Implementation of generic SoA data structures in the CMS software

Eric Cano and Andrea Bocci for the CMS collaboration
CERN, European Organization for Nuclear Research, Meyrin, Switzerland
E-mail: eric.cano@cern.ch

Abstract. GPU applications require a structure of array (SoA) layout for the data to achieve good memory access performance. During the development of the CMS Pixel reconstruction for GPUs, the Patatrack developers crafted various techniques to optimise the data placement in memory and its access inside GPU kernels. The work presented here gathers, automates and extends those patterns, and offers a simplified and consistent programming interface.

The work automates the creation of SoA structures, fulfilling technical requirements like cache line alignment, while optionally providing alignment and cache hinting to the compiler and range checking. Protection of read-only products of the CMS software framework (CMSSW) is also ensured with constant versions of the SoA. A compact description of the SoA is provided to minimize the size of data passed to GPU kernels. Finally, the user interface is designed to be as simple as possible, providing an AoS-like semantic allowing compact and readable notation in the code.

1. Data layout in GPUs: Array of structure vs Structure of Arrays

Compared to CPUs, graphical processing units (GPUs) provide vast amounts of processing power by trading scheduling silicon real estate with arithmetic and logic unit (ALU) space, allowing many computations — in the order of thousands — to be executed in parallel. This trade-off is achieved with multiple ALUs executing the same instruction at the same time on their respective threads, in a lockstep fashion.

The width of this parallel execution — and naming — varies from manufacturer to manufacturer: 8, 16 or 32 threads per wave for Intel, 32 or 64 threads per wavefront for AMD and 32 threads per warp for NVIDIA. Like on CPUs where the various cores share the memory subsystem, the GPU cores executing the lock stepped threads share the same memory controller and cache, easily overloading it if accessing data scattered over many cache lines; the spread of memory accesses should be minimized by adequately laying out the data in memory. In many cases, memory access is the limiting factor for performance.

In a common scenario where each thread processes an instance of a structure, the usual strategy consists in reorganizing classic arrays of structures (AoS) into structures of arrays (SoA) where corresponding elements of successive structures are stored contiguously in cache-aligned columns, as illustrated in figure 1.

While the physical layout in memory is optimized for parallel processing, the per thread logic remains that of an AoS as shown in figure 2.

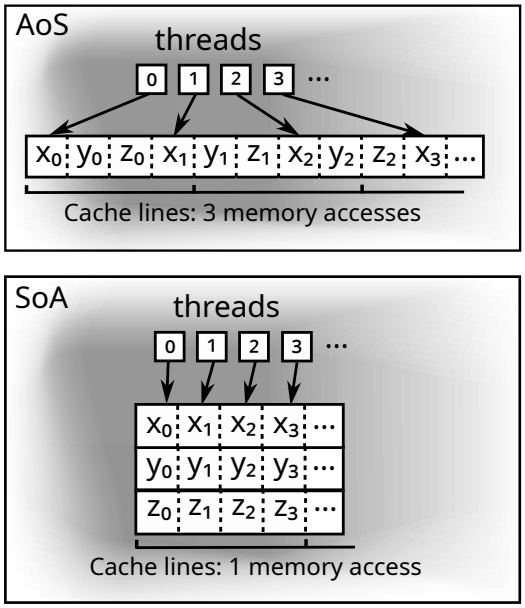

Figure 1. AoS vs SoA access patterns

```
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};
}
```

Figure 2. AoS C++ code

2. Pre-existing implementations of SoA in CMSSW

Prior to the work described here, SoAs were already in place in the CMSSW code in multiple, ad-hoc implementations. Some have compilation time defined sizes, while others are sized at run time; some used multiple memory allocations for each SoA, consequently requiring multiple memory transfers between host and device. The primitives hinting the compiler for cache type choice were also inserted directly in the using code.

3. Generic SoA and managing class hierarchy

The generic SoA described here automates the definition and implementation of runtime sized SoAs, and automatically generates a hierarchy of classes which handle different aspects of the SoA. `Layout`s divide a memory buffer into runtime sized columns, while `View`s provide the interface to the data. The latter are the lightweight structures passed to kernels. They are limited to a pointer for each column, and some sizes. `Buffer`s can be allocated on host memory, pinned host memory or device memory. The `Layout` is memory type agnostic and will subdivide any type of `Buffer`s indifferently.

