IN-MEMORY LAYOUT OF STRUCTURED DATA

KIWAKU, CONTAINERS WELL MADE

January 29, 2026

Adrien HENROT

Joël FALCOU, Hadrien GRASLAND, David CHAMONT

Context

- Unified access to data structures from HPC/HTC
- Simplification of their definition and usage
- Facilitate changes of in-memory layout
- Take advantage of memory hierarchies of modern processors

> This work aims to be a collaboration between HEP and computer science research.

Goal: Combining Ergonomics and Performance

- Data access should ideally be both **efficient** and **simple**
- Logical definition should not constrain optimization
- In-memory layout should be modifiable without algorithm rewrite
- Solution path: towards a memory layout eDSL

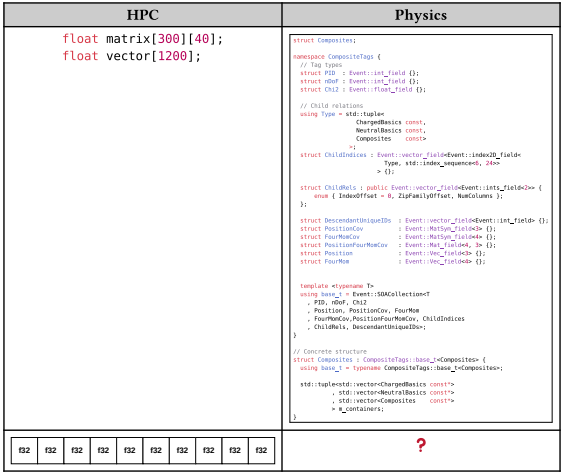
> Active research subject (LLAMA, SOACollection, Awkward Arrays,...)

Tools: C++

- Often used in HPC/HTC, powerfull and generalist language
- **Metaprogramming**: zero-cost abstraction (at runtime)
- **Concepts**: Constraints of generic code (templates)
- More compile time tools: constexpr, ...

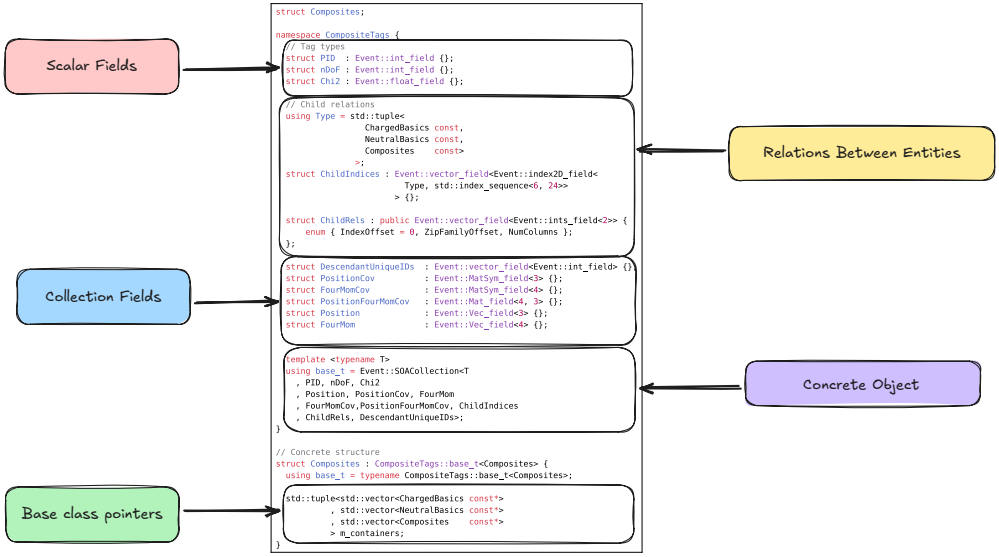
Structured Data

The Problem to be Solved



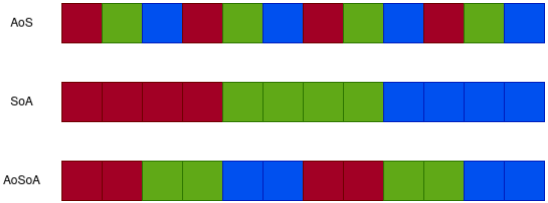
- Efficiently storing data structure coming from physics?
- Generalization beyond that?

