



Optimizing Memory Access Patterns through Automatic Data Layout Transformation (Work in Progress Paper)

Jolly Chen
University of Twente
Enschede, The Netherlands
CERN
Geneva, Switzerland
jolly.chen@cern.ch

Ana Lucia Varbanescu
University of Twente
Enschede, The Netherlands
a.l.varbanescu@utwente.nl

Axel Naumann
CERN
Geneva, Switzerland
axel.naumann@cern.ch

Abstract

In many programming languages, memory access patterns exhibited by an application are dictated by the data structures defined by the programmer, which, in turn, dictate how the data are ordered in memory. Exploring access pattern optimizations is essential for performance: we demonstrate, through several benchmarks, the effects of Array of Structures (AoS) and Structure of Arrays (SoA) layouts on cache utilization, auto-vectorization, and false sharing. Despite these benefits, exploration remains a time-consuming task because it requires rewriting data structure definitions and, very often, computing kernel code to accommodate these changes.

We argue that such changes could and should be automated. In this work, we propose the design of a C++ framework for automatically redefining data structures to modify the data layout and access patterns. Leveraging experimental C++26 reflection and token injection features, we can modify the structure while preserving the original C++ syntax for accessing data. Our framework enables rapid prototyping of access pattern optimizations, potentially unlocking significant performance gains.

CCS Concepts

• **Software and its engineering** → **Source code generation**; • **General and reference** → **Performance**; • **Computing methodologies** → *Parallel programming languages*; • **Computer systems organization** → *Single instruction, multiple data*.

Keywords

Data Layouts, Memory Access Patterns, Automatic Layout Transformations, C++ Reflection, Array of Structures, Structures of Arrays, Array of Structures of Arrays

ACM Reference Format:

Jolly Chen, Ana Lucia Varbanescu, and Axel Naumann. 2025. 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 Companion '25)*, May 5–9, 2025, Toronto, ON, Canada. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3680256.3722203>



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICPE Companion '25*, Toronto, ON, Canada
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1130-5/2025/05
<https://doi.org/10.1145/3680256.3722203>

1 Introduction

As processors continue to grow faster and memory speeds lag behind, data-access time can significantly limit an application's performance. To overcome this "memory wall", mechanisms such as caching and prefetching were introduced [10]. However, the efficient utilization of these mechanisms heavily depends on the memory access pattern in the application's compute code (kernel).

Selecting the best-performing access pattern is a complex and time-consuming task. Software engineers aim to organize related data elements in logical data structures; doing so in many high-performance programming languages (e.g., C++, C, and Julia) results in a specific (pre-determined) memory layout of the data, which in turn dictates the memory access patterns the application exhibits when using the data. For example, Table 1 shows how a C++ data structure definition is related to the memory layout and the syntax used to access the data.

Modifying data layouts to test different access patterns requires rewriting data structure definitions. Very often, this also requires a change in the kernel, as the data access syntax changes with the definition. Especially for applications with complex data structure hierarchies - like arrays of structures (AoS), structures of arrays (SoA), or combinations thereof - such changes are non-trivial. Data structures become even more complex with nested structures, more data members, and array members of variable size. The more complex the data structures become, the larger the search space is, and the more coding effort is required to modify the layout.

Manual transformations also affect the readability of the code. For example, `particles[3].vertex.x` in AoS-style is more intuitive for getting the x-coordinate of the fourth particle's vertex position than `particles.vertex.x[3]` in SoA-style or `particle[0].vertex.x[3]` in AoSoA-style.

The complexity of data layout selection increases further as systems become more heterogeneous. For instance, while modern CPUs are relatively optimized for strided or irregular access patterns thanks to sophisticated caching and prefetching mechanisms, GPUs strongly favor massively parallel access to contiguous data, as they can be grouped into fewer memory requests (*coalescing*). Adapting the layout to each architecture becomes critical to achieving the best possible performance on all devices [8], and it significantly increases the complexity of the search problem.

To address this complex search for the best-performing memory access patterns, our work focuses on automating the conversion between different layouts, enabling faster prototyping and potentially

unlocking more performance. In this paper, we present a quantitative analysis to demonstrate the need for such conversion and introduce our *experimental C++ framework that uses code reflection and generation to transform layouts with minimal code changes*. We assess the possible impact of this framework for several *manually implemented C++ code examples* using different data layouts and showing both the non-trivial search process and the feasible performance gain.

