Evolution of the ATLAS event data model for the HL-LHC

Scott Snyder
S. Swatman, A. Kraszahorkay, P. Gessinger
On behalf of the ATLAS Computing Activity

Brookhaven National Laboratory, Upton, NY, USA

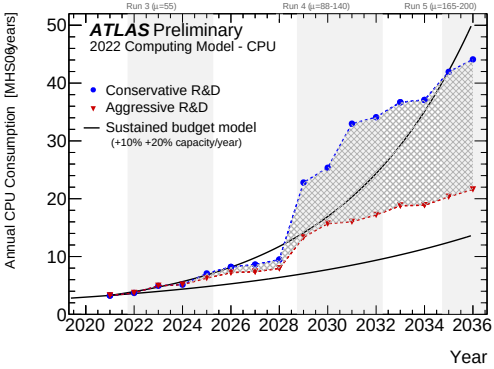
Oct 21, 2024
CHEP 2024

Motivation

Run 4 of the LHC (HL-LHC) will be starting in several years.

- Will record data at a rate 7–10× more than at present.
- Would require an increase in computing resources significantly more than is expected.

Need aggressive software R&D to reduce resource requirements.



[CERN-LGCC-2022-005]

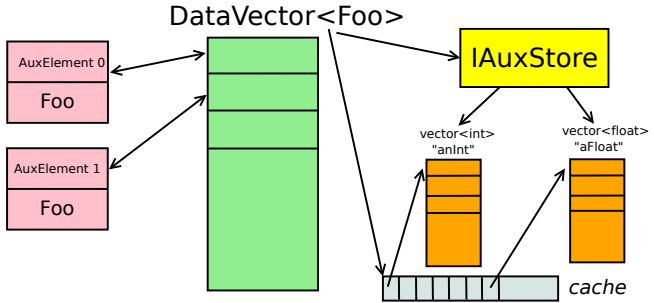
Will discuss some recent developments to the ATLAS EDM motivated by reducing resource requirements and/or by making it easier to share data with accelerators like GPUs.

xAOD event data model

Analysis-level event data model ("xAOD") uses a structure-of-array organization.

- `DataVector<T>` holds pointers to objects. Acts like `std::vector<T*>`.
- Attach data of arbitrary type to elements of `DataVector`.
- Data stored as vectors, in separate "auxiliary store" object accessed via abstract interface.

Almost all analysis object data stored as auxiliary data rather than in `T` instances.



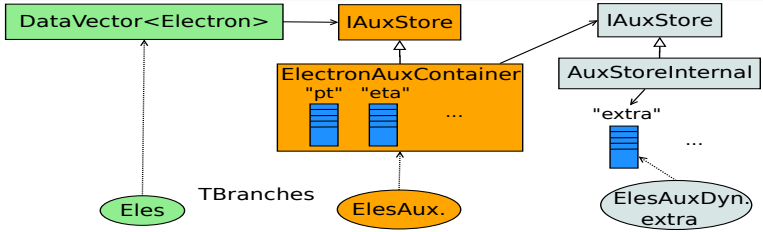
I/O interaction

Most xAOD data kept in a 'static' store object.

- Has a bunch of `std::vector` members.
- Given to ROOT to save.

Static store can reference an additional 'dynamic' store.

- Contains extra variables.
- Storage managed dynamically, but also as `std::vector` instances.
- I/O system treats each dynamic variable individually.



Accessor

xAOD variables usually accessed via an `Accessor` object.

Caches name lookup and type checks.

```
static const SG::Accessor<float> fvarAcc ("fvar");
DataVector<MyContainer> cont = ...;
MyClass& elt = *cont[0];

float f1 = fvarAcc (elt);   // Read/write the variable for one element.
fvarAcc (elt) = calcFVar();

// Can also get the variable directly from the container.
float f2 = fvarAcc (cont, 0);

// Iterate over the variable for all container elements.
for (float f : fvarAcc.getDataSpan (cont)) { ... }
```

Nested (jagged) vectors

Sometimes we want to store a vector of values for each element in a container.

- Where the number of entries may vary from element to element (jagged).
- Usually stored as `std::vector<T>`.
- So the variable will be held in a `std::vector<std::vector<T> >`.

But this is inefficient.

- Multiple scattered memory allocations; three extra pointers per element.
- More difficult to share with coprocessors like GPUs.

Can often do better by flattening the data to a single vector.

- With a separate vector giving indices.
- For example, `{{3, 1, 2}, {4}, {6, 5}}` might be stored as:
- Payload: `{3, 1, 2, 4, 6, 5}`
- Indices: `{3, 4, 6}` (*End* index of each inner vector.)

xAOD representation and linked variables

A jagged vector is represented by *two* xAOD variables.

```
std::vector<SG::JaggedVecElt<float> > fvec; // Contains single 32-bit index
std::vector<float> fvec_linked;             // Vector payload data
```

- First variable holds the end indices. It always has the same length as the container, like ordinary xAOD variables.
- Second variable holds the payload data, and is called a "linked" variable.
- Unlike ordinary xAOD variables, its length is independent from that of the container. It is associated with an ordinary xAOD variable, and is not referenced directly by user code.

Implies that a `JaggedVecElt` cannot be interpreted on its own, but only as part of an array.

Macro `AUXVAR_JAGGEDVEC_DECL` provided to properly declare the two members needed for a static xAOD jagged vector.

Using jagged vectors

`Accessor` classes are specialized for `JaggedVecElt`.