Example from LHCb Code



Classical Data Structures

- **Array of Structure** (AoS)
  - Spatial locality per structure
  - Access by index then by field
- **Structure of Arrays** (SoA)
  - Spatial locality per field
  - Access by field then by index
- **Tiled Structure of Arrays**
  - Mixed locality
  - Access by tile then inside the tile



Example

Structure definition	Data access

`struct pixel {`
`  int r,g,b;`
`};`
`std::array<pixel,10> aos{};`
	`for(int i = 0; i < 10; ++i)`
	`  print("{}", aos[i].r);`

`template<std::size_t N>`
`struct pixels {`
`  std::array<int,N> r,g,b;`
`};`
`pixels<10> soa{};`
	`for(int i = 0; i < 10; ++i)`
	`  print("{}", soa.r[i]);`

`constexpr int simd_with = 2;`
 	
`template<std::size_t N>`
`struct block {`
`  std::array<int,simd_width> r,g,b;`
`};`
`std::array<blocks,5> aosoa{};`
	`for(int i = 0; i < 5; ++i)`
	`  for( int j = 0; j < simd_width; ++j)`
	`  print("{}", aosoa[i].r[j]);`

Observations

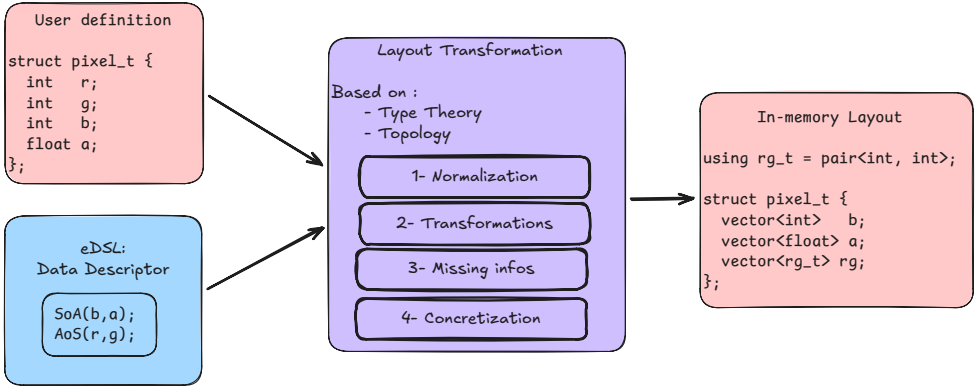
- Fields are fixed: tuple(R,G,B)
- Similar spaces
  - **Logical**: User-defined structure
  - **Physical**: In-memory realization
- The total number of elements is fixed:

$$nb\_elements = \sum fields * \sum pixels = 3 * nb\_pixels$$

- Different in-memory layout → different ways to access fields

Proposed Solution

Pipeline



- How to handle user-defined structures?
- How to use the final structure?

1- Structure Normalization

- Fields are fused
- Structures are flattened
- Collections are nested

API	Implementation

`using info = pair<char, int>;`
 	
`struct hit { vector<info> data; };`
 	
`struct tracks {`
`  vector<hit> hits;`
`  vector<float> momentum;`
`} my_tracks{};`
	`struct tracks {`
	`  vector<float> momentum;`
	`  vector<vector<char>> hits_data_0,`
	`  vector<vector<int>> hits_data_1;`
	`} new_tracks{};`

2- Applying Transformations

- Fields are reindexed
- Containers are re-attributed
- New names cannot be introduced

API	Implementation

`struct tracks {`
`  vector<hit> hits;`
`  vector<float> momentum;`
`} my_tracks{};`
 	
`constexpr auto layout =`
`aos("momentum", "hits.data.1");`
	`struct combined {`
	`  float momentum;`
	`  vector<int> hits_data_1;`
	`}`
	
	`struct part_1 {`
	`  vector<combined> members;`
	`};`

3- Filling Missing Infos

- Unspecified fields are considered (not dropped)
- SoA layout by default
- In reality: Containers are linearized

User Side	Internally

`using info = pair<char, int>;`
 	
`struct hit { vector<info> data; };`
 	
`struct tracks {`
`  vector<hit> hits;`
`  vector<float> momentum;`
`} my_tracks{};`
	`struct combined {...};`
	
	`struct final {`
	`  vector<combined> part;`
	`  vector<vector<char>> hits_data_0;`
	`} final_tracks{};`

4- Concretization

- The structure is built: Index + logical field → Physical element

```
// How it would work at runtime
template<typename T>
constexpr auto final_tracks::resolve(string path, T idxs) {
  if ( path == "hits.data.0" )
    return this.hits_data_0[get<0>(idxs)][get<1>(idxs)];
  else if ( path == "hits.data.1" )
    return this.part[get<0>(idxs)].hits_data_1[get<1>(idxs);
  else
    return this.part[get<0>(idxs)].momentum;
};

auto item = my_tracks.resolve("hits.data.1", pair{0, 0});
```

Actually handled at compile time: a structure layout can't depend on runtime parameters.