SoA structures replaced those manually defined by GPU experts during the initial stages of GPU support addition to CMSSW. Column access functions are now automatically defined with optimization. Memory layout is defined over an unified buffer instead of per-column ones. SoA templates will allow subdetector field experts to generate optimized structures automatically. As an example, the SoA template version definition of a 4 columns structure takes 8 lines in one place, replacing 35 lines over multiple files.

Data transfers from host to device and vice-versa are implemented with full `Buffer` copies. On top of this hierarchy of classes stands the `PortableCollection`s; they handle the allocation of buffers of the proper size and the initialization of the `Layout` on top of the former. The host flavor of `PortableCollection` also manages the serialization and deserialization of data between memory and ROOT files.

4. Technical implementation

The generic SoA supports three types of elements: numeric columns, scalars and Eigen [2] columns. The numeric columns are targeted at numeric types, but can functionally accommodate other classes, with potential performance side effects. Those columns contain as many elements as the SoA does. The scalars hold a single element per SoA and are not available via row access. The last type, Eigen columns can hold vectors and matrices, with one numeric column per vector or matrix entry. The resulting memory layout is illustrated by figure 3

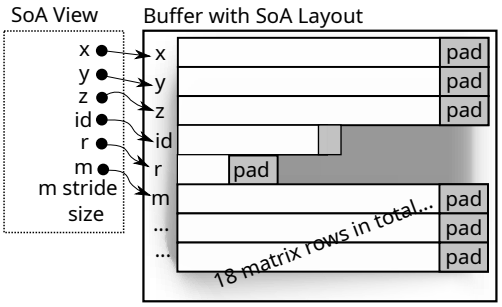

Figure 3. Buffer with SoA Layout and View

The generic SoA implementation targets a notation as concise and readable as possible, keeping the AoS syntax style, as it better represents the problem. Unlike Python, C++ is statically typed: code generation has to happen before compile time. Therefore generic SoAs are implemented in macros leveraging the `Boost::PP` [1] package.

An example of SoA layout declaration is shown on figure 4, with 4 individual columns (lines 9-12), a single scalar value (line 9) and an Eigen matrix (lines 4 and 17). Their use in a CUDA kernel is shown figure 5 with a full row initialization (columns and Eigen matrix) on line 18, and some column values computations on line 22.

The logic rows are accessed with `operator[]`, and can be stored in lightweight proxy variables (line 21) to allow concise notation as illustrated by the first 2 operands of line 22.

```
1  namespace portabletest {
2    // this typedef is needed because commas
3    // confuse macros
4    using Matrix = Eigen::Matrix<double, 3, 6>;
5
6    // SoA layout with x, y, z, id, m fields
7    GENERATE_SOA_LAYOUT(TestSoALayout,
8      // columns: one value per element
9      SOA_COLUMN(double, x),
10     SOA_COLUMN(double, y),
11     SOA_COLUMN(double, z),
12     SOA_COLUMN(int32_t, id),
13     // scalars: one value for the
14     // whole structure
15     SOA_SCALAR(double, r),
16     // Eigen columns
17     SOA_EIGEN_COLUMN(Matrix, m))
18   using TestSoA = TestSoALayout<>;
19 }  // namespace portabletest
```

Figure 4. SoA declaration

```
1  static __global__ void testAlgoKernel(
2     portabletest::TestSoA::View view,
3     int32_t size) {
4    const int32_t thread = blockIdx.x *
5      blockDim.x + threadIdx.x;
6    const int32_t stride = blockDim.x *
7      gridDim.x;
8    const portabletest::Matrix
9      matrix{{1, 2, 3, 4, 5, 6},
10            {2, 4, 6, 8, 10, 12},
11            {3, 6, 9, 12, 15, 18}};
12
13   if (thread == 0) {
14     view.r() = 1.;
15   }
16   for (auto i = thread; i < size;
17     i += stride) {
18     view[i] = {0., 0., 0., i, matrix * i};
19     // Alternate ways to access the rows,
20     // member by member
21     auto vi = view[i];
22     vi.x() = vi.y() = view[i].z() = 0.;
23   }
24 }
```

Figure 5. SoA use in a CUDA kernel

4.1. Layout class: memory management

The necessary `Buffer` size can be computed by a static helper function from the `Layout` class. The columns layout inside the buffer is computed by the constructor of the `Layout` class; it will divide the `Buffer` by computing the column addresses, adding padding at the end of columns if necessary to ensure cache line alignment. Additionally, stride length and total size is computed for Eigen columns, to be used by Eigen itself and serialization, respectively.

