SOAContainer

Manuel Schiller

University of Glasgow

July 18th, 2018

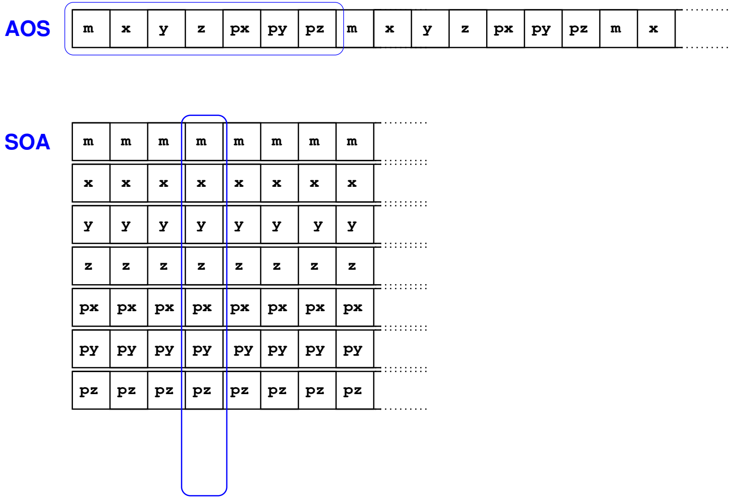
outline

- AOS and SOA
- `SOAContainer` framework
  - how code looks
  - fields and skins
  - views
  - zipping views
- summary

SOA basics

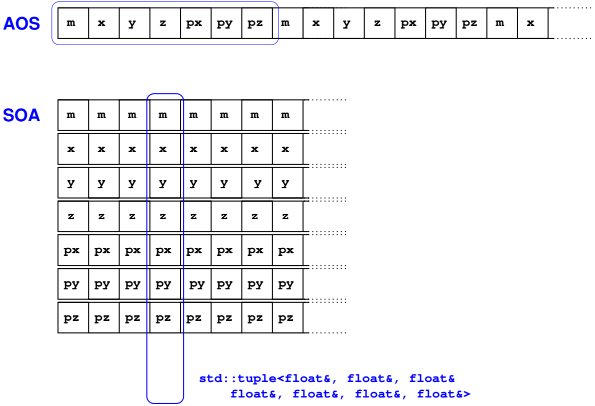
SOA basics

AOS and SOA



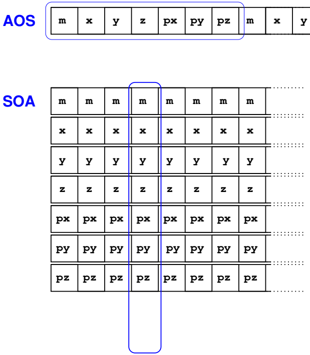
- SOA has advantages over AOS when processing vectors of objects
- CPU and compiler like to vectorise SOA, cache loves it!

SOA introduction



- SOA doesn’t have objects in memory
- it has bits of objects scattered in memory
- tuples of references model this quite naturally
- but user interface is terrible…

SOA implementation

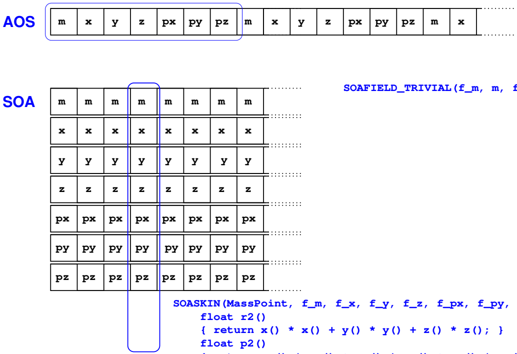



```
SOAFIELD_TRIVIAL(f_m, m, float);

struct f_m {
    // ...
    float& x() { /* ... */ }
    const float& m() { /* ... */ }
    // ...
};
```

- use *fields* to describe
  - type
  - name of getters/setters
  - other operations on single data item

