Data layout: our reality

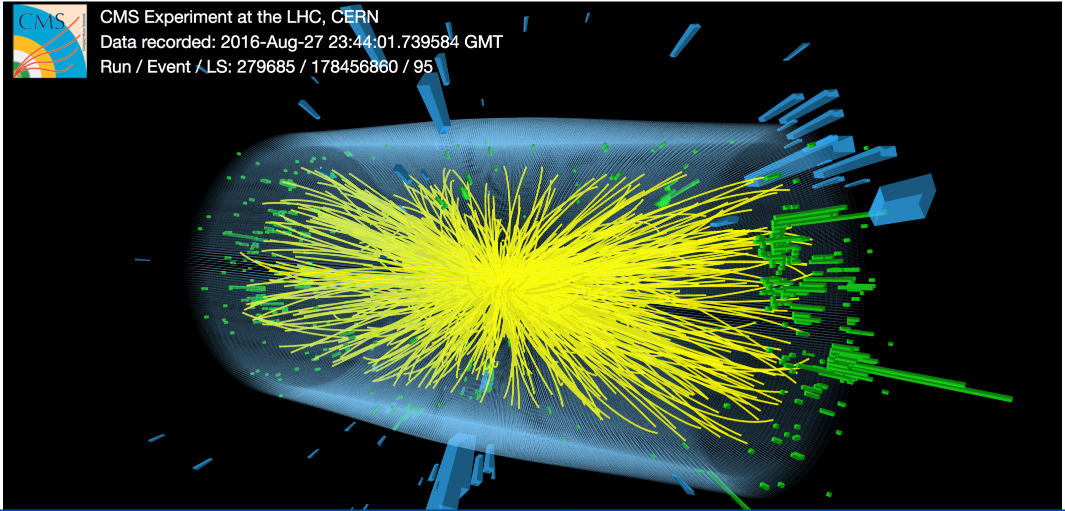
Eric Cano[1] on behalf of LHC experiments

[1] CERN, Geneva

Outline

- Event data processing and storage
- CMS data layout
  - Layout
  - User interface
  - Layout definition
  - Macro implementation
- ALICE data layout
- ATLAS data layout
- LHCb data layout

Event pileup=30 here, 140+ for HL-LHC


CMS
CMS Experiment at the LHC, CERN
Data recorded: 2016-Aug-27 23:44:01.739584 GMT
Run / Event / LS: 279685 / 178456860 / 95

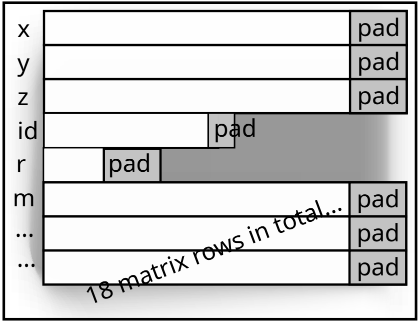
Event data processing

High energy physics context (simplified)

- Consecutive events are physically independent
- Events processed one by one by independent threads/processes.
  - Embarrassingly parallel problem at that scale
- Events: collections of $\mathcal{O}(10k)$ physical objects (detector hits)
- Turned into successive collections of reconstructed objects
  - Hits → clusters → tracks → vertexes, jets, particules. . .
  - High combinatorial complexity at that scale, joining:
    - hits into clusters
    - clusters into tracks
    - tracks into vertices

Efficient data access in GPUs and storage

Buffer with SoA Layout



- GPU efficient data layout of objects
  - Structure of arrays (SoA)
  - Coalesced access by concurrent threads
  - Minimizes the number of cache lines needed per GPU load/store operation
- Transfer efficiency (no unified memory): columns collocated in a single buffer per object collection
- Using the ROOT framework to store data
- Requires members annotations (`[[clang::annotate ...]]`)

CMS: Defining SoAs

- Macro based SoA definition

```
#include <Eigen/Core>
#include <Eigen/Dense>
#include "DataFormats/SoATemplate/interface/SoALayout.h"

GENERATE_SOA_LAYOUT(TestSoATemplate,
                    SOA_COLUMN(double, x),
                    SOA_COLUMN(double, y),
                    SOA_COLUMN(double, z),
                    SOA_COLUMN(double, value),
                    SOA_EIGEN_COLUMN(Eigen::Vector3d, a),
                    SOA_EIGEN_COLUMN(Eigen::Vector3d, b),
                    SOA_EIGEN_COLUMN(Eigen::Vector3d, r),
                    SOA_SCALAR(uint32_t, someNumber))

using TestSoA = TestSoATemplate<>;
```

CMS: Using SoAs

```
// Device-only producer kernel
__global__ void producerKernel(TestSoA::View soa, const unsigned int numElements) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (0 == i)
    soa.someNumber = 42;
  if (i >= numElements)
    return;
  auto si = soa[i];
  si.value() = sqrt(si.x() * si.x() + si.y() * si.y() + si.z() * si.z());
  si.r() = si.a().cross(si.b());
}
```

- Almost AoS syntax
  - "array element" is a reference, not a copy
  - element member access requires the use of `operator()`
- Eigen vector and matrix elements also strided in $N * M$ columns.

