Evolution of data structures for heterogeneous reconstruction in CMSSW

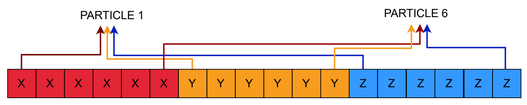
Beltrame L. [1, 2] Pantaleo F. [2] Bocci A. [2] Cano E. [2]

On behalf of the CMS Collaboration

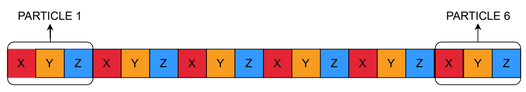
[1]Polytechnic of Milan [2]European Organization for Nuclear Research (CERN)

Abstract

An SoA stores each field in its own contiguous array, in contrast to the Array of Structures (AoS) layout, where object fields are interleaved.



SoA Layout



AoS Layout

While object-oriented designs are conceptually affine to High Energy Physics, they often map to AoS and become inefficient. Especially for heterogeneous computing, which is widely used in CMSSW, SoA layouts provide better cache locality, and coalesced GPU accesses. To support more flexible and efficient workflows, we deliver a user-friendly, portable SoA that preserves an object-oriented syntax while achieving data-oriented design efficiency.

SoA in CMSSW

The CMSSW SoA implementation strategy divides layout organization, data-access patterns, and metadata, implemented using `Boost::PP` library, from memory ownership, allocation, and host↔device transfers, which are handled by `PortableCollection` on top of alpaka for performance portability across CPUs and GPUs.

```
GENERATE_SOA_LAYOUT(PositionSoATemplate,
    SOA_COLUMN(float, x),
    SOA_COLUMN(float, y),
    SOA_COLUMN(float, z),
    SOA_EIGEN_COLUMN(Eigen::Vector3f, vel),
    SOA_SCALAR(float, time))

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

```
//  Portable Collection
PortableCollection<PositionSoA, Device>
            position(elems, queue);
// Portable Collection View
auto positionView = position.view();
positionView.time() = /* scalar value */
positionView.x()[i] = /* value */
positionView[i].y() = /* value */
positionView[i].vel() = /* Eigen vector */
```

Custom Element Methods

With a simple syntax, it is possible to declare user-defined methods directly within the SoA backend. Methods definition is supported for AoS `element` and `const_element`, as well as for `View` and `ConstView`. C++ reflection (partially available from C++26) shows promising results for this application.

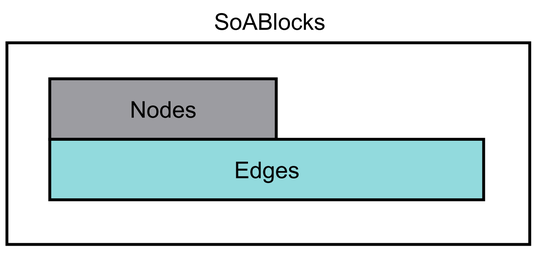
```
GENERATE_SOA_LAYOUT(
  PositionSoATemplate,
  SOA_COLUMN(float, x),
  SOA_COLUMN(float, y),
  SOA_COLUMN(float, z),
  SOA_EIGEN_COLUMN(Eigen::Vector3f, vel),
  SOA_METHODS(
    void sort() {
        // sort the columns
    }
  ),
  SOA_CONST_METHODS(
    std::array<float, 3> centroid() const {
        // return the centroid
    }
  ),
  SOA_SCALAR(float, time)
)
```

```
GENERATE_SOA_LAYOUT(
  PositionSoATemplate,
  SOA_COLUMN(float, x),
  SOA_COLUMN(float, y),
  SOA_COLUMN(float, z),
  SOA_EIGEN_COLUMN(Eigen::Vector3f, vel),
  SOA_ELEMENT_METHODS(
    void velocity() {
        // compute velocity
    }
  ),
  SOA_CONST_ELEMENT_METHODS(
    std::array<float, 3> position() const {
        // return the position
    }
  ),
  SOA_SCALAR(float, time)
)
```

SoA Blocks

`SoABlocks` group columns into *blocks*: all columns within a block share the same length, while different blocks may have different lengths. This blocked layout is designed with future `ROOT::RNTuple`–based serialization in mind and remains fully compatible with user-defined methods, enabling custom SoAs (e.g., association maps, graphs, etc.). `SoABlocks::View` exposes internal Views by user-defined names and it has been integrated with `PortableCollection`.

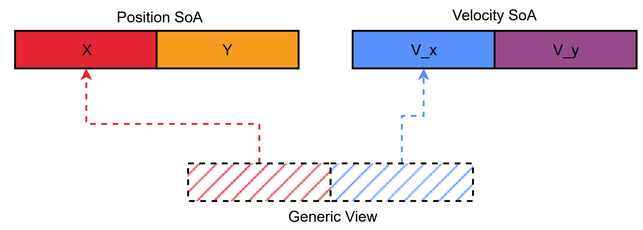
```
GENERATE_SOA_LAYOUT(SoAEdges, SOA_COLUMN(Edge, edges))
GENERATE_SOA_LAYOUT(SoANodes, SOA_COLUMN(Node, nodes))

