Bernhard Manfred Gruber
CERN, CASUS, HZDR, TU Dresden

Data layout and reflection in C++

C++ data layout portability workshop
November 30th, 2023

SPONSORED BY THE Federal Ministry of Education and Research

Data layout goes far beyond SoA

SoA is the next common thing after AoS, deserves its own solution

But there is more to data layout

Motivation: Single source performance portable code is challenging

Different hardware arch., data set, algorithms, threading, access patterns, etc. require different data layouts

Different data layouts require different indexing syntax

Changing data layout requires rewriting code

Ideally: data layout abstraction layer

Let's look at two case studies using LLAMA

LLAMA = Low Level Abstraction of Memory Access
— A C++17 library for data layout abstraction using TMP

Simplified HEP analysis on CPU: B2HHH decay using LHCb Open Data

EM article transport simulation on GPU: AdePT

```
data[i].x
data.x[i]
data[i/8].x[i%8]
data(i).x()
```

Indexing different data layouts.
Here: AoS, SoA, AoSoA8, CMS SoA

User data structure definition

```cpp
struct H1{} h1;
struct H2{} h2;
struct H3{} h3;
struct PX{} px;
struct PY{} py;
struct PZ{} pz;
struct ProbK{} probK;
struct ProbPi{} propPi;
struct IsMuon{} isMuon;
```

Business as usual for type-based template metaprogramming

```cpp
using H = llama::Record<
    llama::Field<PX, double>,
    llama::Field<PY, double>,
    llama::Field<PZ, double>,
    llama::Field<ProbK, double>,
    llama::Field<ProbPi, double>,
    llama::Field<IsMuon, int>>;

using Event = llama::Record<
    llama::Field<H1, H>,
    llama::Field<H2, H>,
    llama::Field<H3, H>>;
```

Data structure configuration and creation

```
auto extents = llama::ArrayExtentsDynamic<std::size_t, 1>{...};
auto mapping = ... Event ... extents ...;
auto allocator = ...;
auto accessor = ...;
auto view = llama::allocViewUninitialized(mapping, allocator, accessor);

runCompute(view);
```