CMS: Using SoAs (II)

```
for (size_t i = 0; i < numElements; ++i) {
  auto si = h_soahd[i];
  // Tuple assignment...
  // elements are: x, y, z, a, b, r
  auto v1 = 1.0 * i + 1.0;
  auto v2 = 2.0 * i;
  auto v3 = 3.0 * i - 1.0;
  if (i % 2) {
    si = {v1, v2, v3, {v1, v2, v3}, {v3, v2, v1}, {0, 0, 0}};
  } else {
    si.x() = si.a()(0) = si.b()(2) = v1;
    si.y() = si.a()(1) = si.b()(1) = v2;
    si.z() = si.a()(2) = si.b()(0) = v3;
  }
}
```

- List initializer
- Full element to element copy also availble

CMS: Macros implementation

```
#define _ITERATE_ON_ALL(MACRO, DATA, ...) BOOST_PP_SEQ_FOR_EACH(MACRO, DATA, BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__))
[...]
#define _ACCUMULATE_SOA_ELEMENT_IMPL(VALUE_TYPE, CPP_TYPE, NAME)                                              \
  _SWITCH_ON_TYPE(VALUE_TYPE,                                                                                \
      /* Scalar */                                                                                           \
      ret += cms::soa::alignSize(sizeof(CPP_TYPE), alignment);                                               \
      ,                                                                                                      \
      /* Column */                                                                                           \
      ret += cms::soa::alignSize(elements * sizeof(CPP_TYPE), alignment);                                    \
      ,                                                                                                      \
      /* Eigen column */                                                                                     \
      ret += cms::soa::alignSize(elements * sizeof(CPP_TYPE::Scalar), alignment) * CPP_TYPE::RowsAtCompileTime \
             * CPP_TYPE::ColsAtCompileTime;                                                                  \
  )

/* The for_each construct requires a 2-stage macro */
#define _ACCUMULATE_SOA_ELEMENT(R, DATA, TYPE_NAME) _ACCUMULATE_SOA_ELEMENT_IMPL TYPE_NAME
[...]
#define GENERATE_SOA_LAYOUT(CLASS, ...)                                                                      \
  template <CMS_SOA_BYTE_SIZE_TYPE ALIGNMENT = cms::soa::CacheLineSize::defaultSize,                         \
            bool ALIGNMENT_ENFORCEMENT = cms::soa::AlignmentEnforcement::relaxed>                            \
  struct CLASS {                                                                                             \
[...]
    static constexpr byte_size_type computeDataSize(size_type elements) {                                    \
      byte_size_type ret = 0;                                                                                \
      _ITERATE_ON_ALL(_ACCUMULATE_SOA_ELEMENT, ~, __VA_ARGS__)                                               \
      return ret;                                                                                            \
    }                                                                                                        \
```

CMS: Macros implementation

Resulting (clang-formated) code for example above: size computation helper function

```
static constexpr byte_size_type computeDataSize(size_type elements) {
  byte_size_type ret = 0;
  ret += cms::soa::alignSize(elements * sizeof(double), alignment);
  ret += cms::soa::alignSize(elements * sizeof(double), alignment);
  ret += cms::soa::alignSize(elements * sizeof(double), alignment);
  ret += cms::soa::alignSize(elements * sizeof(uint32_t), alignment);
  ret += cms::soa::alignSize(elements * sizeof(Eigen::Vector3d::Scalar), alignment) *
         Eigen::Vector3d::RowsAtCompileTime * Eigen::Vector3d::ColsAtCompileTime;
  ret += cms::soa::alignSize(elements * sizeof(Eigen::Vector3d::Scalar), alignment) *
         Eigen::Vector3d::RowsAtCompileTime * Eigen::Vector3d::ColsAtCompileTime;
  ret += cms::soa::alignSize(elements * sizeof(Eigen::Vector3d::Scalar), alignment) *
         Eigen::Vector3d::RowsAtCompileTime * Eigen::Vector3d::ColsAtCompileTime;
  ret += cms::soa::alignSize(sizeof(uint32_t), alignment);
  return ret;
```

- Overall this 8 members SoA generates 1800 LoC
- Macros themselves make up 2 files totaling 1400 LoC