2 The impact of the data layout

To demonstrate the impact of data layout conversion, we empirically investigate AoS and SoA versions for a set of benchmarks representing common access patterns. We assess performance and the layout effects on cache efficiency, auto-vectorization, and false sharing. We show that the "perfect" layout - i.e., the best performing one - is both critical for performance and non-trivial to determine intuitively/analytically.

2.1 Experimental Setup

All benchmarks were executed on a 64-bit machine with an Intel Core i9-9900K CPU. This processor supports AVX and AVX2 vector instructions and features a three-level cache hierarchy with capacities of 32KB (L1), 256KB (L2), and 16MB (L3). The cache line size is 64 bytes. The layout conversions in the benchmarks are *manually implemented* and compiled using GCC 14.2.0 with optimizations, auto-vectorization, and C++23 features enabled using the flags `-O3 -std=c++23 -march=skylake -ftree-vectorize`.

For more stable results, we disabled dynamic CPU frequency scaling using `cpupower frequency-set -governor performance` and pinned the threads to the number of required cores using `likwid-pin`¹. For benchmarking, we rely on Google Benchmark, which simplifies testing a kernel for multiple sets of parameters.

2.2 AoS to SoA

AoS and SoA organize and access the same data in very different ways. Depending on the hardware architecture and the parallelism model, the choice between the two data organization methods has significant performance implications. In the following sections, we demonstrate the impact of changing between the two organizations.

2.2.1 Cache-efficiency In modern processors, the smallest memory access unit is a cache line. A cache line size of 64 bytes can fit 8 double-precision values. Efficient cache utilization involves fully utilizing the data stored in every cache line and increased reuse of the same data - *spatial* and *temporal* locality, respectively.

To illustrate the effect on cache utilization when converting from AoS to SoA, we experiment with kernels operating on a data structure with N members. The AoS form uses one array of such N -field elements; the SoA form uses one structure with N "internal" arrays. We refer to one structure in the AoS (one array element) and the set of corresponding elements in SoA (corresponding elements in N arrays) as a *data element*. We further use two types of kernels: memory- or compute-intensive, classified based on their arithmetic intensity. They are implemented as follows:

- (1) **Memory-intensive:** for each data element, we perform 2 subtractions, 2 multiplications, and 1 addition per iteration.

- (2) **Compute-intensive:** for each data element, we perform 16 subtractions, 15 multiplications, 15 additions, and 16 divisions per iteration. We do not use compute-intensive methods such as `std::sqrt` and `std::log` because not all compilers automatically vectorize these by default.

The kernels are executed on both the AoS and the SoA configuration on a collection of 10 million data elements. Only the first two data members are accessed for each data element. The input data structure has N double-precision floating-point data members, with N varied as 2, 4, 8, or 20. The access pattern thus leaves 0%, 50%, 75%, or 90% of the members unused. The results of this experiment are shown in Figure 1.

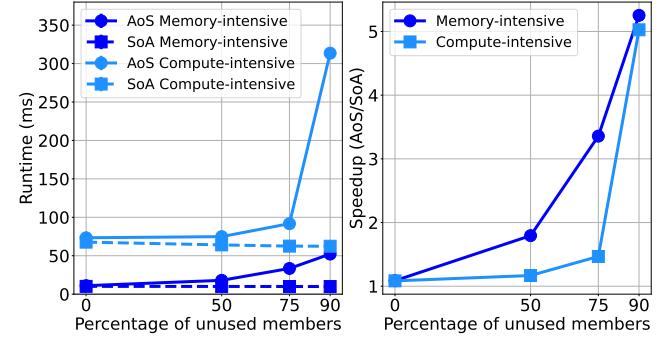


Figure 1: Average runtime results, with standard deviation error bars, and speedup of SoA over AoS for the memory-intensive and compute-intensive kernel with sequential access of the data elements.

In Figure 1, we observe that SoA outperforms AoS when accessing data members sparsely (i.e., unused data members > 0%). In these cases, SoA results in better cache efficiency because the accessed members are stored contiguously in memory, whereas AoS causes unnecessary loads of unused data members when loading a cache line. The performance penalty of unused data in a cache line grows more substantial with larger cache line sizes, as seen in GPUs. In contrast, when the data structure is fully accessed, the performance for both layouts is comparable because the full cache line is utilized with both layouts.

