

# A deeper look at SoA in CMSSW

November 7<sup>th</sup>, 2023

Andrea Bocci, Eric Cano

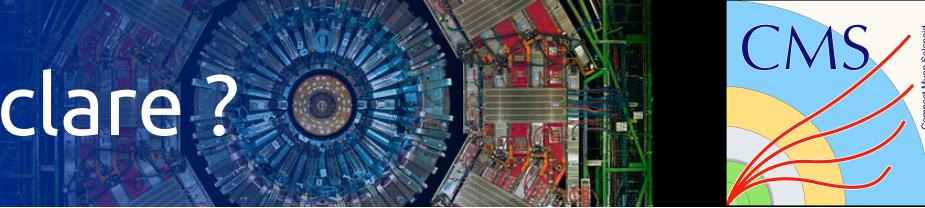
CERN, EP-CMD

# what do we declare ?



```
using PFRecHitsNeighbours = Eigen::Matrix<int32_t, 8, 1>;  
  
GENERATE_SOA_LAYOUT(PFRecHitSoALayout,  
                     SOA_COLUMN(uint32_t, detId),  
                     SOA_COLUMN(float, energy),  
                     SOA_COLUMN(float, time),  
                     SOA_COLUMN(int, depth),  
                     SOA_COLUMN(PFLayer::Layer, layer),  
                     SOA_EIGEN_COLUMN(PFRecHitsNeighbours, neighbours),  
                     SOA_COLUMN(float, x),  
                     SOA_COLUMN(float, y),  
                     SOA_COLUMN(float, z),  
                     SOA_SCALAR(uint32_t, size)  
)  
  
using PFRecHitSoA = PFRecHitSoALayout<>;
```

# what do we declare?



```
using PFRecHitsNeighbours = Eigen::Matrix<int32_t, 8, 1>;
```

type alias to avoid  
commas in macro call

```
GENERATE_SOA_LAYOUT(PFRecHitSoALayout)
```

declare a column:

- field type
- field name
- capacity is fixed\*  
at construction

```
SOA_COLUMN(uint32_t, detId),
```

```
SOA_COLUMN(float, energy),
```

```
SOA_COLUMN(float, time),
```

```
SOA_COLUMN(int, depth),
```

```
SOA_COLUMN(PFLayer::Layer, layer),
```

```
SOA_EIGEN_COLUMN(PFRecHitsNeighbours, neighbours),
```

```
SOA_COLUMN(float, x),
```

```
SOA_COLUMN(float, y),
```

```
SOA_COLUMN(float, z),
```

```
SOA_SCALAR(uint32_t, size)
```

Eigen matrix object in SoA format,  
with one column per matrix element

declare a scalar:

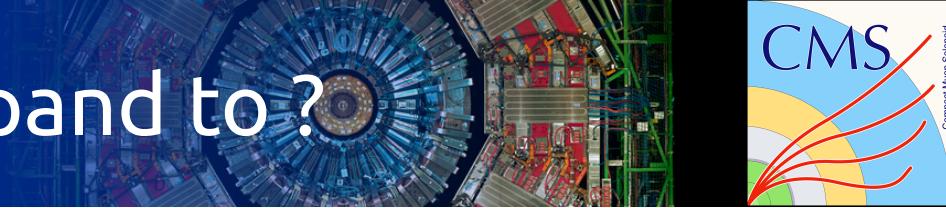
- field type
- field name
- single element

```
using PFRecHitSoA = PFRecHitSoALayout<>;
```