SOA implementation




```
SOAFIELD_TRIVIAL(f_m, m, float);
```

```
SOASKIN(MassPoint, f_m, f_x, f_y, f_z, f_px, f_py, f_pz) {
    float r2()
    { return x() * x() + y() * y() + z() * z(); }
    float p2()
    { return px() * px() + py() * py() + pz() * pz(); }
};
```

- use a *skin* to describe
  - which fields make up an SOA object
  - give the underlying tuple a nice interface
  - gets all the methods defined in the fields
  - can define methods that access more than one field

aims

- exploiting modern CPUs/GPUs is hard enough
  - `SOAContainer` should help you try SOA code quickly, with low barrier of entry
  - code should be readable, and do what it says
  - writable by experts and non-experts alike
  - produce reasonable code to start with (profile, optimize later)
- will not magically solve your performance problems
  - brain not outdated yet: need good idea and right algorithm
  - may still want to look at assembly to judge if compiler needs help (simpler loops, break dependencies, ...)
  - compilers may not autovectorise the way you want them to

→ let's see how `SOAContainer` looks in practise!

SOA examples

SOA examples

trivial *fields* and *skins*

first step: define data (members)

```
namespace MyPoint {
    // a field struct x: type float, accessors: (const) float& x() (const)
    SOAFIELD_TRIVIAL(x, x, float); // field name, accessor name, type
    SOAFIELD_TRIVIAL(y, y, float); // same for y, z
    SOAFIELD_TRIVIAL(z, z, float);
    // make a skin Point, with fields x, y, z
    SOASKIN_TRIVIAL(PointSkin, x, y, z);
}
```

- defining trivial *fields* and *skins* is easy
  - trivial fields just have getters/setters
  - trivial skins are just the sum of their fields
- both fields and skins are *types*
  (although you don't really want to read their definitions)
- the skin keeps a list of member fields
- it is a good idea to keep a skin and its fields in a common namespace

using SOA::Container

```
namespace MyPoint {
    // fields - you saw these on the last slide ...
    SOASKIN_TRIVIAL(PointSkin, x, y, z);
}

using namespace MyPoint;
// define a container - std::vector as underlying storage
SOA::Container<std::vector, PointSkin> c;
c.reserve(10);
// fill the container with sth.
for (unsigned i = 0; i < 10; ++i)
    c.emplace_back(1.f * i, 2.f * i, 3.f * i); // x, y, z
// sort
std::sort(c.begin(), c.end(),
    [] (const auto& a, const auto& b) { return a.x() < b.x(); });
// print
for (auto el: c) {
    std::cout << "point (" << el.x() << ", " << el.y() << ", " <<
        el.z() << ")" << std::endl;
}
c.clear();
std::cout << "c is " << (c.empty() ? "empty" : "full") << std::endl;
```

- `SOA::Container` has the interface of `std::vector`
    - you know how to use it

complex *fields*

- sometimes, having just a getter and a setter isn't enough

```
namespace MyHit {
    // fields ...
    SOAFIELD(StripNo, unsigned, // field name, type
        SOAFIELD_ACCESSORS(stripNo); // getters and setters named stripNo
        // we define our own methods:
        // apparently, odd strips 768 and above are noisy
        bool isNoisy() const noexcept
        { return (this->stripNo() >= 768) && (this->stripNo() & 1); }
    );
    // ... more fields ...
}
```

- the SOAFIELD macro does the heavy lifting
- SOAFIELD_ACCESSORS(accname) reduces typing for standard getters/setters
  - you can of course also code your own:

```
namespace MyHit {
    SOAFIELD(StripNo, unsigned, // field name, type
        type stripNo() const noexcept { return this->_get(); }
        void setStripNo(unsigned strip) noexcept { this->_get() = strip; }
        bool isNoisy() const noexcept
        { return (this->stripNo() >= 768) && (this->stripNo() & 1); }
    );
}
```

complex *skins*

- trivial skins are just the sum of their fields' methods
- sometimes, you need more than that:

```
namespace Point {
    SOAFIELD_TRIVIAL(x, x, float);
    SOAFIELD_TRIVIAL(y, y, float);
    struct rphi_tag {};
    SOASKIN(Skin, x, y) {
        // inherit default constructors, assignment operators from underlying tuple
        SOASKIN_INHERIT_DEFAULT_METHODS(Skin); // name of skin class
        // our own constructors
        Skin(rphi_tag, float r, float phi) :
            Skin(r * std::cos(phi), r * std::sin(phi))
        {}
        // our extra methods
        float r2() const noexcept
        { return this->x() * this->x() + this->y() * this->y(); }
        float r() const noexcept { return std::sqrt(r2()); }
        float phi() const noexcept
        { return std::atan2(this->y(), this->x()); }
    };
}
```

