integration with the new `ROOT::RNTuple` [50] feature for columnar data storage will be crucial to achieve better I/O performance for both SoA and AoS, introducing significant improvements on file sizes and read/write speed.

- **Introduction of AoSBlocks:** mirroring the concept existing for SoABlocks, an equivalent AoSBlocks abstraction is planned to allow lossless conversion between SoABlocks and AoSBlocks partitions, strengthening the duality between layouts. Currently, the transposition function is disabled for SoABlocks, and it is a clear case

of functionality and coherence that must be integrated within the implementation design. The composition of AoS layout types should implement the AoSBlocks.

- **Decoupling size from capacity:** at present, the logical size of an SoA coincides with the physically allocated capacity. However, in some cases, it is not possible to know exactly how much memory should be allocated. Currently, users keep track of this information using a suitable `SOA_SCALAR` called, for example, `size`, and store the actual valid content size of the structure. Support for elastic capacity as part of the data structure backend could make the code easier to write and understand, decrease error-proneness and potentially improve performance and memory utilization by allowing more flexible and adaptive data management.
- **MultiView abstraction:** a still under development feature is planned to allow concatenated access to multiple instances of the same layout type. Users would thus interact with various data structures as if they formed a single continuous bunch of data, making it possible for algorithms to operate transparently across several objects without needing to handle them individually. The development of this feature is linked to the previous one: concatenating multiple views requires being sure about the number of elements of the structure, in order to compute the exact offset and avoid reading or writing garbage data.

In parallel to the main CMSSW SoA development, the repository [11], which benchmarks different data structure implementations, also foresees several future improvements. The most relevant planned developments include adding standard deviation information to confidence interval plots to better represent performance variability, implementing and testing GPU kernel benchmarking (currently limited to CPU), and investigating observed performance mismatches across different implementations, particularly between the baseline and CMSSW versions. Moreover, the benchmarking framework should be extended to facilitate the integration and testing of new deployments with minimal effort. Finally, profiling should be enhanced by providing more meaningful and detailed insights using tools such as Intel VTune [32] and NVIDIA Nsight [33]

The steps outlined above already include the necessary steps to reach full maturity: a progressive transition toward C++ reflection, hybrid AoS/SoA support, and improved I/O integration. The SoA package is now not only a high-performance data container, but also a strategic component in future CMS software evolution.

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth Series. Addison-Wesley Professional, 12 2004. ISBN 978-0321227256. URL `http://www.staroceans.org/e-book/c-template-metaprogramming-concepts-tools-and-techniques-from-boost-and-beyond_compress.pdf`.

[2] alpaka community. alpaka - Reference. URL `https://alpaka.readthedocs.io/en/latest`.

[3] B. Alves, A. Bocci, M. Kortelainen, F. Pantaleo, and M. Rovere. Heterogeneous techniques for rescaling energy deposits in the cms phase-2 endcap calorimeter. *EPJ Web of Conferences*, 251:04017, 01 2021. doi: 10.1051/epjconf/202125104017.

[4] N. Andriotis, A. Bocci, E. Cano, L. Cappelli, A. Di Pilato, L. Ferragina, G. Hugo, M. J. Kortelainen, M. Kwok, J. J. Olivera Loyola, F. Pantaleo, A. Perego, W. Redjeb, M. Dewing, J. Esseiva, and the CMS Collaboration. Evaluating performance portability with the cms heterogeneous pixel reconstruction code. 295:11008, 2024. doi: 10.1051/epjconf/202429511008. URL `https://www.epj-conferences.org/articles/epjconf/abs/2024/05/epjconf_chep2024_11008/epjconf_chep2024_11008.html`.

[5] S. Balducci. Cluestering: a high-performance density-based clustering library for scientific computing, 2024.