optional template arguments enable  
bounds checking, different alignment, etc



# what does it expand to?



```
template <std::size_t ALIGNMENT = cms::soa::CacheLineSize::defaultSize,
          bool ALIGNMENT_ENFORCEMENT = cms::soa::AlignmentEnforcement::relaxed>
struct PFRecHitSoALayout {
    struct Metadata { /* ... */ };
    const Metadata metadata() const { return Metadata(*this); }

    using ConstView = ConstViewTemplate<cms::soa::RestrictQualify::enabled, cms::soa::RangeChecking::disabled>;
    using View = ViewTemplate<cms::soa::RestrictQualify::enabled, cms::soa::RangeChecking::disabled>;

    PFRecHitSoALayout(std::byte *mem, size_type elements) : mem_(mem), elements_(elements) {
        organizeColumnsFromBuffer();
    }

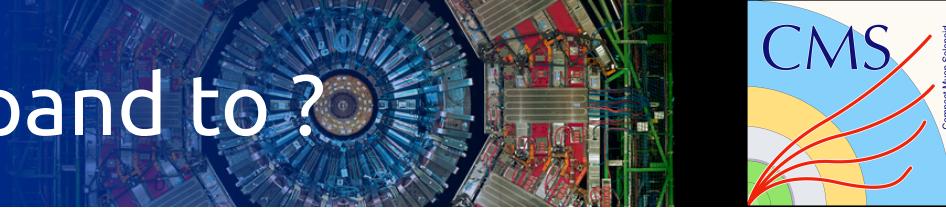
    static constexpr byte_size_type computeDataSize(size_type elements);
    template <typename T>
    void ROOTReadStreamer(T &onfile);
    void ROOTStreamerCleaner();
    void organizeColumnsFromBuffer();

    // data members ...
}
```

the View provides access to the data,  
and can be trivially copied e.g. to GPUs

almost 2000 lines of code !

# what does it expand to?

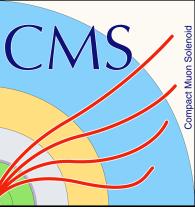


```
// ...  
  
std::byte *mem_ [[clang::annotate("!")]];  
size_type elements_;  
size_type const scalar_ = 1;  
byte_size_type byteSize_ [[clang::annotate("!")]] = 0;  
uint32_t *detId_ [[clang::annotate("[elements_]")]] = nullptr;  
float *energy_ [[clang::annotate("[elements_]")]] = nullptr;  
float *time_ [[clang::annotate("[elements_]")]] = nullptr;  
int *depth_ [[clang::annotate("[elements_]")]] = nullptr;  
PFLayer::Layer *layer_ [[clang::annotate("[elements_]")]] = nullptr;  
size_type neighboursElementsWithPadding_ = 0;  
PFRecHitsNeighbours::Scalar *neighbours_ [[clang::annotate("[neighboursElementsWithPadding_]")]] = nullptr;  
byte_size_type neighboursStride_ = 0;  
float *x_ [[clang::annotate("[elements_]")]] = nullptr;  
float *y_ [[clang::annotate("[elements_]")]] = nullptr;  
float *z_ [[clang::annotate("[elements_]")]] = nullptr;  
uint32_t *size_ [[clang::annotate("[scalar_]")]] = nullptr;  
};  
  
using PFRecHitSoA = PFRecHitSoALayout<>;
```

# what does it expand to?



```
// ...                                mark as transient  
  
std::byte *mem_ [[clang::annotate("!")]];      only #ifdef __CLING__ to avoid warnings  
size_type elements_;  
size_type const scalar_ = 1;                      mark as variable-length array  
byte_size_type byteSize [[clang::annotate("!")]] = 0;  
uint32_t *detId_ [[clang::annotate("[elements_]")]] = nullptr;  
float *energy_ [[clang::annotate("[elements_]")]] = nullptr;  
float *time_ [[clang::annotate("[elements_]")]] = nullptr;  
int *depth_ [[clang::annotate("[elements_]")]] = nullptr;  
PFLayer::Layer *layer_ [[clang::annotate("[elements_]")]] = nullptr;  
size_type neighboursElementsWithPadding_ = 0;  
PFRecHitsNeighbours::Scalar *neighbours_ [[clang::annotate("[neighboursElementsWithPadding_]")]] = nullptr;  
byte_size_type neighboursStride_ = 0;  
float *x_ [[clang::annotate("[elements_]")]] = nullptr;  
float *y_ [[clang::annotate("[elements_]")]] = nullptr;  
float *z_ [[clang::annotate("[elements_]")]] = nullptr;  
uint32_t *size_ [[clang::annotate("[scalar_]")]] = nullptr;  
};  
                                         use scalar_ data member, "[1]" does not seem to work  
  
using PFRecHitSoA = PFRecHitSoALayout<>;
```