Quite similar to std::mdspan

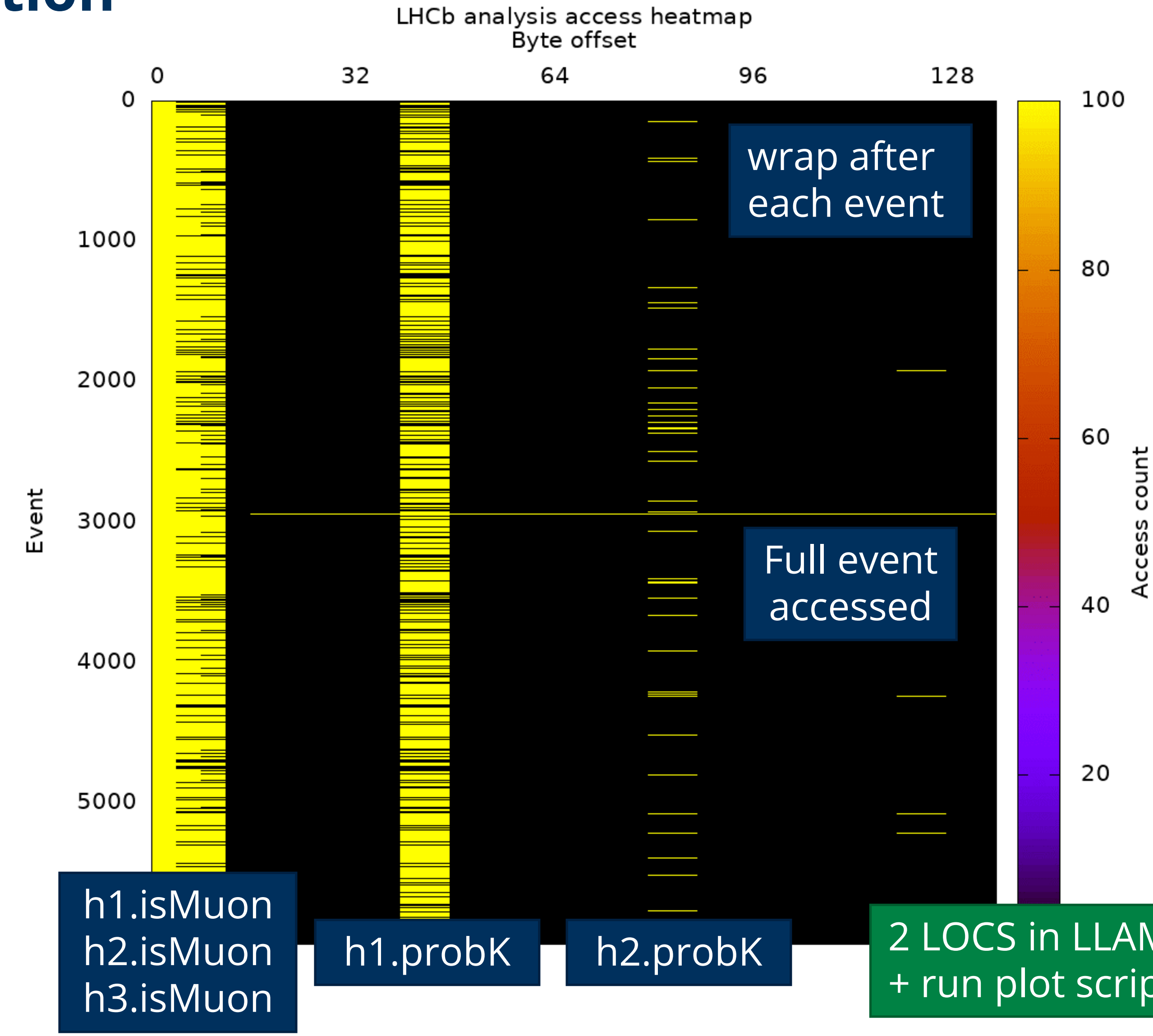
Access syntax and instrumentation

Proxy ref.

