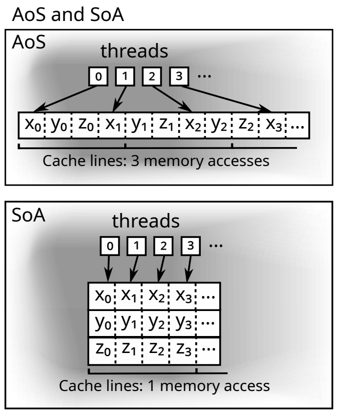
Implementation of generic SoA data structures in the CMS software

Eric Cano, Andrea Bocci

21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research
23 - 27 October 2022, Bari Italy

Data layout in GPUs: Array of structure (AoS) vs Structure of Arrays (SoA)


AoS and SoA
AoS
threads
0
1
2
3
...
x0 y0 z0 x1 y1 z1 x2 y2 z2 x3 ...
Cache lines: 3 memory accesses
SoA
threads
0
1
2
3
...
x0 x1 x2 x3 ...
y0 y1 y2 y3 ...
z0 z1 z2 z3 ...
Cache lines: 1 memory access

- GPUs drive multiple ALUs with a shared scheduler
- Maximizes silicon space devoted to ALUs and therefore computing power
- Executes 32, and up to 64 threads simultaneously in a lockstep fashion
  - AMD: 64 or 32 threads per wavefront
  - Intel: 8, 16 or 32 threads per wave
- Load/store operations for all threads must complete before next instruction issue
  - Can imply loading many cache lines for a single instruction
  - Slowing down the processing
- Common scenario: each thread processes one structure instance from an array
- Best performance achieved when all concurrent threads variables adjacent in memory
- Reorganize data in structures of arrays (SoA), with cache-line aligned arrays
- SoA ensures minimal memory access operations by collocating homologous variables from each structure instance into cache aligned columns in memory.

Reference AoS code:

```
struct AoS {
  static const size_t SIZE = 54;
  struct Element {
    double x, y, z;
    uint32_t id;
    Eigen::Matrix<double, 3, 6> m;
  };
  Element elements[SIZE];
  double r;
};
```

```
AoS aos;
const Eigen::Matrix<double, 3, 6> matrix{
  {1, 2, 3, 4, 5, 6},
  {2, 4, 6, 8, 10, 12},
  {3, 6, 9, 12, 15, 18}};

for (uint32_t i = 0; i < AoS::SIZE; i++) {
  if (i == 0)
    aos.r = 1.0;
  aos.elements[i]  =
    { 0., 0., 0., i, matrix * i};
}
```

Pre-existing implementations of SoA in CMSSW

- SoA use widespread in CMSSW [1]
- Multiple ad-hoc implementations
- Either compile time or run time sized
- Needing a single or multiple memory allocations and likewise host-device memory transfers
- Using coherence cache hinting primitives manually

Generic SoA

- Automates definition and implementation of runtime sized SoA
- Hierarchy of objects:
- `Layout` divides a buffer into runtime sized columns
- `View` is the interface to the data. Lightweight, this is the structure passed to kernels
  - Just pointers to columns, size
- `Buffers` host memory, pinned host memory or device memory, allocated from the framework (CUDA or Alpaka)
- `PortableCollection` wraps the `Layout` and `View`
  - Manages the buffer allocation
  - Provides an interface for memory transfers
  - Manages serialization to ROOT files

Generic SoA code example

- C++ statically typed: code generation before compile time with macros
- Based on `Boost::PP` [3]
- Structure definition:

```
namespace portabletest {
  // the typedef is needed because commas confuse macros
  using Matrix = Eigen::Matrix<double, 3, 6>;

  // SoA layout with x, y, z, id, m fields
  GENERATE_SOA_LAYOUT(TestSoALayout,
                      // columns: one value per element
                      SOA_COLUMN(double, x),
                      SOA_COLUMN(double, y),
                      SOA_COLUMN(double, z),
                      SOA_COLUMN(int32_t, id),
                      // scalars: one value for the whole structure
                      SOA_SCALAR(double, r),
                      // Eigen columns
                      SOA_EIGEN_COLUMN(Matrix, m))
  using TestSoA = TestSoALayout<>;
}  // namespace portabletest
```

- Syntax close to AoS (Here in CUDA kernel, note added '`operator()`' call)

```
static __global__ void testAlgoKernel(portabletest::TestDeviceCollection::View view,
   int32_t size) {
  const int32_t thread = blockIdx.x * blockDim.x + threadIdx.x;
  const int32_t stride = blockDim.x * gridDim.x;
  const portabletest::Matrix
    matrix{{1, 2, 3, 4, 5, 6}, {2, 4, 6, 8, 10, 12}, {3, 6, 9, 12, 15, 18}};

  if (thread == 0) {
    view.r() = 1.;
  }
  for (auto i = thread; i < size; i += stride) {
    view[i] = {0., 0., 0., i, matrix * i};
  }
}
```

- Other ways to use the SoA (reference to 'row'):