2.2.2 Auto-vectorization A second performance factor can be observed in Figure 1: we see a more pronounced performance benefit for the compute-intensive kernel because SoA also benefits SIMD vectorization due to the contiguous memory layout.

Auto-vectorization is a compiler optimization that converts scalar operations into vector operations using Single Instruction, Multiple Data (SIMD) instructions. This improves performance by enabling parallel processing of multiple data elements using vector registers. The data involved in the vectorized instructions must be contiguous in memory to be loaded into a vector register.

Using `objdump -D`, we analyzed the generated assembly code to determine which instructions are vectorized by the compiler. In Table 2 and 3, we summarize the instruction types observed in the generated assembly code of the kernels for 0% and 90% unused data, respectively. When the number of data elements is not evenly divisible by the size of the vector registers, the computational loop is divided into two parts: a main loop that performs fully vectorized operations and a remainder loop that handles the remaining

¹<https://github.com/RRZE-HPC/likwid>

Table 1: Examples of C++ data structure definitions, the corresponding memory layout, and syntax for accessing data elements. In these examples, we define data structures with four data *members* (x_1, x_2, x_3, x_4) with three different layouts: Array of Structures (AoS), Structure of Arrays (SoA), Array of Structures of Arrays (AoSoA). The illustrated memory layouts depict four data *elements* for AoS and SoA, and $3 \times 4 = 12$ elements for AoSoA.

C++ Data Structure Definition	Memory Layout	Access Syntax
<pre>struct S { double x1, x2, x3, x4; }; using AoS = std::array<S, 4>;</pre>		<code>auto first = aos[0].x1;</code>
<pre>struct SoA { std::array<double, 4> x1,x2,x3,x4; };</pre>		<code>auto first = soa.x1[0];</code>
<pre>using AoSoA = std::array<SoA, 3>;</pre>		<code>auto first = aosoa[0].x1[0];</code>

elements using either smaller vector registers or scalar operations. The summary table highlights only the assembly operations used in the main loop, as these instructions define the overall performance for large input data.

Instructions prefixed with *v* are “Vector” instructions; instructions ending with *sd* operate on scalar values, while instructions ending in *pd* operate on packed values, i.e., multiple values in parallel. The *fmadd* and *fnmadd* instructions denote “Fused Multiply Add” and “Fused Negative Multiply Add” respectively. These instructions perform multiplication and addition in a single step.

Table 2: Assembly Instruction Mix for different parameter combinations with 2 data members (0% Unaccessed)

Memory Intensive		Compute Intensive	
AoS	SoA	AoS	SoA
<i>vmovupd</i>	<i>vmovupd</i>	<i>vmovupd</i>	<i>vmovupd</i>
<i>vunpckhpd</i>		<i>vunpckhpd</i>	
<i>vunpcklpd</i>		<i>vunpcklpd</i>	
<i>vpermpd</i>		<i>vpermpd</i>	
		<i>vaddpd</i>	<i>vaddpd</i>
<i>vsubpd</i>	<i>vsubpd</i>	<i>vsubpd</i>	<i>vsubpd</i>
<i>vmulpd</i>	<i>vmulpd</i>	<i>vmulpd</i>	<i>vmulpd</i>
		<i>vdivpd</i>	<i>vdivpd</i>
<i>vfmadd132sd</i>	<i>vfmadd132pd</i>	<i>vfmadd231pd</i>	<i>vfmadd231pd</i>
		<i>vfnmadd231pd</i>	<i>vfnmadd231pd</i>

In Table 2, we observe that both AoS and SoA benefit from vectorized arithmetic operations when 100% of the data members are accessed. Vectorization enhances performance by processing multiple iterations simultaneously. The only difference between AoS and SoA is in loading the data into vector registers. For SoA, the data can be loaded directly using the *vmovupd* instruction because it is stored contiguously and in the correct order. Instead, AoS requires reordering the data within vectors using a sequence of instructions including *vmovupd*, *vunpckhpd*, *vunpcklpd*, and *vpermpd*.