Array subscript

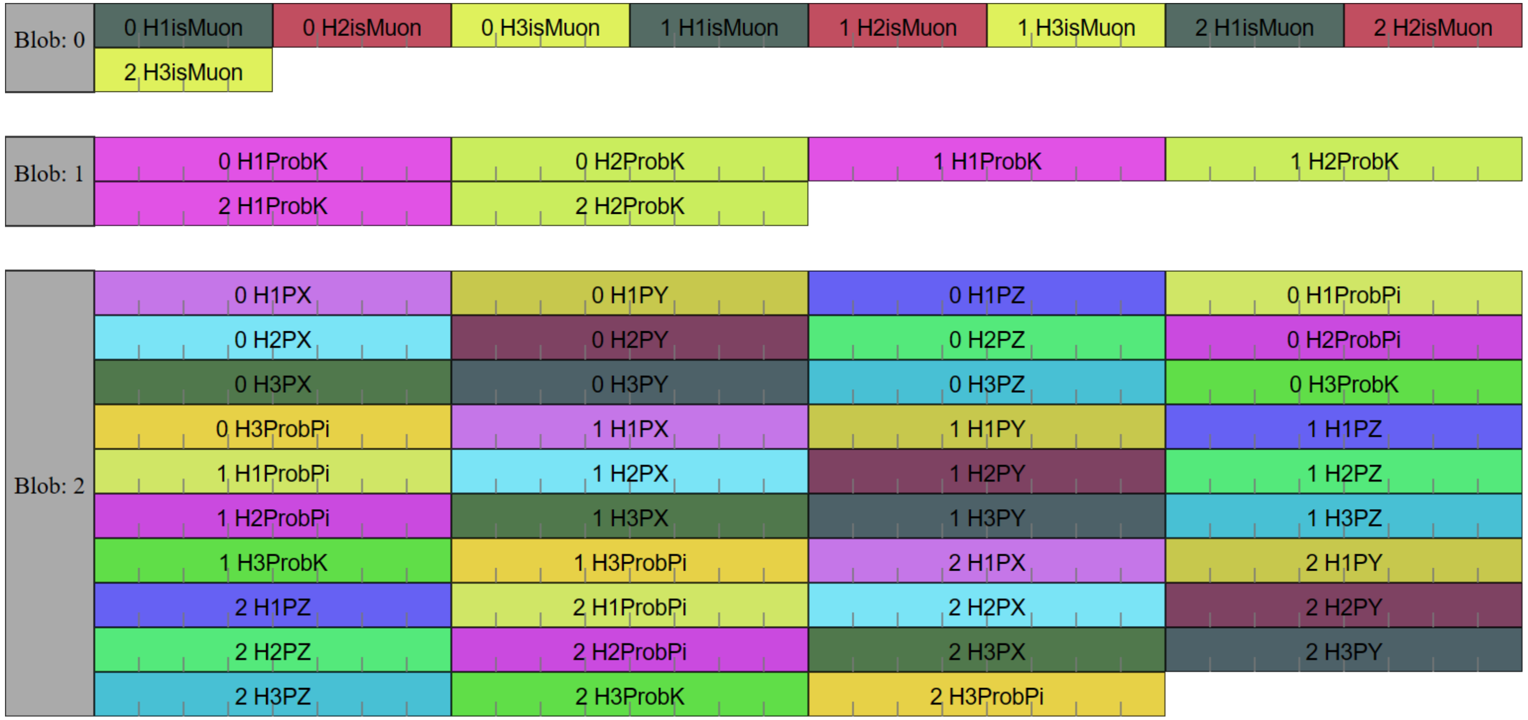
```
#pragma omp parallel for
for(size_t i = 0; i < n; i++) {
    auto&& event = view[i];

    if(event(h1)(isMuon)) continue;
    if(event(h2)(isMuon)) continue;
    if(event(h3)(isMuon)) continue;

    if(event(h1)(probK) < probKCut) continue;
    if(event(h2)(probK) < probKCut) continue;
    if(event(h3)(probK) < probKCut) continue;

    if(event(h1)(probPi) > probPiCut) continue;
    if(event(h2)(probPi) > probPiCut) continue;
    if(event(h3)(probPi) > probPiCut) continue;

    // compute bmass ...

    hists[omp_get_thread_num()].Fill(bmass);
}
```