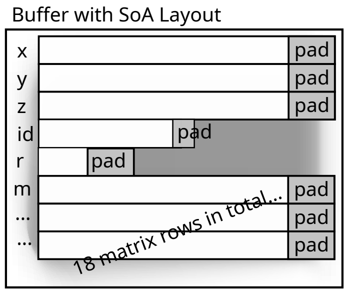
```
  auto vi = view[i];
  vi.x() = vi.y() = view[i].z() = 0.;
```

Access hinting, switchable range checking

- Generic SoA generates templates
- Defaults fit common use cases

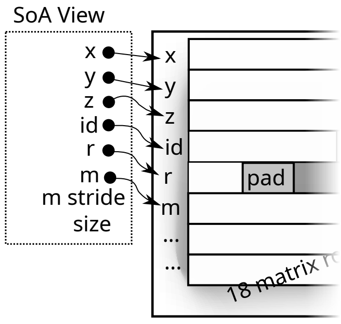
```
template < std::size_t ALIGNMENT = cms::soa::CacheLineSize::defaultSize,
           bool ALIGNMENT_ENFORCEMENT = cms::soa::AlignmentEnforcement::relaxed>
  struct Layout;

template < std::size_t VIEW_ALIGNMENT = cms::soa::CacheLineSize::defaultSize,
           bool VIEW_ALIGNMENT_ENFORCEMENT = cms::soa::AlignmentEnforcement::relaxed,
           bool RESTRICT_QUALIFY = cms::soa::RestrictQualify::enabled,
           bool RANGE_CHECKING = cms::soa::RangeChecking::disabled>
  struct View;
```

SoA Layouts in memory



- Layout helper function computes necessary buffer size from the number of elements
- Layout computes column addresses at construction time
- Add padding at end of columns for cache line alignment
- Also computes auxiliary strides and sizes for Eigen and serialization
- Cache line size configurable at compile time

SoA Views



- Minimal memory footprint to minimize kernel launch parameters size
  - one pointer per column
  - size (number of elements)
  - stride size for Eigen elements
- View is the data access interface
- Const variant of the class
- Constructor variant taking per-column bare pointers
  - Map existing structure to View interface
  - Used during porting to SoA
- Layout classes provide automatically corresponding trivial View
- Views can be redefined independently
  - View elements can refer to elements from any number of layouts and views
- Optional cache tuning, alignment checking
- Optional range checking

Portable collections: buffer management

- Available for Alpaka and CUDA
  - CMS is currently moving from CUDA to Alpaka [5] [6] [4]
- Host and device side
- Handles buffer allocation, layout creation and view mapping
- Host side allocation, view passed to device
- Direct buffer / layout access for memcpy between host and device

Portable collections: function

```
using TestDeviceCollection = cms::cuda::PortableDeviceCollection<portabletest::TestSoA>;

TestDeviceCollection deviceProduct(size_, ctx.stream()};

testAlgoKernel(deviceProduct.view(), deviceProduct->metadata().size());

cudatest::TestHostCollection hostProduct{size_, ctx.stream()};

cms::cuda::copyAsync(hostProduct.buffer(), deviceProduct.const_buffer(),
  deviceProduct.bufferSize(), ctx.stream());
```

ROOT and serialization/persistency

- Serialization to ROOT [2] files straightforward
- Layout can be written and read back to/from ROOT file
- Custom streamer automatically generated to read into pinned host memory
- Class description for Layout and Portable collection dictionary in XML file

```
<lcgdict>
  <class name="portabletest::TestSoA">
    <field name="mem_" comment="!"/>
    <field name="byteSize_" comment="!"/>
    <field name="x_" comment="[elements_]"/>
    <field name="y_" comment="[elements_]"/>
    <field name="z_" comment="[elements_]"/>
    <field name="id_" comment="[elements_]"/>
    <field name="m_" comment="[mElementsWithPadding_]"/>
    <field name="r_" comment="[scalar_]"/>
  </class>
  <class name="portabletest::TestSoA::View"/>
</lcgdict>
```

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

Outcome in practice

- `SiPixelDigis` and `SiPixelClusters` ported to SoA
- Device allocation reduced by 4 per event and host by 1
- Per-column allocation and hand split buffer both replaced with generic SoA

References

[1] CMSSW on GitHub. https://cms-sw.github.io/.

[2] The Boost Library Preprocessor Subset for C/C++. https://www.boost.org/doc/libs/1_67_0/libs/preprocessor/doc/index.html.

[3] Benjamin Worpitz. Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures, Sep 2015.

[4] Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E. Nagel, and Michael Bussmann. Alpaka - an abstraction library for parallel kernel acceleration. IEEE Computer Society, May 2016.

[5] A. Matthes, R. Widera, E. Zenker, B. Worpitz, A. Huebl, and M. Bussmann. Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the alpaka library. Jun 2017.

[6] ROOT Data analysis framework. https://root.cern.ch//.

Conclusion and further developments

- Previously scattered SoA handling knowledge consolidated in a single package
- Repetitive tasks for SoA structures automated
  - Fully for C++ code
  - Still in development for dictionary XML
- Additional features considered
  - Extra arrays of different size
  - Per track hits collection
  - Histograms