Table 3: Assembly Instruction Mix for different parameter combinations with 20 data members (90% Unaccessed)

Memory Intensive		Compute Intensive	
AoS	SoA	AoS	SoA
<i>vmovsd</i>	<i>vmovupd</i>	<i>vmovsd</i>	<i>vmovupd</i>
<i>add</i>		<i>add, vaddsd</i>	<i>vaddpd</i>
<i>vsubsd</i>	<i>vsubpd</i>	<i>vsubsd</i>	<i>vsubpd</i>
<i>vmulsd</i>	<i>vmulpd</i>	<i>vmulsd</i>	<i>vmulpd</i>
		<i>vdivsd</i>	<i>vdivpd</i>
<i>vfmadd132sd</i>	<i>vfmadd132pd</i>	<i>vfmadd132sd</i>	<i>vfmadd132pd</i>
		<i>vfnmadd231sd</i>	<i>vfnmadd231pd</i>

In Table 3, however, we observe that only SoA benefits from vectorization. This is because the data used across iterations is only contiguous with the SoA layout.

2.2.3 False-Sharing False sharing is an effect that occurs when multiple threads modify data in the same cache line. Since a cache line is the smallest unit of operation in cache coherency protocols, the line is invalidated. It must be reloaded when other threads access it, even if they modify different elements in the line. As a result, performance degrades due to excessive cache misses.

Writes to the same cache line by different threads must be avoided to mitigate false sharing. Given, for example, an application where different threads write to uniquely different data structure members, false sharing can be avoided by choosing SoA over AoS.

To illustrate this effect, we consider a data structure with 8 double-precision floating-point data members and distribute the computation of these members across the available threads. Specifically, we experiment with configurations using 1, 2, 4, and 8 threads, where each thread writes to 8, 4, 2, and 1 consecutive data members, respectively. In this kernel, we perform 10 million sequential iterations over a smaller container containing 128 data elements. This increases the likelihood that threads write simultaneously to different data members of the same data element. Using the Google benchmark flag `-benchmark_perf_counters=L1-dcache-load-misses`, we also collected hardware performance counter data for L1 cache misses using *perf*.

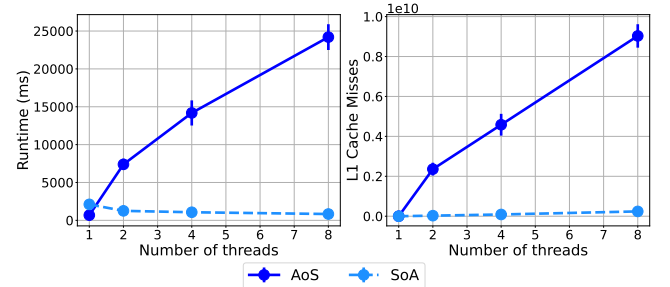


Figure 2: Average runtime and number of L1 data cache misses, with standard deviation error bars, of the false sharing example kernel.

Figure 2 presents the results of this experiment. We observe that the AoS layout results in an almost linear increase in runtime as the thread count increases. This behavior occurs because our test

system can fit exactly 8 doubles (i.e., one struct) into a single cache line. Consequently, when different threads simultaneously update the same data element, they write to the same cache line, resulting in an increase in cache misses due to false sharing. On the other hand, the SoA layout shows a constant runtime, as its data members are stored in separate arrays, effectively avoiding simultaneous access to the same cache line.

2.3 SoA to AoS

In previous examples, we saw that SoA performs better than AoS, but this is not always the case. An example of AoS outperforming SoA is when all data members are accessed, but we iterate over the elements in the AoS with a stride.

Consider a kernel that iterates over pairs of AoS/SoA structures with 8 data members. In each iteration, we perform arithmetic operations on the pair and write to an output structure, i.e., $aos_{out}[i].x = compute(aos_1[i].x, aos_2[i].x)$ or equivalently, $soa_{out}.x[i] = compute(soa_1[i], soa_2[i])$ for data members x_1 to x_8 . We iterate over 10 million data elements with a stride of 1, 2, 4, and 8. A stride of 1 is the same as accessing the elements sequentially.