[6] D. Barthou, G. Grosdidier, R. Dolbeau, O. Pene, and O. Brand-Foissac. Automated Code Generation for Lattice Quantum Chromodynamics and beyond. *25th Conference on Computational Physics*, 8 2014. URL `https://arxiv.org/pdf/1208.4035`.

[7] A. Bocci. Optimising the cms gpu reconstruction: scheduling, efficiency, stability. Slides for the talk at ACAT 2025, Hamburg, Germany, 2025. URL `https://indico.cern.ch/event/1488410/contributions/6606728/`

attachments/3132740/5557833/A.%20Bocci%20-%20Optimising%20the%20CMS%20GPU%20reconstruction_%20scheduling,%20efficiency,%20stability.pdf.

[8] A. Bocci, V. Innocente, M. Kortelainen, F. Pantaleo, and M. Rovere. Heterogeneous reconstruction of tracks and primary vertices with the cms pixel tracker. *Frontiers in big data*, 3:601728, 2020. ISSN 2624-909X. doi: 10.3389/fdata.2020.601728. URL https://europepmc.org/articles/PMC7931873.

[9] A. Bocci, A. Czirkos, A. D. Pilato, F. Pantaleo, G. Hugo, M. Kortelainen, W. Redjeb, and on behalf of the CMS collaboration. Performance portability for the cms reconstruction with alpaka. *Journal of Physics: Conference Series*, 2438(1):012058, feb 2023. doi: 10.1088/1742-6596/2438/1/012058. URL https://doi.org/10.1088/1742-6596/2438/1/012058.

[10] E. Cano. Implementation of generic SoA data structure in the CMS software. *21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research,*, 3 2023. URL https://cds.cern.ch/record/2872252.

[11] J. Chen, L. Beltrame, O. G. Rietmann, and M. Holzer. SoA-different-implementation. URL https://github.com/cern-nextgen/wp1.7-soa-benchmark.

[12] J. Chen, L. Beltrame, and O. G. Rietmann. A Unified Interface for Different Memory Layouts. In *Proceedings of ACAT 2025: 20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Hamburg, Germany, Sep 2025. URL https://indico.cern.ch/event/1488410/contributions/6562815/contribution.pdf. Oral contribution ID 115.

[13] J. Chen, A. L. Varbanescu, and A. 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. doi: 10.1145/3680256.3722203. URL http://dx.doi.org/10.1145/3680256.3722203.

[14] T. Childers, M. J. Kortelainen, M. Kwok, A. Strelchenko, and Y. Wang. Porting cms heterogeneous pixel reconstruction to kokkos, 2021. URL https://arxiv.org/abs/2104.06573.

[15] Clang contributors. Clang reference manual. URL https://releases.llvm.org/14.0.0/tools/clang/docs/ClangCommandLineReference.html#actions.

[16] CMSSW contributors. CMS software, . URL https://github.com/cms-sw/cmssw.

[17] CMSSW contributors. CMSSW Application Framework, . URL https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookCMSSWFramework.

[18] A. Collaboration. The alice experiment at the cern lhc. *Journal of Instrumentation*, 3(08):S08002, aug 2008. doi: 10.1088/1748-0221/3/08/S08002. URL https://doi.org/10.1088/1748-0221/3/08/S08002.

[19] A. Collaboration. The ATLAS Experiment at the CERN Large Hadron Collider. *Journal of Instrumentation*, 3(08):S08003, aug 2008. doi: 10.1088/1748-0221/3/08/S08003. URL https://doi.org/10.1088/1748-0221/3/08/S08003.

[20] C. Collaboration. The cms experiment at the cern lhc. *Journal of Instrumentation*, 3 (08):S08002, 2008. doi: 10.1088/1748-0221/3/08/S08004. URL https://cds.cern.ch/record/1129810.

[21] C. N. Collaboration. High-Luminosity LHC. URL https://home.cern/resources/faqs/high-luminosity-lhc.