Member access



2 LOCS in LLAMA + run plot script

The optimal layout



Using pure data layout

Separate h[1-3].isMuon into AoS

Separate h[1-2].probK into AoS

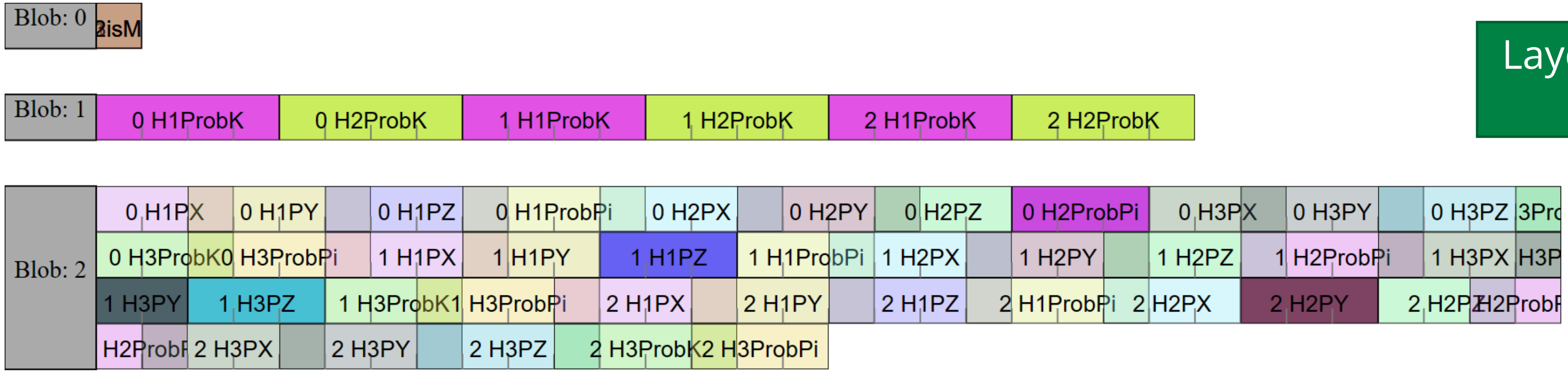
Remaining fields as AoS