- *complex skins* like the one above let you do pretty much anything you want...

views and zips

views and zips

views (1/2)

- a *view* is a fixed size sequence of (some fields of) a container
  - every SOA container is also a view
  - think of SOA container without `push_back`, `emplace_back`, `insert`, `erase`, `clear`
  - you can take a container/view and extract only some fields

```
namespace MyPoint {
    SOAFIELD_TRIVIAL(x, x, float);
    SOAFIELD_TRIVIAL(y, y, float);
    SOAFIELD_TRIVIAL(z, z, float);
    SOASKIN(Skin, x, y, z) {
        SOASKIN_INHERIT_DEFAULT_METHODS(Skin);
        float r() const noexcept
        {
            return std::sqrt(this->x() * this->x() + this->y() * this->y() +
                this->z() * this->z());
        }
    };
}
SOAContainer<std::vector, MyPoint::Skin> c = /* from somewhere */
auto xyview = c.view<MyPoint::x, MyPoint::y>();
auto mul0xz = xyview[0].x() * xyview.y();
```

  - operation is cheap: copy 2 iterators per field
  - by default, view gets trivial skin, i.e. sum-of-fields

views (2/2)

- by default, view gets trivial skin, i.e. sum-of-fields
- if you want things to be more clever, you can specify your own skin

```
namespace MyPoint {
    SOAFIELD_TRIVIAL(x, x, float);
    SOAFIELD_TRIVIAL(y, y, float);
    // see last slide for details
}
namespace MyPoint2D {
    SOASKIN(Skin, MyPoint::x, MyPoint::y) {
        SOASKIN_INHERIT_DEFAULT_METHODS(Skin);
        float r() const noexcept
        { return std::sqrt(this->x() * this->x() + this->y() * this->y()); }
    };
}
SOAContainer<std::vector, MyPoint::Skin> c = /* from somewhere */
auto fancyxyview = c.view<MyPoint2D::Skin>();
auto el0r = fancyxyview[0].r();
```

vectorised straight line (1/2)

- consider trivial example: straight line from points

```
namespace HitPair { // pair of hits in two layers
    SOAFIELD_TRIVIAL(xhit0, xhit0, float);
    SOAFIELD_TRIVIAL(zhit0, xhit0, float);
    SOAFIELD_TRIVIAL(xhit1, xhit1, float);
    SOAFIELD_TRIVIAL(zhit1, zhit1, float);
    SOASKIN_TRIVIAL(Skin, xhit0, zhit0, xhit1, zhit1);
}
namespace XZLine { // straight line
    SOAFIELD_TRIVIAL(x0, x0, float);
    SOAFIELD_TRIVIAL(z0, z0, float);
    SOAFIELD_TRIVIAL(tx, tx, float);
    SOASKIN(Skin, x0, z0, tx) {
        SOASKIN_INHERIT_DEFAULT_METHODS(Skin);
        Skin(float xhit0, float zhit0, float xhit1, float zhit1) noexcept :
            Skin(0.5f * (xhit0 + xhit1), 0.5f * (zhit0 + zhit1),
                (xhit1 - xhit0) / (zhit1 - zhit0))
        {}
        float x(float z) const noexcept
        { return (z - this->z0()) * this->tx() + this->x0(); }
    };
}
SOA::Container<std::vector, HitPairs::Skin> hitpairs = /* from somewhere */;
SOA::Container<std::vector, XZLine::Skin> xzlines;
xzlines.reserve(hitpairs.size());
for (auto el: hitpairs)
    xzlines.emplace_back(el.xhit0(), el.zhit0(), el.xhit1(), el.zhit1());
```

zips (2/2)