[22] L. Collaboration. The lhcb detector at the lhc. *Journal of Instrumentation*, 3(08): S08005, 2008. doi: 10.1088/1748-0221/3/08/S08005. URL https://cds.cern.ch/record/1129809.

[23] B. community. Reference Boost Preprocessor Library for C/C++, . URL https://www.boost.org/doc/libs/1_89_0/libs/preprocessor/doc/index.html.

[24] C. community. C++ reference: mdspan declaration, . URL https://en.cppreference.com/w/cpp/container/mdspan.html.

[25] C. community. C++ reference: span declaration, . URL https://en.cppreference.com/w/cpp/container/span.html.

[26] C. community. C++ reference: tuple declaration, . URL https://en.cppreference.com/w/cpp/utility/tuple.html.

[27] C. community. C++ reference: union declaration, . URL https://en.cppreference.com/w/cpp/language/union.html.

[28] C. community. C++ reflection proposal, . URL https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2996r13.html.

[29] T. community. TechStriker. https://www.techstrikers.com/c-language/compilation-process-in-c.php, .

[30] K. constributors. Kokkos core libraries. URL https://github.com/kokkos/kokkos#readme.

[31] T. Devadithya, K. Chiu, and W. Lu. C++ Reflection for High Performance Problem Solving Environments. *SpringSim '07: Proceedings of the 2007 Spring Simulation Multiconference - Volume 1*, pages 435–440, 3 2007. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2058bb40e6b80504ba1084452fd9c126cd19f891.

[32] I. developers. Intel VTune profiler, . URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler-download.html.

[33] N. developers. NVIDIA Nsight System, . URL https://docs.nvidia.com/nsight-systems/index.html.

[34] L. Evans and P. Bryant. LHC Machine. *Journal of Instrumentation*, 3(08):S08001, aug 2008. doi: 10.1088/1748-0221/3/08/S08001. URL https://doi.org/10.1088/1748-0221/3/08/S08001.

[35] R. Fabian. *Data Oriented Design.* Self-published, 1 edition, 9 2018. ISBN 9781916478701. URL https://www.dataorienteddesign.com/.

[36] B. M. Gruber. Updates on the Low-Level Abstraction of Memory Access. 2 2023. URL https://arxiv.org/pdf/2302.08251v1.

[37] B. M. Gruber, G. Amadio, J. Blomer, A. Matthes, R. Widera, and M. Bussmann. LLAMA: The Low-Level Abstraction for Memory Access. 2 2022. URL https://arxiv.org/pdf/2106.04284.

[38] 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.

[39] A. Herten. Many cores, many models: Gpu programming model vs. vendor compatibility overview. page 1019–1026, 2023. doi: 10.1145/3624062.3624178. URL https://doi.org/10.1145/3624062.3624178.

[40] H. Homann and F. 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. URL https://m-sp.org/downloads/array2018.pdf.

[41] V. Innocenti. UltimateSOA: a trivial SoA binding to your beloved OO data hierarchy. URL https://twiki.cern.ch/twiki/bin/view/Main/VIUltimateSOA.

[42] *Intel 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation,

2023. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Section 11.6: Advanced Vector Extensions (AVX and AVX2).

[43] J. Jaroslavceva. A new trackster linking algorithm based on graph neural networks for the cms experiment at the large hadron collider at cern, 2023.

[44] Kokkos contributors. Kokkos Reference Documentation. URL https://kokkos.org/kokkos-core-wiki/.

[45] LLVM. Clang Compiler User's Manual: Command Line Reference. https://releases.llvm.org/18.1.6/tools/clang/docs/ReleaseNotes.html.

[46] F. Lombardelli. Reflection in C++ tramite template metaprogramming. Master's thesis, University of Pisa, 2005.

[47] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. 6 2015. doi: 10.4230/LIPIcs.ECOOP.2015.999.

[48] G. Mei and H. Tian. Impact of data layouts on the efficiency of GPU-accelerated IDW interpolation. *Natural Science Foundation of China*, 2 2016. URL https://arxiv.org/pdf/1402.4986.