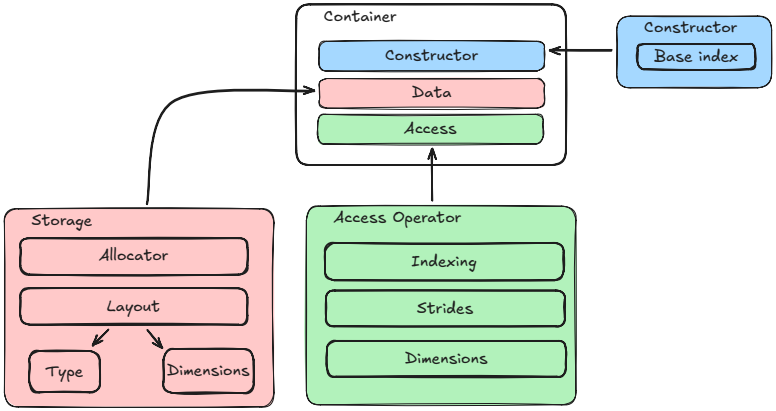
Kiwaku

Multidimensional Arrays Library

- C++20 and onward library
- Two main composants
  - kwk::view and kwk::table (*ownership* semantic)
- Options to define properties

Options	Values
`kwk::source`	pointers, static arrays, contiguous ranges
`kwk::size`	`kwk::of_size(dn...)`, integers: 1D
`kwk::label`	String like value
`kwk::allocator`	A type modeling `kwk::concepts::allocator`

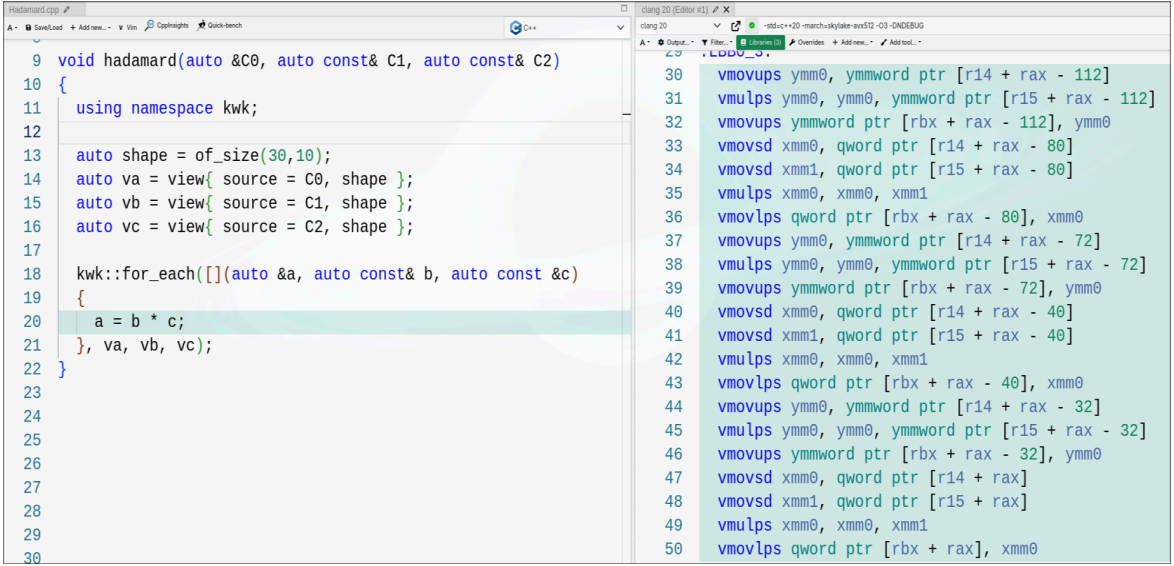
Internal Organization



- Type: element to store: 🏗️
- Allocator: Handles memory: 🚶
- Storage Order:
  Order of the dimensions: 🏗️
- Dimensions:
  Sizes per dimensions: 🏗️/🚶
- Strides:
  Padding/Dimension jump: 🏗️
- Base index: default index value: 🚶
- Indexing: boundary conditions: 🏗️
- Layout: Where are the data: 🏗️