- in many cases, you have input vectors of (SOA) objects
- incremental calculation of the output

```
// ... see last slide for field and skin definitions ...
SOA::Container<std::vector, HitPairs::Skin> hitpairs = /* from somewhere */;
SOA::Container<std::vector, XZLine::Skin> xzlines;
xzlines.reserve(hitpairs.size());
for (auto el: hitpairs)
    xzlines.emplace_back(el.xhit0(), el.zhit0(), el.xhit1(), el.zhit1());
```

- ideally, want convenient (skinned) interface of both inputs and outputs together ("object composition"):

```
// zip together hit pairs with the resulting calculated line
auto xzstubs = zip(hitpairs, xzlines);
auto fs = xzstubs.front(); // first stub
std::cout << "first␣stub:␣(" << fs.xhit0() << "," << fs.zhit0() << ")-(" <<
    fs.xhit1() << "," << fs.zhit1() << ")␣=>␣z0=" << fs.z0() << "␣x0=" <<
    fs.x0() << "␣tx=" << fs.tx() << std::endl;
```

view and *zip* summary

- zip returns a skinned view of the underlying range
  - cheap operation: copies a pair of iterators per field
  - can take any number of inputs
  - all must be same length (or asserts)
  - no duplicate field types!
- *view* and *zip* allow for flexible vectorised compute kernels
  - compute new quantities based on the input
  - return a uniform view of result and (some of) input variables
- precisely what is needed in many pattern reco algorithms

- new views get the trivial skin composed of the included fields
- if you need something else, you can pass your own skin
- views of subranges work, too:

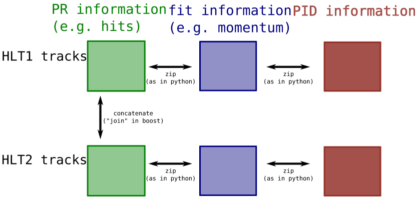
```
SOA::Container<std::vector, HitPairs::Skin> c = /* ... */;
// first eight hit0s:
auto first8hit0 = c.view<HitPairs:xhit0, HitPairs::xhit1>(c.begin(), c.begin() + 8);
```

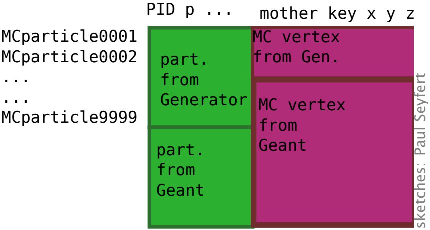
use cases in LHCb

- had hoped SOAContainer would have seen some adoption by now
  - a lot of colleagues tell me how important and useful it is…
  - … but when in doubt:
    - they work on something else,
    - or prefer to roll their own
- nevertheless, there are a couple of prototypes out there:
  - variant of Pixel tracking: aim to use SOAContainer to compare AOS and SOA; Renato Quagliani will add typedefs to make SOA variant as soon as he has time: AOSPixelTracking.cpp, AOSPixelTracking.h
  - I have my own standalone pixel tracking SOA prototype (with ideas from Rainer Schwemmer and Daniel Campora – I still a fair bit of development to do): velo-phi-drdz2
- I won't show algorithm code here, since it looks just like AOS code!

SOAContainer for upgrade event model

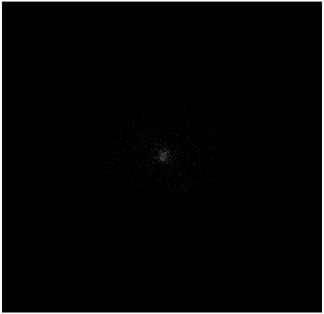
- people like SOAContainer's ability to `zip` for upgrade event model
- together with `boost::join`, it's a powerful mix to
  - aggregate information from different kinds computations in a single object
  - aggregate information from different origins in a shared collection





SOAContainer pointers

- there's finally a tutorial on the web
- some example code also on gitlab
- nice example: gravitational N-body simulation (think galaxy formation)