[49] S. Meyers. Effective C++. URL https://gist.github.com/asambol/fa234c747ba4a677dee7b2ddaa64778d.

[50] G. Parolini. RNTuple Attributes: a RNTuple-native metadata system. In *Proceedings of ACAT 2025: 20th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Hamburg, Germany, Sep 2025. URL https://indico.cern.ch/event/1488410/contributions/6562809/contribution.pdf. Oral contribution ID 61.

[51] J. Pennycook, S. Hammond, S. Wright, A. Herdman, I. Miller, and S. Jarvis. An investigation of the performance portability of opencl. *Journal of Parallel and Distributed Computing*, 73:1439–1450, 11 2013. doi: 10.1016/j.jpdc.2012.07.005.

[52] A. Perego. Towards a time-aware global event interpretation at the cms experiment, 2023.

[53] PyTorch contributors. AOTInductor: Ahead-Of-Time Compilation for Torch.Exported Models, . URL https://docs.pytorch.org/docs/stable/torch.compiler_aot_inductor.html.

[54] PyTorch contributors. Using `torch::from_blob(...)` with const data, . URL https://discuss.pytorch.org/t/using-torch-from-blob-with-const-data/141597.

[55] P. K. Radtke and T. Weinzierl. *Compiler Support for Semi-manual AoS-to-SoA Conversions with Data Views*, page 301–314. Springer Nature Switzerland, 2025. ISBN 9783031856976. doi: 10.1007/978-3-031-85697-6_20. URL http://dx.doi.org/10.1007/978-3-031-85697-6_20.

[56] P. K. Radtke and T. Weinzierl. Annotation-guided AoS-to-SoA conversions and GPU offloading with data views in C++, 2025. URL https://arxiv.org/abs/2502.16517.

[57] ROOT contributors. ROOT Reference Documentation. URL https://root.cern/doc/v632/.

[58] S. F. Schifano. Performance impact of AoS vs SoA in Lattice Boltzmann Method on GPU. Presentation slides, 2017. Available at: https://agenda.infn.it/event/12156/contributions/12401/attachments/9151/10350/SCHIFANO_schifano-bari17.pdf.

[59] M. T. Schiller. SOAContainer. URL https://gitlab.cern.ch/LHCbOpt/SOAContainer.

[60] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandié, and R. Bird. Cabana: A Performance Portable Library for Particle-Based Simulations. *The Journal of Open Source Software*, 5 2022. URL https://www.researchgate.net/publication/359861145_Cabana_A_Performance_Portable_Library_for_Particle-Based_Simulations.

[61] M. Springer and H. Masuhara. Inner Array Inlining for Structure of Arrays Layout. *Proceedings of 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2018. URL https://m-sp.org/downloads/array2018.pdf.

[62] M. Springer, Y. Sun, and H. Masuhara. Ikra-Cpp: A C++/CUDA DSL for Object-Oriented Programming with Structure-of-Arrays Layout. *Workshop on Programming Models for SIMD/Vector Processing*, 2 2018. URL https://m-sp.org/downloads/wpmvp2018.pdf.

[63] J. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for

heterogeneous computing systems. *Computing in science & engineering*, 12:66–72, 05 2010. doi: 10.1109/MCSE.2010.69.

[64] R. Strzodka. Abstraction for AoS and SoA Layout in C++. *GPU Computing Gems Jade Edition*, 12 2012. URL https://asc.ziti.uni-heidelberg.de/content/St11ASX_CUDA.pdf.

[65] M. Swenson and T. Patil. Fast Hydraulic Erosion of Procedural Generated Terrain. Technical report, Carnegie Mellon University, 12 2019. Available at: https://github.com/patiltanma/15618-FinalProject.

[66] SYCL contributors. SYCL Reference Documentation. URL https://github.khronos.org/SYCL_Reference/.