Act like a `std::vector`. Can convert to/from a vector, and common vector operations (both `const`/non-`const`) are supported.

```
static const SG::Accessor<SG::JaggedVecElt<float> > fvec("fvec");
DataVector<MyContainer> cont = ...;
MyClass& elt = *cont[0];

fvec(elt) = std::vector<float> {1.5, 2.5, 3.5};
assert (fvec(elt).size() == 3);
assert (fvec(elt)[1] == 2.5);
std::vector<float> v = fvec(elt);
fvec(elt).push_back(4.5);
for (float f : fvec(elt)) ...
```

Using jagged vectors (ranges)

Can also deal with a range over all the nested vectors in a container.

Acts like a `std::vector<std::vector<T> >`.

```
static const SG::Accessor<SG::JaggedVecElt<float> > fvec("fvec");
DataVector<MyContainer> cont = ...;

auto r = fvec.getDataSpan (cont);
std::cout << r[1].size() << " " << r[1][0];
r[2].push_back (4.5);
for (auto v : r) {
  for (float& f : v) {
    std::cout << " " << f;
    f += 1;
  }
}
```

Packed links

ATLAS EDM has two ways of storing persistent references between containers in the event store.

- `DataLink<CONT>` references a container of type `CONT`.
- `ElementLink<CONT>` references an element of a container of type `CONT`.

The transient representation of `ElementLink` takes four words.

For some HL-LHC tracking applications, this can take a significant amount of memory.

Can this memory footprint be reduced?

Store a list of unique `DataLinks` to the containers involved along with a list of (container, element) index pairs.

(Similar to the pre-xAOD `ElementLinkVector` class.)

Packed link variables

A packed link is again represented by two xAOD variables.

```
// Each PackedLink holds an 8-bit index into plinks_linked and a
// 24-bit element index.
std::vector<SG::PackedLink<CONT> > plinks;

// Links for each unique container referenced.
std::vector<DataLink<CONT> > plinks_linked;
```

- The list of `DataLinks` is again a linked variable.
- The first entry in `plinks_linked` is a null link, so a null element link can be consistently represented as a `PackedLink` containing all 0's.

Again, macros are provided for declaring static `PackedLink` members.

Using packed links

Accessor classes are specialized for PackedLink and vector<PackedLink>.

Acts like a (vector of) ElementLink.

```
static const SG::Accessor<SG::PackedLink<Cont> > plink("plink");
static const SG::Accessor<std::vector<SG::PackedLink<Cont> > > pvec("pvec");
DataVector<Cont> cont = ...;
MyClass& elt = ...;

plink(elt) = ElementLink<Cont> (cont, 1);
plinkvec(elt).push_back (ElementLink<Cont> (cont, 2));
assert (plink(elt).index() == 1);
assert (plinkvec(elt).size() == 1);
assert (plinkvec(elt)[0].index() == 2);
for (ElementLink<MyContainer> el : plinkvec(elt)) ...
```

Range-based access works as well.

Flexible memory allocation

Can change the allocator used for the `std::vector` holding variables.

In particular, `std::pmr::polymorphic_allocator` can be used instead of `std::allocator`.

```
// Auxiliary store class.
class MyClassAuxContainer_v1 : public xAOD::AuxContainerBase {
  using xAOD::AuxContainerBase::xAOD::AuxContainerBase;
  // Define an auxilary variable of type int using a polymorphic_allocator.
  AUXVAR_DECL (int, anInt, std::pmr::polymorphic_allocator);
  // This will declare a member
  //  std::vector<int, std::polymorphic_allocator<int> > anInt;
  // and also set up the needed registration.
};
  // Making an instance.
  std::pmr::memory_resource* res = ...;
  auto auxstore = std::make_unique<MyClassAuxContainer_v1> (res);
```

Further comments

I/O for the new features mentioned here is fully functional for `TTree`.

- Changing allocator on a container is both backwards and forwards compatible.

Testing with RNtuple in progress.
Not supported:

- Schema evolution of dynamic variables.
- Non-default allocators.

Some of these features were only fully available very recently.

- So no performance information as of yet.

Vecmem

[J. Phys.: Conf. Ser. 2438 012050]

- Library for managing memory in heterogeneous systems.
- Part of ATLAS/ACTS R&D on parallelization.
- Provides C++17 memory resources to manage device memory for CUDA, SYCL, and HIP.
- Plus STL-like containers to use in device code as well as support for types with structure-of-arrays organization.

- Allows collaboration between host xAOD-based code and device non-xAOD code.
- Possible extra copy on host not significant compared to moving data between host and device.
- Could also take advantage of xAOD allocator support to use Vecmem to directly manage xAOD memory on the host in such a way that it is shared with accelerators.

Summary

Several new features added to the ATLAS EDM

- Jagged vectors: storing nested vector xAOD variables using a flattened (jagged) representation.
- Packed links: Compact in-memory storage for vectors of links between objects in the event store.
- Arbitrary allocators: Allows managing the allocation of xAOD data via the C++17 `memory_resource` interface.

Full integration in Athena ongoing.

- Including Python-based columnar analysis.

Vecmem library provides efficient interoperation between host and GPU code.

IAuxStore

A key feature is the ability to change the auxiliary store implementation through the abstract interface. Some types used:

Each xAOD type has a static auxiliary store chained to a dynamic store.

In trigger: implementation specialized for storage in raw data stream.

On input: implementation allowing on-demand reading of items.

"Shallow copy" store: records writes, forwards reads for unknown items to another store.