CMS: Macros implementation

- Data members of the same layout
- Annotations for ROOT serialization
  - Only if the compiler is cling, the dictionary generator of ROOT to avoid warnings
- Most likely the trickiest bit to port to reflection

```
std::byte* mem_ [[clang::annotate("!")]];
size_type elements_;
size_type const scalar_ = 1;
byte_size_type byteSize_ [[clang::annotate("!")]];
double* x_ [[clang::annotate("[elements_]")]] = nullptr;
double* y_ [[clang::annotate("[elements_]")]] = nullptr;
double* z_ [[clang::annotate("[elements_]")]] = nullptr;
uint32_t* value_ [[clang::annotate("[elements_]")]] = nullptr;
size_type aElementsWithPadding_ = 0;
Eigen::Vector3d::Scalar* a_ [[clang::annotate("[aElementsWithPadding_]")]] = nullptr;
byte_size_type aStride_ = 0;
size_type bElementsWithPadding_ = 0;
Eigen::Vector3d::Scalar* b_ [[clang::annotate("[bElementsWithPadding_]")]] = nullptr;
byte_size_type bStride_ = 0;
size_type rElementsWithPadding_ = 0;
Eigen::Vector3d::Scalar* r_ [[clang::annotate("[rElementsWithPadding_]")]] = nullptr;
byte_size_type rStride_ = 0;
uint32_t* someNumber_ [[clang::annotate("[scalar_]")]] = nullptr;
```

CMS: SoA experience

- Reliable data layout generation
- Fully integrated with memory management, event data framework and serialization
- Also internalizes previously duplicated code (`restrict` access)
- Lightweight support load
- But macros were tedious to debug
  - Involved copy-pasting the macro expansion in the source to understand issues in generated code
  - Similar feature probably needed for reflection

ALICE: data formats

- Apache Arrow for AODs
- On GPU mostly AoS and SoA (implemented manually) for select use cases.
  - For TPC tracking, SoA was not always helpful, since we are limited by random access to the hit array.
  - Often we do not fetch hits adjacent in memory, but we need all coordinates of a hit, so sometimes AoS gives even better locality.
  - Where SoA is beneficial and needed, it is simply implemented manually as independent C++ array.
- All data transfer to / from GPU is by copying flat linear data buffers. Data structures are such, that the buffers are self-contained (with some pre / postprocessing).

ATLAS: data formats

- SoA is a lot more work in progress at the moment
- Working on jagged container in VecMem for tracking (on GPU)
- Concrete VecMem based EDM for GPU

```
vecmem::host_memory_resource m_mem;
m_vec({vecmem::vector<int>({1, 2, 3, 4}, &m_mem),
       vecmem::vector<int>({5, 6}, &m_mem),
       vecmem::vector<int>({7, 8, 9, 10}, &m_mem),
       vecmem::vector<int>({11}, &m_mem), vecmem::vector<int>(&m_mem),
       vecmem::vector<int>({12, 13, 14, 15, 16}, &m_mem)},
       &m_mem),
m_data(vecmem::get_data(m_vec)),
m_jag(m_data) {}

// Fill the jagged vector buffer with just the odd elements.
vecmem::jagged_device_vector device_vec(output_data);
for (std::size_t i = 0; i < m_vec.size(); ++i) {
    for (std::size_t j = 0; j < m_vec.at(i).size(); ++j) {
        if ((m_vec[i][j] % 2) != 0) {
            device_vec[i].push_back(m_vec[i][j]);
        }
    }
}
```

ATLAS: data formats (II)

- much more dynamic xAOD containers on host
- Runtime polymorphic way of handling SoA containers in Athena

```
          obj = new SG::AuxElement();
    interface.push_back( obj );
    obj->auxdata< int >( "AnInt" ) = 7;
    obj->auxdata< float >( "AFloat" ) = 0.7;
```

- Concrete VecMem based EDM for GPU
- GPU code will be able to get hold of the host arrays through std::span like "views"
- If C++ allows us to put a nicer API on top of these types of variadic containers we would be very happy about
- Don't think it will play any significant role in how we would do I/O with these types

LHCb: data formats

- SoA Event model for SIMD CPU
- Type based template meta programming (using tag types)
- Builiding block: field
- Composed into SoACollections, and SOARelations
- Iterated on via proxies with scalar and SIMD flavors. Proxies reference a chunk of N objects, contiguous or scattered
- Field access with a type-parameterized getters

LHCb: data formats (II)

```
// Define tags:
struct Momentum : float_field {};
struct LHCbIds : vector_field<int_field> {};
// Define collection:
struct Tracks : SOACollection<Tracks, Momentum, LHCbIds> {};

// Push N elements to the end of tracks
auto proxy = tracks.emplace_back<simd>();
proxy.field<Momentum>().set(momentum);

// Iterate over tracks N elements at a time
for (const auto& proxy : tracks.simd()) {
  auto momentum = proxy.get<Momentum>();
}
```

Conclusion

- A variety of SoA approaches from experiment to experiment
- Different approaches to solve the member access problem
  - Code generation
  - Type indexed (like in LLAMA)
  - String indexed
- Reflection should make both user and framework code lighter (no more need to pick one)