[67] C. team. Cabana. URL https://github.com/ECP-copa/Cabana/tree/master.

[68] J. P. Wellisch, C. Williams, and S. Ashby. SCRAM: Software configuration and management for the LHC Computing Grid project, 2003. URL https://arxiv.org/abs/cs/0306014.

[69] N. Wilna. *Data-Oriented Design for Games*. Manning, 3 2024. ISBN 9781633435612.

[70] W. Yongwei. Static Reflection in C++. URL https://accu.org/journals/overload/32/184/wu/.

[71] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. Nagel, and M. Bussmann. Alpaka - an abstraction library for parallel kernel acceleration. 02 2016. doi: 10.48550/arXiv.1602.08477.

List of Figures

1.1 A comparison of the Array-of-Structs (AoS) and Struct-of-Arrays (SoA) memory layouts [51] . . . 2

2.1 AoS Layout . . . 7
2.2 SoA Layout . . . 8
2.3 AoSoA Layout . . . 10
2.4 Compilation process [29] . . . 11
2.5 Next Generation Trigger for CMS HLT . . . 16
2.6 CMSSW Framework diagram [17] . . . 18

4.1 Graph represented as a Structure of Arrays. Nodes and edges are organized in separate blocks. . . . 49
4.2 Building a generic View from x-direction columns for position and velocity 56
4.3 deepCopy the x-direction columns for position and velocity . . . 56
4.4 The View of a SoABlocks points to two different SoA Views. . . . 60
4.5 The SoABlocks is built by copying different SoAs. . . . 60
4.6 Machine Learning framework before and after the heterogeneous inference using SoAs . . . 61

5.1 SoABlocks vs traditional SoAs in CPU . . . 68
5.2 SoABlocks vs traditional SoAs in GPU . . . 69
5.3 AoS performance in CPU . . . 69
5.4 AoS performance in GPU . . . 70
5.5 Pointers vs Spans in CPU . . . 70
5.6 Pointers vs Spans in GPU . . . 71
5.7 Three SoAs vs Generic View in CPU . . . 72
5.8 Three SoAs vs Generic View in GPU . . . 72
5.9 Copy the data vs Generic View in CPU . . . 73
5.10 Copy the data vs Generic View in GPU . . . 73
5.11 Manual copies vs deepCopy in CPU . . . 74
5.12 Manual copies vs deepCopy in GPU . . . 75

5.13 SoA converter vs Standard approach 76
5.14 JIT vs AOT 77
5.15 Data structure conversion in CPU 77
5.16 Data structure conversion in GPU 78
5.17 N-body comparison in CPU 81
5.18 Invariant Mass comparison in CPU 81

Acknowledgements

I would like to acknowledge all the people who supported me during the development of this thesis and my academic path. First of all, I would like to thank Davide for his support and his guidance in helping me to present this work in the most coherent and a ccurate way. I want to express my gratitude to Felice, Andrea, and Eric, who have been true mentors for me during this year. They introduced me to my research path and contributed greatly to my personal growth. These thanks extend to my colleagues and friends at CERN, especially Simone (x2), Aurora, Tiziano, Christine, Markus, and Wahid. You have been amazing office mates, and my experience would not have been the same without you. I am grateful to my supervisor, Cristina, who allowed me to develop this thesis and has always been present and supportive. I want to thank my girlfriend, Nikitha, who brings out the best in me in every situation. My thanks also go to my friends, especially Luca, Vasileios, and Andrea, with whom I have shared both my joys and my fears, and who contributed so much to my peacefulness. Lastly, I want to thank my brother Giacomo, the best, and my parents, Luca and Rossella, as well as my grandparents, Gianna, Duvilio, Maria and Danilo, my uncle Davide and my aunt Marilla. I will always share the happiness of this achievement with them, and they have encouraged and supported me more than anyone else. To conclude, winning is important, but it is not the only thing that matters. *Ad maiora!*