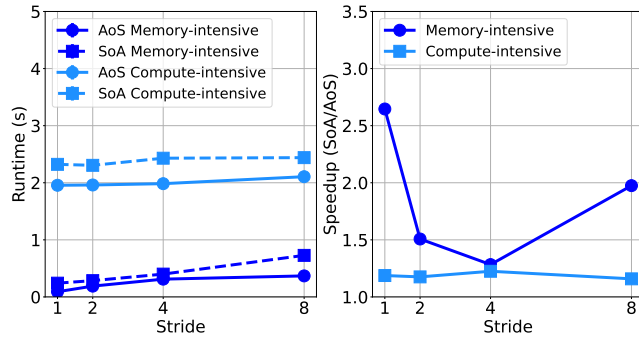


Figure 3: Average runtime, with standard deviation error bars, and speedup of AoS over SoA for a memory-intensive and compute-intensive kernel with sequential and strided access of data elements in pairs of AoS or SoA.

Figure 3 displays the result of this experiment. We observe that AoS performs better than SoA for non-zero strides. In the AoS layout, each iteration accesses an entire cache line, as a single structure’s data members are stored in memory contiguously. In contrast, in the SoA layout, the data members are stored in separate arrays. As a result, accessing each data member in a given iteration involves a greater distance between the memory locations of the data members, as each member is spaced by the number of data elements in the array. When the access is strided, the data for the data members in and across iterations are stored non-contiguously for SoA. Meanwhile, for AoS, the data is contiguous within iterations, allowing for vectorization.

In summary, our analysis shows the impact of different layouts, highlighting the performance differences and manual coding challenges. Next, we show our approach to automatically convert layouts to gain these benefits.

3 Automatic Layout Transformation

This section introduces our work-in-progress library, **reflmem++**², that enables automatic data layout conversion by modifying the C++ data structure definition. To achieve this, we utilize experimental C++ reflection features [1, 4] proposed for the upcoming C++26 standard (and beyond).

3.1 Requirements

We have the following requirements for our solution:

- R1. **Minimal performance overhead:** The solution should have a similar runtime performance to the manual implementation of a different layout.
- R2. **Minimal change in data access:** Modifying the data layout should have minimal impact on what syntax is used to access the data, minimize the number of code changes needed to use the transformer, and ensure that the interface remains familiar.
- R3. **Contiguous allocation:** The data of all members need to be stored in a contiguous chunk of memory, with proper alignment and possible reordering. This enables easier and faster copying of the data (to GPUs).
- R4. **Support variable-sized arrays:** The solution should support a variable number of data elements, with each data member potentially having a different size. This includes handling structures such as *jagged arrays*, where an AoS contains a vector member, and each data element may have a varying number of elements in its vector. Additionally, the solution should support adding and removing elements to the dynamically sized arrays.

3.2 C++ Reflection and Token Injection

The main reflection proposal paper [4] introduces a new operator, `^^x`, which “*lifts*” the operand `x` to a reflection value of opaque type `std::meta::info`. It provides facilities for inspecting a reflection value at compile-time, such as `type_of` and `identifier_of`. To produce grammatical elements that are interpreted as program code from reflection values, one can “*splice*” the reflection using the syntax `[: refl :]`. Listing 1 illustrates how the type and identifier of the data member in a simple data structure `S1` can be queried.

Listing 1: C++ Reflection Example (<https://godbolt.org/z/YnKv4nMWx>)

```

1 struct S1 { int x; };
2
3 // compile-time variable
4 constexpr std::meta::info member =
5     nonstatic_data_members_of(^^S)[0];
6
7 // Check the reflection of the type.
8 static_assert(type_of(member) == ^^int);
9
10 // Check the type of the member.
11 static_assert(
12     std::same_as<typename[ : type_of(member) :], int>);
13
14 // Check the identifier of the member variable.
15 static_assert(identifier_of(member) == "x");

```

A separate proposal [1] extends the main paper to include code injection in the form of token sequences. A token sequence can be defined using the syntax `^^{ ... }`, which can be injected into the program code after the evaluation of a compile-time `constexpr`

²<https://github.com/cern-nextgen/reflmempp>

block. This powerful feature, combined with the code inspection facilities, allows us to generate new data structure definitions. For example, Listing 2 defines a `std::vector<int>` data member named `x_vec` within the structure `S2`.