🏗️: Compile Time Info
🚶: Runtime Info

User API



```
Hadamard.cpp
A- Save/Load + Add new... - Vim CppInsights Quick-bench C++
 9  void hadamard(auto &C0, auto const& C1, auto const& C2)
10  {
11    using namespace kwk;
12
13    auto shape = of_size(30,10);
14    auto va = view{ source = C0, shape };
15    auto vb = view{ source = C1, shape };
16    auto vc = view{ source = C2, shape };
17
18    kwk::for_each([](auto &a, auto const& b, auto const &c)
19    {
20      a = b * c;
21    }, va, vb, vc);
22  }
23
24
25
26
27
28
29
30
```

```
clang 20 (Editor #1)
clang 20 -std=c++20 -march=skylake-avx512 -O3 -DNDEBUG
A- Output... - Filter... - Libraries (3) Overrides + Add new... - Add tool...
.LBB0_3:
30    vmovups ymm0, ymmword ptr [r14 + rax - 112]
31    vmulps ymm0, ymm0, ymmword ptr [r15 + rax - 112]
32    vmovups ymmword ptr [rbx + rax - 112], ymm0
33    vmovsd xmm0, qword ptr [r14 + rax - 80]
34    vmovsd xmm1, qword ptr [r15 + rax - 80]
35    vmulps xmm0, xmm0, xmm1
36    vmovlps qword ptr [rbx + rax - 80], xmm0
37    vmovups ymm0, ymmword ptr [r14 + rax - 72]
38    vmulps ymm0, ymm0, ymmword ptr [r15 + rax - 72]
39    vmovups ymmword ptr [rbx + rax - 72], ymm0
40    vmovsd xmm0, qword ptr [r14 + rax - 40]
41    vmovsd xmm1, qword ptr [r15 + rax - 40]
42    vmulps xmm0, xmm0, xmm1
43    vmovlps qword ptr [rbx + rax - 40], xmm0
44    vmovups ymm0, ymmword ptr [r14 + rax - 32]
45    vmulps ymm0, ymm0, ymmword ptr [r15 + rax - 32]
46    vmovups ymmword ptr [rbx + rax - 32], ymm0
47    vmovsd xmm0, qword ptr [r14 + rax]
48    vmovsd xmm1, qword ptr [r15 + rax]
49    vmulps xmm0, xmm0, xmm1
50    vmovlps qword ptr [rbx + rax], xmm0
```

Advanced Layout

- **kwk::tiles**:
  - Each tile can have its own iteration pattern
- **Interleaving**:
  - Across flat fields: ordering
  - Across inner structures: hierarchy
- **Heterogeneous data**:
  - Field types (product types)
  - Inner array sizes (jagged arrays)

Proof Of Concept

- Standard structure definition
- Simple way to define a reorganization

```
using namespace kumi; using namespace kwk;

struct pixel { using is_record_type = void; int r,g,b; float a; };

// Adatation of struct to record protocol
std::tuple_size<pixel> : std::integral_constant<std::size_t, 4>;
template<> struct std::tuple_element<0, pixel>{ using type = field_capture<"r", int>;   };
template<> struct std::tuple_element<1, pixel>{ using type = field_capture<"g", int>;   };
template<> struct std::tuple_element<2, pixel>{ using type = field_capture<"b", int>;   };
template<> struct std::tuple_element<3, pixel>{ using type = field_capture<"a", float>; };

using pixel_map       = std::vector<pixel>;
constexpr auto layout = soa("g"_f, "a"_f);

auto pixels = structure(pixel_map{}, layout, 120);
```

With C++26/ROOT Reflection

- Standard structure definition
- Simple way to define a reorganization

```
using namespace kumi; using namespace kwk;

struct pixel { int r,g,b; float a; };

using pixel_map       = std::vector<pixel>;
constexpr auto layout = soa("g"_f, "a"_f);

auto pixels = structure(pixel_map{}, layout, 120);
```

Data Access

- Uniform access between different layouts
- Algorithms are independent of physical layout
- Efficient data access is guaranteed

```
auto to_grayscale = [](auto elt) {
    auto &[r,g,b,a] = elt;
    int gray = static_cast<int>(0.2126f * r + 0.7152f * g + 0.0722f * b);
    r = g = b = gray;
    return tuple{r,g,b,a};
};

int main() {
  auto rng = pixels.as_range();
  std::ranges::transform(rng, rng.begin(), to_grayscale);
}
```

Beyond std: Kiwaku Contexts

- Kiwaku supports several execution contexts
  - **simd**: EVE for vectorization
  - **kiwaku**: Native Kiwaku algorithms
  - **standard**: STL compatibility

- Different contexts → specific traversal algorithms

```
auto context = kwk::simd;
auto context = kwk::kiwaku;
auto context = kwk::standard;
```

In the end Kiwaku aims at supporting other contexts, for example: *multi-thread*

Conclusion

- High level abstraction with no impact on performance
- Simplified changes of memory layout
- Separation of concerns: Optimization/algorithms
- Simplification to write generic code
- Portable performance and equivalent to specialized code

Next Steps

- Clean up jagged array support
- Sum types support (unions, std::variant, ...)
- Sparse data representations (*COO*, *CSC*, *CSR*, ...)
- Context composition (SIMD, Multi-thread, GPU, ...)
- Writing/adaptating traversal algorithms
- Automate layout selection? (C++26 reflection, ROOT fallback)

Thanks for your attention

- Kiwaku: https://github.com/jfalcou/kiwaku/
- Kumi: https://github.com/jfalcou/kumi/

This work is funded by IN2P3 technical direction.

Annexes

Benchmark: Toy example