4.2. View class: data access

The `View` class is designed to contain the minimal necessary variables to ensure data access, and the corresponding functions. This minimal memory footprint ensures efficient kernel launches. The `View` contains the size of the columns, one pointer to each of them or scalar, plus the stride of the Eigen columns — avoiding any size computation kernel side.

The `View` — the interface to the data — provides a logic row accessor in the form of `operator[]`. The row object provides accessors to column components for the selected row. `operator[]` optionally range checks indices. The accessors to scalar components are direct members of the `View`. Likewise, extra column accessors provide pointers to each column component.

A `const` variant of the class is available, ensuring that products consumed by CMSSW modules remain immutable. This class is distinct, completely forbidding write access even via `const` casting.

Another constructor variant, useful for adapting an existing code base, allows per column initialization, bypassing buffer splitting and providing the SoA interface without buffer allocation.

The `View` classes can be defined with any set of components from multiple `Layout`s and `View`s. Nevertheless, the most common use of the `View` is the trivial one, an automatically generated `View` which provides access to every component of the `Layout`.

The constructor of the `View` can optionally validate column alignment. All non-Eigen accessors provide optional compiler cache hinting, ensuring use of the non-coherent cache on nVidia GPUs and similar optimizations in other environments.

4.3. Cache hinting, range checking and other tunable behaviors

Optional behaviors of `Layouts` and `Views` are selected at compilation time as template parameters. The defaults are usually the right choice, and most common setup. The parameters and their defaults are shown in figure 6.

```
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;
```

Figure 6. SoA template parameters and defaults

5. Portable collections: buffer management

`PortableCollection`s are templates parametrized on `Layouts`. They manage the allocation of `Buffers` and the creation of `Layouts` and `Views`. As CMS is currently moving from CUDA to Alpaka [3] [4] [5], `PortableCollection`s exist in multiple flavors: both for CUDA and Alpaka, and with variants for host side and devices in each case. All versions handle the allocation of the corresponding buffers. The `PortableCollection` then provides access to the buffers for data transfers between host and device, the views for data access, and can be serialized to and from ROOT files [6] — for the host use case. A CUDA code example is illustrated in figure 7 and the necessary ROOT streaming declaration in figure 8

The ROOT data files generated from the `PortableCollection`s can be readily used in bare ROOT, but the default memory layout used by ROOT when reading them back is not

```
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());
```

Figure 7. Instanciation and device-to-host copy of a portable collection

appropriate for use with GPUs. Optimal ROOT serialization is achieved with automatically generated functions that ensure proper memory allocation and data placement at read time. The streamer of the `PortableCollection` allocate the memory and delegates the copying of the data from the ROOT file into the memory columns to the `Layout` streamer.

```
<lcgdict>
  <class name="portabletest::TestHostCollection"/>
  <read sourceClass="portabletest::TestHostCollection"
        targetClass="portabletest::TestHostCollection"
        version="[1-]" source="portabletest::TestSoALayout<128, false> layout_;"
        target="buffer_,layout_,view_" embed="false">
  <![CDATA[
    portabletest::TestHostCollection::ROOTReadStreamer(newObj, onfile.layout_);
  ]]>
  </read>
  <class name="edm::Wrapper<portabletest::TestHostCollection>" splitLevel="0"/>
</lcgdict>
```

Figure 8. Layout and streamer declaration for ROOT streaming

6. Conclusion, status and further developments

So far, the pixel local reconstruction has been ported to this generic SoA approach. Systematic use of a generic SoA reduced memory allocation number, and simplified code. The previously scattered SoA knowledge is now consolidated in a single package and its use automated. Some manual XML description of the data structures are still necessary, automation of this step is under development.

Multi layout data collections are also in the works. They will allow keeping in the same product sets of related data of different sizes, like tracks and hits with cross reference by index.

Sub buffer, column level access is also investigated to optimize some use cases.

7. References

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

[2] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[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//.