Listing 2: Token Injection Example (<https://godbolt.org/z/8Pa9vz9s9>)

```
1 struct S2 {
2   constexpr {
3     constexpr auto x = nonstatic_data_members_of(~S1)[0];
4     // \id(e) is replaced with an identifier created
5     // by the (concatenation of multiple) string-like e.
6     queue_injection(~{
7       std::vector<typename[: \(\type_of(x)) :]>
8       \id(identifier_of(x), "_vec"sv);
9     });
10 }
```

These features are currently implemented in the experimental reflection compiler from EDG, available on <https://godbolt.org/>. Since the reflection features are still under review, the syntax, semantics, and/or existence are not guaranteed. The essential features for our solution are the inspection of the type and identifier of a structure's data members, as well as the injection/generation of new data members based on reflected information.

Based on our experience with the experimental compiler, we found writing reflection and code injection code to be generally straightforward. The syntax is fairly intuitive and resembles C++, making it easy to work with. However, at the time of writing, when an invalid token sequence is injected, the compiler only points to the end of the `constexpr` block, even if it contains multiple injections. Furthermore, the provided error message, "not a constant value," is uninformative. As a workaround, we first implemented a manual version of the desired transformation to visualize the expected code before generalizing the solution with reflection. Ideally, we would have debugging tools that can preview the generated code, which would greatly enhance the development process.

3.3 Design and Implementation

We have designed and implemented the conversion of AoS to SoA. Within the namespace `rmpp`, we implemented a structure named `vector` that gets injected with code, turning it into the SoA version of an AoS. The structure `rmpp::vector` is templated on a user-defined data structure that serves as the proxy structure for an AoS view of the internal data. We have one requirement for this structure: data members of fundamental types (e.g., `int`, `double`, `float`) should be declared as reference types (i.e., `int&`, `double&`, `float&`) to ensure that the view does not own the data. We refer to this view structure as a *Structure-of-References* (SoR).

To illustrate the conversion performed by `reflmem++`, we use the `Particle` structure in Listing 3, containing nested structure data members, with member methods at each nesting level. To get an SoA version of the AoS with `Particle`, the user declares the structure `rmpp::vector<Particle>`. This structure stores the data in a contiguous chunk of bytes (i.e., `std::vector<byte>`), allowing for easy data copying.

The transformation is enabled by four components:

- **Data Allocation:** Our implementation provides a constructor that accepts the number of data elements as an argument. Given the size, we compute the number of bytes needed per SoA array

Listing 3: User-defined structures

```
1 template<typename T>
2 struct LorentzVector {
3   T &x, &y, &z, &t;
4   T Pt2() const { return std::sqrt(x*x + y*y); }
5   void SetPxPyPzE(T px, T py, T pz, T e) {
6     x = px; y = py; z = pz; t = e;
7   };
8
9   template<typename T>
10  struct Cartesian3D {
11    T &x, &y, &z;
12    void SetY(T yy) { y = yy; } };
13
14  template <typename T>
15  struct PositionVector3D {
16    Cartesian3D<T> coords;
17    void SetY(T yy) { coords.SetY(yy); } };
18
19  struct Particle {
20    int &id;
21    LorentzVector<double> mom; // momentum
22    PositionVector3D<double> ref; // reference
23
24    void SetId(int i) { id = i; }
25    double Pt2() const { return mom.Pt2(); } }
```

member to allocate the correct amount of bytes for the chunk storage, potentially padding to ensure correct alignment.

- **Generate SoA data members:** Based on the input SoR, we generate SoA array members by iterating over the data members of the SoR. For example, `Particle`'s `int id` member gets converted to `std::span<int>` `a`. A `std::span<T>` is a non-owning abstraction of a contiguous sequence of data of type `T`. If the data member is another struct, we recurse into the data members of that struct to define an SoA version of the nested struct.
- **Assigning SoA references:** Initializing the SoA spans to point to the correct offsets in the byte storage.
- **Overload indexing:** Overload the operator `[]` such that when the `rmpp::vector` object is indexed, it returns an object of the given SoR type (i.e., `Particle`), where the references point to locations in the storage.