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;
    }
  ))
```



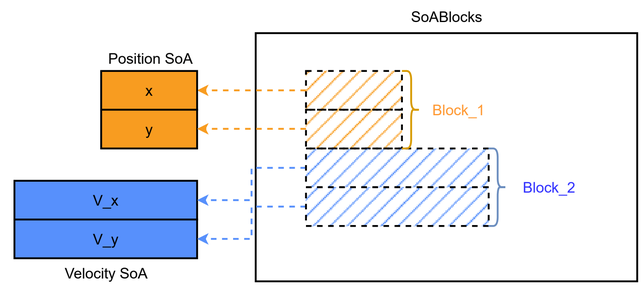

```
void graph_function() {
    auto graphView = graph.view();
    auto size = graphView.metadata().size()[0];
    for (std::size_t i = 0; i < size; i++) {
        Node node = graphView.nodes()[i];
        // operator[] to get the set of edges
        EdgeSet edgeSet = graphView[node];
        // do something with edgeSet
    }
}
```

Generic SoA overlay

Building a SoA view by combining columns from different SoAs enables greater flexibility since it is possible to access data from different memory blobs using a single object. The `Metarecords` struct provides the utilities to build such a View.



The natural extension of this feature for `SoABlocks` is to consider different SoA views that will overlay and build a `SoABlocks::View`.



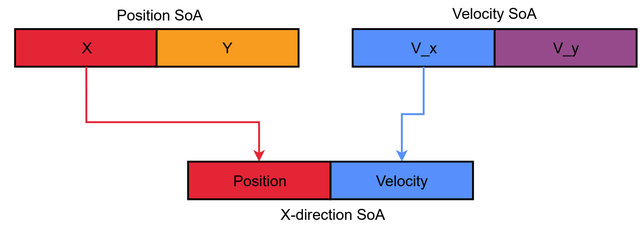
The user syntax is pretty simple:

```
auto positionRecords = positionView.records();
auto velocityRecords = velocityView.records();
// Building the View from different memory blobs
GenericSoA::View genericView(positionRecords.x(),
                             velocityRecords.v_x());
```

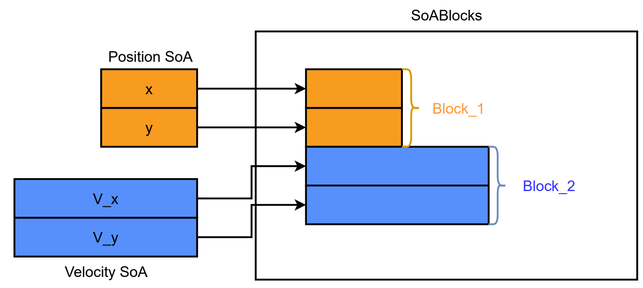
ML inference Poster:

Heterogeneous Data Transfer

It is possible to `deepCopy` the `GenericView` we have defined so far into a physical memory layout. The `deepCopy` provides a backend-agnostic abstraction for memory transfers using alpaka. Data transfer is performed column by column, using the concept of `Descriptor` (and its const variant) to describe an SoA (index, size, pointers, and stride) and enabling compile-time loop over columns without relying on names.



In case of `SoABlocks`, the concept is the same and the `deepCopy` is performed view by view.



The syntax for the user is the following, where `sizes` is either an integer or an array of integers:

```
// Deep copy a view in a new PortableCollection
PortableCollection<GenericSoA, Device> genericSoA(sizes, queue);
genericSoA.deepCopy(genericView, queue);
```

Future developments

Many new improvements and features are being added for CMSSW's SoA:

- Improving data access, moving to the modern `std::span`s of C++20.
- Supporting the new RNTuple feature from ROOT.
- `MultiSoAViewManager`: class that simulates a single view access pattern concatenating a variable number of SoA `View`s with the same layout.

Aknowledgment

Supported by the Eric Wendy Schmidt Fund for Strategic Innovation (grant agreement SIF-2023-004)