With data reduction

Pack h[1-3].isMuon into 1 bit each

Change type of h[1-2].probK to float

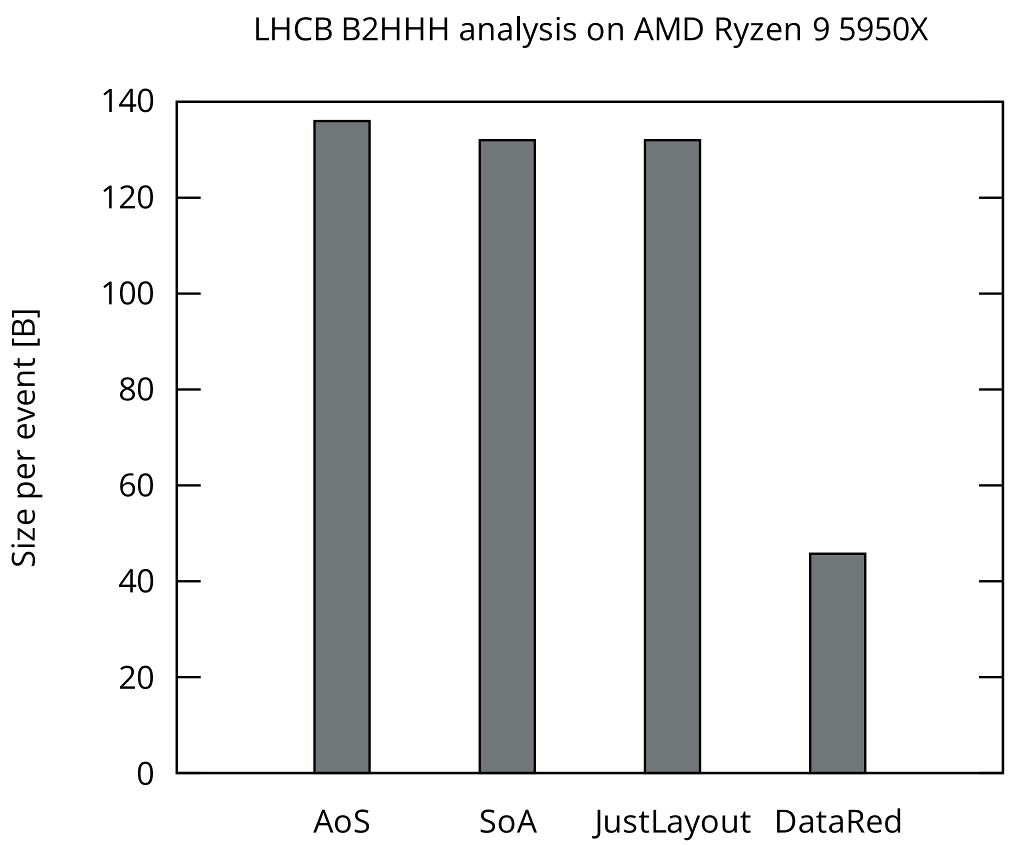
Bitpack the rest



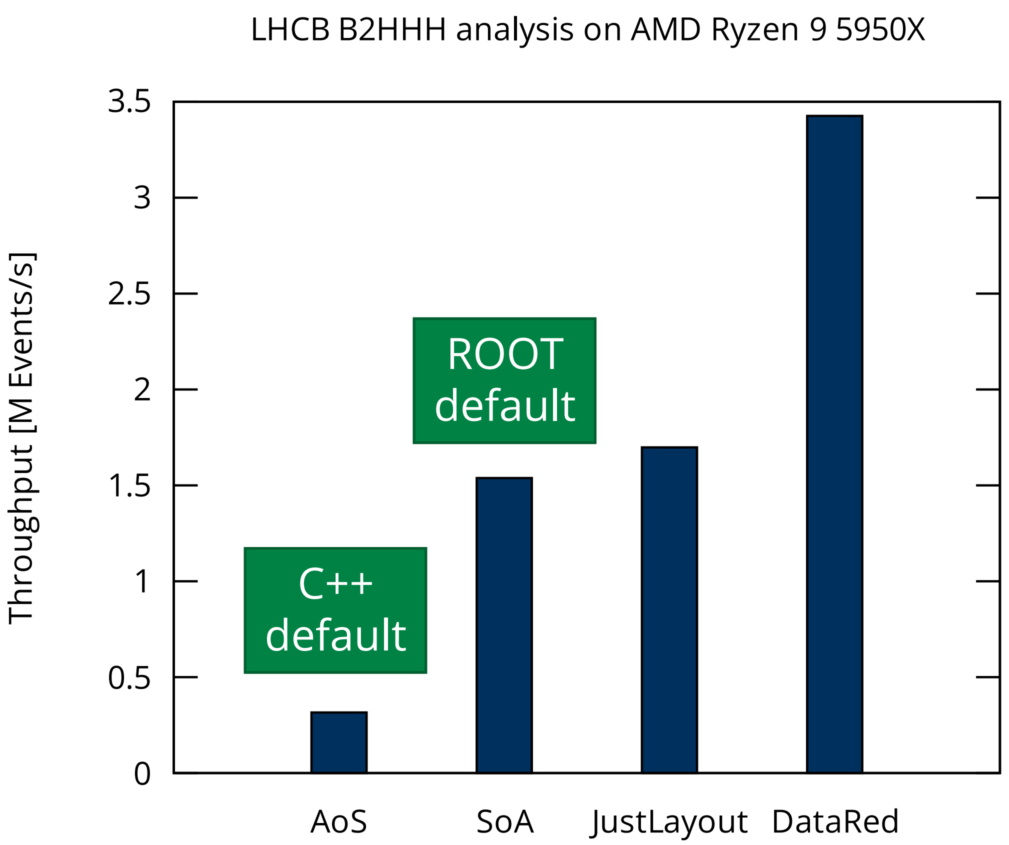
Layout visualization of 3 events
1 LOC with LLAMA