  - compute-heavy code
  - can switch between SOA and AOS with a typedef!
  - clang 5.0 autovectorises: full factor 4 speedup one can expect from my CPU!

summary

summary

summary

- SOAContainer framework is there, and ready for you to play with!
  - convenient SOA prototyping
  - almost "feels" like normal vector of structs
  - new objects can be composed on the fly ("zipping")

  → easy, structured and efficient compute kernels

- get the code, and start hacking today!
  - I'm happy to answer questions
  - in any case, I'd like to hear from you how it goes:
    `Manuel.Schiller@cern.ch`

interested in other cool stuff? kdtree, MMapVector, see backup!

backup

backup

MMapVector

- mmap makes OS map contents of a file/device into memory
  - swapping is implemented that way, OS loads executables with mmap
  - very good I/O performance (raw disk speeds, OS very well tuned!)
- MMapVector has familiar std::vector-like interface:
  - can be used for HUGE (larger than RAM) vectors – fast, invisible I/O

```
class Candidate { /* ... */ };
MMapVector<Candidate> hugevector; // backed by a temp. file
hugevector.reserve(10000000000ull);
for (uint64_t i = 0; i < 10000000000ull; ++i)
    hugevector.push_back(generate());
// do sth with it...
for (const auto& c: hugevector) doSth(c);
```

- π^0 calibration in LHCb (runs online) uses it since last year
  - iterative calibration: read same data over and over
  - then you may not want the overhead of ROOT's I/O
  - $>$ factor 5 faster: MMapVector plus better work distribution among HLT nodes

MMapVector example: π^0 calibration

- expensive part: I/O to calibrate each CALO cell in $\pi^0 \to \gamma\gamma$

```
#include <cmath>
#include <cassert>
#include "MMapVector.h"
#include "TH1D.h"
int main() {
    struct Candidate {
        double px1, py1, pz1, E1, px2, py2, pz2, E2;
        unsigned cell1, cell2;
    };
    const double* calibs = getCellCalibs();
    const unsigned mycellid = worker_get_my_cellid()
    MMapVector mmaptuple<Candidate>("/tmp/mytuple.mmap", MMapVectorBase::ReadOnly);
    TH1D hpimass("hpimass", "pimass", 1000, 1e2, 2e2);
    for (const auto& c: mmaptuple) { // expensive
        assert(mycellid == c.cell1 || mycellid == c.cell2);
        const auto E  = calibs[c.cell1] * c.E1  + calibs[c.cell2] * c.E2;
        const auto px = calibs[c.cell1] * c.px1 + calibs[c.cell2] * c.px2;
        const auto py = calibs[c.cell1] * c.py1 + calibs[c.cell2] * c.py2;
        const auto pz = calibs[c.cell1] * c.pz1 + calibs[c.cell2] * c.pz2;
        const auto m2 = E * E - px * px - py * py - pz * pz;
        hpimass.Fill(std::sqrt(m2));
    }
    fit_and_update_calib(mycellid, hpimass); // cheap
    return 0;
}
```

MMapVector example: convert ROOT tuple

- converting from a ROOT-style tuple is also straightforward:

```
#include <cstdint>
#include "TFile.h"
#include "TTree.h"
#include "MMapVector.h"
int main()
{
    TFile f("mytuple.root", "READ");
    TTree* t = (TTree*) f.Get("decaytree");
    struct Candidate {
        double px, py, pz, E;
    } c;
    t->SetBranchAddress("px", &c.px);
    t->SetBranchAddress("py", &c.py);
    t->SetBranchAddress("pz", &c.pz);
    t->SetBranchAddress("E", &c.E);
    MMapVector<Candidate> mmaptuple("/tmp/mytuple.mmap",
        MMapVectorBase::ReadWriteCreate);
    uint64_t ncands = t->GetNumEntries();
    mmaptuple.reserve(ncands);
    for (uint64_t i = 0; i < ncands; ++i) {
        t->GetEntry(i);
        mmaptuple.push_back(c);
    }
    return 0;
}
```