# how do we use them ?

```
// generic SoA-based product in host memory
template <typename T>
class PortableHostCollection {
public:
    using Layout = T;
    using View = typename Layout::View;
    using Buffer = cms::alpakatools::host_buffer<std::byte[]>; const variants left out for simplicity

    // part of the ROOT read streamer
    static void ROOTReadStreamer(PortableHostCollection* newObj, Layout& layout);

    PortableHostCollection(int32_t elements, alpaka_common::DevHost const& host)
        : buffer_{cms::alpakatools::make_host_buffer<std::byte[]>(Layout::computeDataSize(elements))},
          layout_{buffer_->data(), elements},
          view_{layout_} {} allocate memory for the SoA
build the SoA in the buffer
build the view from the SoA

private:
    std::optional<Buffer> buffer_; //! mark as transient
    Layout layout_; //!
    View view_; //! mark as transient
};
```



# how do we read them back?



- with bare ROOT
  - individual columns can be read without special dictionaries
- within CMSSW
  - SoA are embedded in a `PortableHostCollection<T>`
  - `PortableHostCollection<T>` uses an explicit read streamer
    - destroy the default-constructed object created by ROOT
    - construct in place a `PortableHostCollection<T>` with the correct size
      - allocate a single memory buffer
      - construct the SoA layout to create the columns and scalars within the buffer
      - construct the View to the layout
    - call the SoA `ROOTReadStreamer` to copy the content into the newly allocated buffer
    - free the memory allocated by ROOT



# how do we read them back?



```
<class name="reco::PFRecHitHostCollection"/>
<read
    sourceClass="reco::PFRecHitHostCollection"
    targetClass="reco::PFRecHitHostCollection"
    version="[1-]"
    source="reco::PFRecHitSoA layout_;" 
    target="buffer_,layout_,view_"
    embed="false">
    <![CDATA[
        reco::PFRecHitHostCollection::ROOTReadStreamer(newObj, onfile.layout_);
    ]]>
</read>
```

```
// part of the ROOT read streamer
static void ROOTReadStreamer(PortableHostCollection* newObj, Layout& layout) {
    newObj->~PortableHostCollection();
    // use the global "host" object returned by cms::alpakatools::host()
    new (newObj) PortableHostCollection(layout.metadata().size(), cms::alpakatools::host());
    newObj->layout_.ROOTReadStreamer(layout);
    layout.ROOTStreamerCleaner();
}
```



# how do we read them back?

```
template <typename T>
void ROOTReadStreamer(T &onfile) {
    auto size = onfile.metadata().size();
    memcpy(detId_, onfile.detId_, sizeof(uint32_t) * onfile.elements_);
    memcpy(energy_, onfile.energy_, sizeof(float) * onfile.elements_);
    memcpy(time_, onfile.time_, sizeof(float) * onfile.elements_);
    memcpy(depth_, onfile.depth_, sizeof(int) * onfile.elements_);
    memcpy(layer_, onfile.layer_, sizeof(PFLayer::Layer) * onfile.elements_);
    memcpy(neighbours_, onfile.neighbours_, sizeof(PFRecHitsNeighbours::Scalar) * neighboursElementsWithPadding_);
    memcpy(x_, onfile.x_, sizeof(float) * onfile.elements_);
    memcpy(y_, onfile.y_, sizeof(float) * onfile.elements_);
    memcpy(z_, onfile.z_, sizeof(float) * onfile.elements_);
    memcpy(size_, onfile.size_, sizeof(uint32_t));
}

void ROOTStreamerCleaner() {
    delete[] detId_; detId_ = nullptr;
    delete[] energy_; energy_ = nullptr;
    delete[] time_; time_ = nullptr;
    delete[] depth_; depth_ = nullptr;
    delete[] layer_; layer_ = nullptr;
    delete[] neighbours_; neighbours_ = nullptr;
    delete[] x_; x_ = nullptr;
    delete[] y_; y_ = nullptr;
    delete[] z_; z_ = nullptr;
    delete[] size_; size_ = nullptr;
}
```

# help needed



- how can we make this work better ?
- some ideas
  - implement the read streamer in C++, without the need for XML
  - document and extend the use of clang attributes and annotations ?
  - let the framework provide the memory area to ROOT
- **we need your suggestions !**