Listing 4: Generated SoA for Particle

```
1 namespace rmpp {
2   template<typename T>
3   struct vector<Particle> {
4     std::vector<bytes> storage;
5     std::span<int> id;
6
7     struct LorentzVectorSoA {
8       std::span<double> x, y, z, t;
9     };
10    LorentzVectorSoA mom;
11
12    struct PositionVector3DSOA {
13      struct Cartesian3DSOA {
14        std::span<double> x, y, z;
15      };
16      PositionVector3DSOA coords;
17    };
18    PositionVector3DSOA ref;
19
20    Particle operator[](const size_t idx) {
21      return Particle{
22        id[idx], LorentzVector<double>{
23          mom.x[idx], mom.y[idx], mom.z[idx], mom.t[idx]},
24        PositionVector3D{Cartesian3D{
25          ref.coords.x[idx], ref.coords.y[idx],
26          ref.coords.z[idx]}}
27      };
28    }
29  };
30 }
```

Listing 4 displays the result of the transformation for the `Particle` structure. In Listing 5, we show how the generated structure can be used in a computational kernel. The structure allows for AoS-style

indexing, thanks to the overload of the indexing operator overload. This syntax is equivalent to the access syntax for a data structure defined as an AoS in C++, ensuring that no changes are needed in kernel code when applying the transformation. It also supports SoA-style indexing due to the injection of SoA data members with equivalent names. Thus, our solution effectively decouples the data-access interface from the memory layout.

Listing 5: Usage Example (<https://godbolt.org/z/4EfQ81Y4>)

```

1 using SoA = rmpp::vector<Particle>;
2 SoA particles(3);
3
4 // Access with SoA syntax
5 for (int i = 0; i < particles.id.size(); i++) {
6     particles.id[i] = i;
7 }
8
9 // Access with AoS syntax
10 particles[0].id = 100;
11 particles[1].ref.coords.z = 8888;
12
13 // Access to member methods
14 particles[0].SetId(200);
15 particles[1].ref.SetX(9999);
16 particles[2].mom.SetPxPyPzE(0, 0, 0, 0);

```

3.4 Challenges and Limitations

Our current prototype only supports converting AoS to SoA. We foresee several challenges and possible limitations in meeting the requirements (subsection 3.1).

First, it is unknown how much the use of an SoR impacts run-time performance (R1); we plan to measure this. Moreover, it is still unclear whether requiring an SoR as input from the user is reasonable, given how much code rewriting is needed to use our framework (R2). We have only considered scalar, struct, and fixed-size container types for the data members. However, we also need to support dynamically sized data members, such as `std::vector`, but this complicates the allocation of the data in a single chunk for different layouts (R3 and R4).

A potential limitation of our work is that reflection and token injection can significantly inflate compile times. The maximum recursion depth might also limit the size of the data structures and the level of nested structures our solution can handle.

4 Related Work

Our work shares the same goals as other methods that automate memory access pattern optimizations. The general process consists of two key steps: (1) inspecting the original data structure and/or kernel to determine the changes needed for a layout transformation, and (2) modifying the data structure definition and/or kernel code to implement the alternative layout. In Table 4, we summarize the approaches taken by related works.

For the first step, we identify three types of interfaces that can facilitate this process: *Domain-specific Language (DSL)*, *Directives or Pragmas*, and *Application Programming Interface (API)*. Our solution is an API, as we provide the class `rmpp::vector<T>` for users to define a data structure with SoA layout and AoS access syntax. We use code reflection in C++ to automatically gather details on the original data structure `T` at compile-time. By design, our approach requires significantly less code changes compared to previous works.

For the second step, a common solution is a source-to-source compiler that outputs code for a data structure with the desired

Table 4: Summary of the type of user interface, code generation method, stage at which the transformation occurs, and the target devices for related work. API = Application Programming Interface, DSL = Domain-Specific Language, S2S = source-2-source compiler, MP = Metaprogramming, CT = compile-time, RT = run-time

Reference	Interface	Method	Stage	Devices
Dymaxion++ [3]	Directives	S2S	RT	GPU
Grewe [5]	-	S2S	CT	CPU, GPU
HARDSI [9]	DSL	S2S	CT	CPU
LLAMA [6]	API	MP	CT	CPU, GPU
DL [11]	Directives	S2S	RT	GPU
Wende [12]	API	S2S	CT	CPU
CMS [2]	API	MP	CT	CPU, GPU
SoAx [7]	API	MP	CT	CPU, GPU
reflmem++	C++ API	MP	CT	CPU, GPU

layout. Existing metaprogramming features of the language(s) are also used to generate or manipulate code at compile-time. Some works apply layout changes fully at compile-time, while others allow dynamic layouts by enabling transformations at runtime. Our work employs experimental C++ token injection features, which are more powerful than current metaprogramming features (templates and macros), allowing for more complex changes at compile-time. In addition, by relying on native C++ features, we do not depend on another compiler.

The types of supported layouts vary for the different approaches. A range of works focuses on conversions between structured container types such as AoS, SoA, and AoSoA [2, 6, 7, 11, 12]. Others focus only on reordering data elements in scalar (nested) arrays (i.e., vectors and matrices) [3, 5, 9]. In our solution, we plan to support different structured containers, which can contain array data members with potential reordering of elements in the arrays.

5 Conclusion

To help select the best memory layout for a given application on various systems, we propose an automatic layout transformation framework. Our framework is motivated by preliminary empirical results, which show that (manual) layout conversion can lead to significant performance gain, but can be difficult to implement by hand. Instead, our framework, `reflmem++`, can automatically change layouts and access patterns within C++ code. With this method, we can modify the memory layout of a data structure by *generating new data structure definitions without requiring changes in existing kernel code*. Moreover, the solution preserves the access to user-defined member functions.

We pursue three directions of (future) research. We study end-to-end performance and programmability for `reflmem++`. We plan to compare our solution using code reflection with manual conversion and previously proposed methods using DSLs, APIs, and/or code generation for specific microbenchmarks and real applications. We also plan to extend the framework to support more layouts and member data types, such as Array-of-Structure-of-Arrays and variable-sized arrays.

Acknowledgments

This work has been funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.

References

- [1] Andrei Alexandrescu, Barry Revzin, and Daveed Vandevoorde. 2024. *Code Injection with Token Sequences*. WG21. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p3294r2.html>
- [2] Eric Cano. 2023. *Data layout: our reality*. Technical Report. CMS Collaboration. <https://indico.cern.ch/event/1347968/contributions/5674250/attachments/2763185/4812420/Data%20layout%20our%20reality.pdf>
- [3] Shuai Che, Jiayuan Meng, and Kevin Skadron. 2014. Dymaxion++: A Directive-Based API to Optimize Data Layout and Memory Mapping for Heterogeneous Systems. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops* (Phoenix, AZ, USA). IEEE, New York, NY, USA, 916–924. doi:10.1109/IPDPSW.2014.104
- [4] Wyatt Childers, Peter Dimov, Dan Katz, Barry Revzin, Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2024. *Reflection for C++26*. WG21. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2996r8.html>
- [5] Dominik Grewe, Zheng Wang, and Michael F. P. O’Boyle. 2013. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Shenzhen, China). IEEE, New York, NY, USA, 1–10. doi:10.1109/CGO.2013.6494993
- [6] Bernhard Manfred Gruber, Guilherme Amadio, Jakob Blomer, Alexander Matthes, René Widera, and Michael Bussmann. 2023. LLAMA: The low-level abstraction for memory access. *Software: Practice and Experience* 53, 1 (2023), 115–141. doi:10.1002/spe.3077 Publisher: Wiley Online Library.
- [7] Holger Homann and Francois Laenen. 2018. SoAx: A generic C++ Structure of Arrays for handling particles in HPC codes. *Computer Physics Communications* 224 (March 2018), 325–332. doi:10.1016/j.cpc.2017.11.015
- [8] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. 2011. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. *Transactions on Parallel and Distributed Systems* 22, 1 (1 2011), 105–118. doi:10.1109/TPDS.2010.107
- [9] Riyane Sid Lakhdar, Henri-Pierre Charles, and Maha Kooli. 2020. Data-layout optimization based on memory-access-pattern analysis for source-code performance improvement. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems* (St. Goar, Germany) (SCOPES ’20). Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3378678.3391874
- [10] Sally A. McKee. 2004. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers* (Ischia, Italy, 14) (CF ’04). Association for Computing Machinery, New York, NY, USA, 162. doi:10.1145/977091.977115
- [11] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W. Hwu. 2012. DL: A data layout transformation system for heterogeneous computing. In *2012 Innovative Parallel Computing (InPar)* (San Jose, CA, USA). IEEE, New York, NY, USA, 1–11. doi:10.1109/InPar.2012.6339606
- [12] Florian Wende. 2019. C++ Data Layout Abstractions through Proxy Types. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (Rio de Janeiro, Brazil). IEEE, New York, NY, USA, 758–767. doi:10.1109/IPDPSW.2019.00126