Benchmark

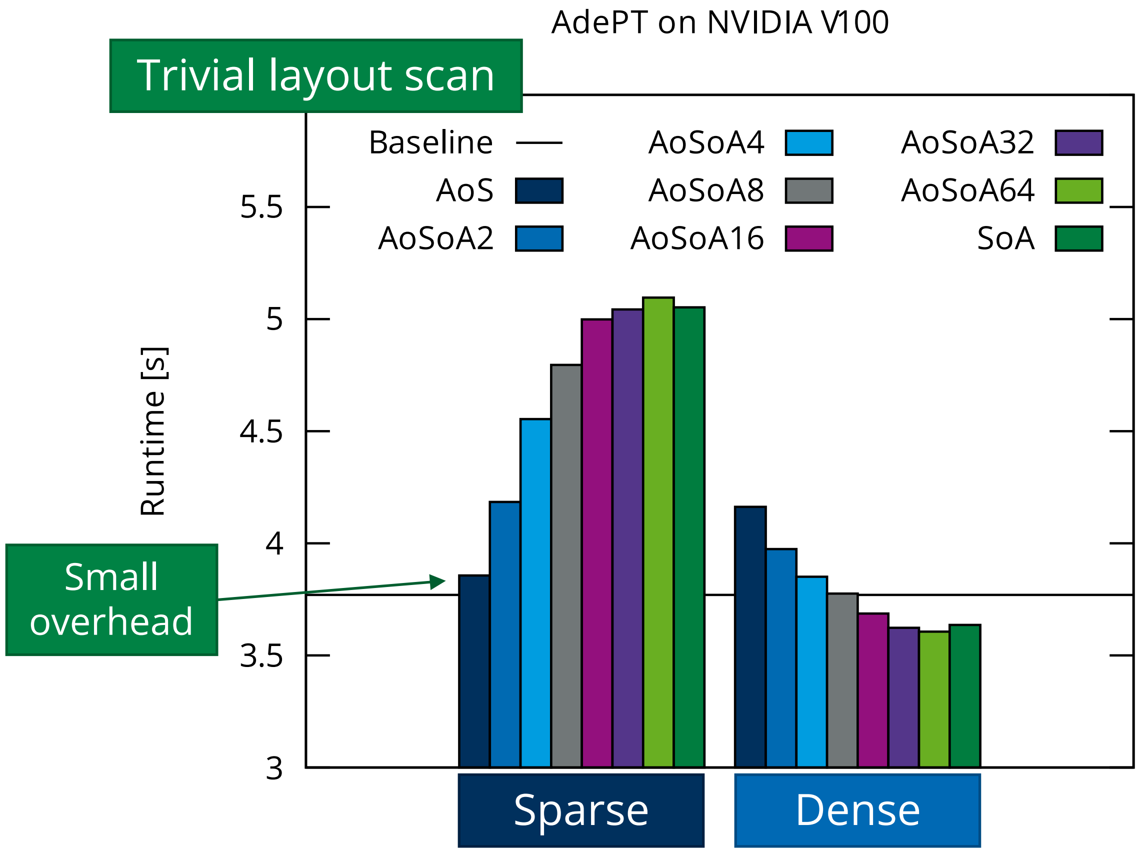
In-memory size



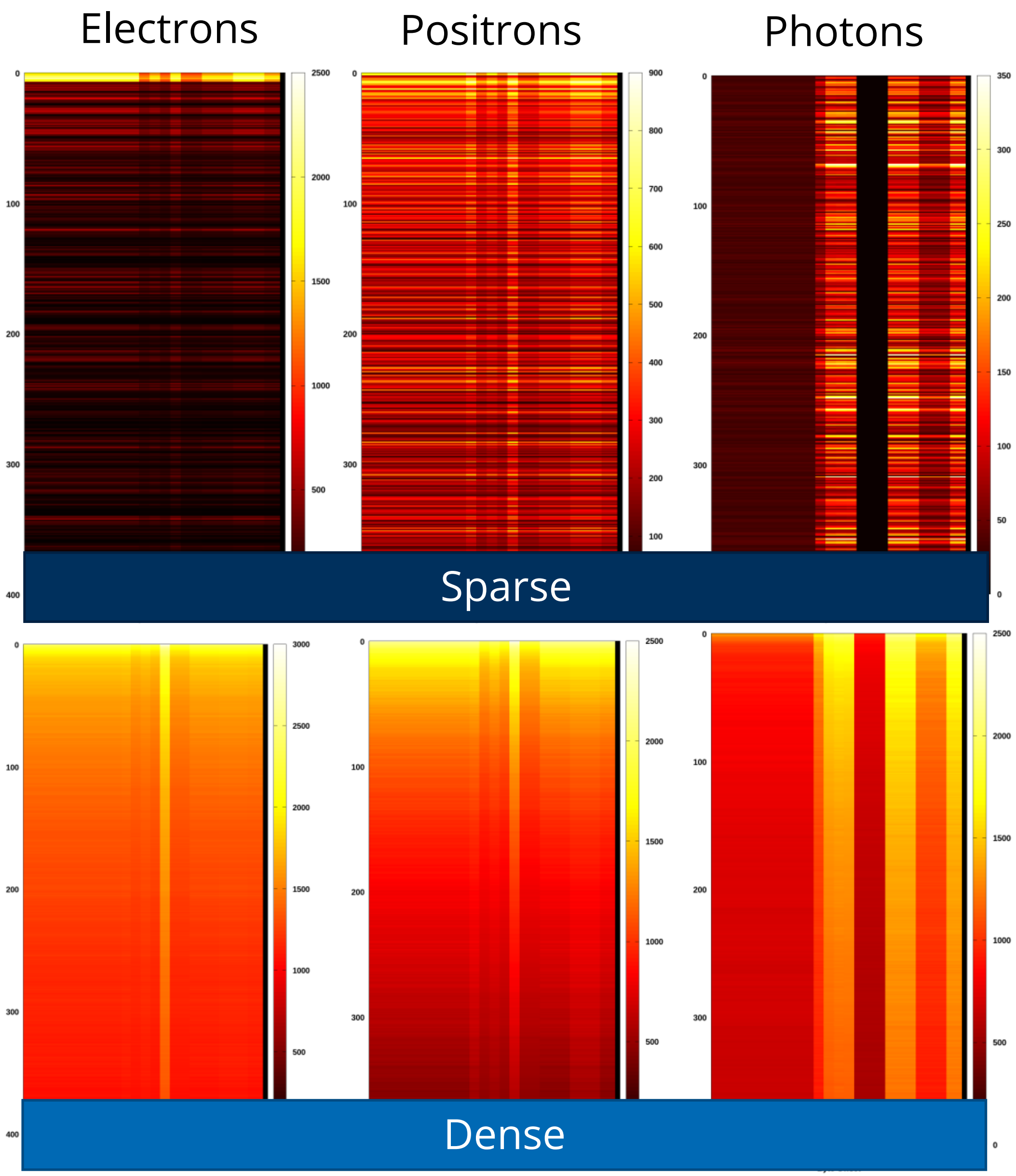
Throughput



Case study highlights: AdePT



Changing layout: edit 1 LOC, recompile, rerun



More on AdePT here, all results published here.

Full data layout abstraction

Requires:

Decoupling of data layouts from algorithms
- Layout independent data structure description (i.e. structs, arrays)
- Layout independent access (i.e. subscripting, member access, refs., ptrs., ...)

Language to express generic data layouts (eDSL?)
- Like mdspan's layouts + data members

Value types, reference types, iterators, pointer types?

Copy back and forth between different layouts

Decoupling of data layout from allocation, like mdspan

Uniform syntax, like view[i](h1)(probK)

Better: native C++ syntax, like view[i].h1.probK
- Impossible with TMP, requires macros or reflection

Enables:

Generic and layout-independent programming

Changing data layout is a 1LOC change
- Propagated via templates, like ranges/iterators/mdspan
- No modification of algorithms needed

Trivial data layout autotuning
- Data layouts become a parameter space

Systematic data layout performance engineering

Profiling data structures via instrumentation

Hardware-specific customization
- Switch to best layout on each architecture
- Use accessors, e.g. cache-bypassing stores

Heterogeneous performance portability
- In combination with e.g. Kokkos, alpaka, SYCL, ...

Reflection in C++ - the paper trail

Among first papers: N1775, A Case for Reflection, 2005

…

Type based: N3996, N4111, N4451, P0194, Static reflection, 2014-2018

Type based: P0385, Static reflection – Rationale, design and evolution, 2016-2017

Reflection TS (draft): N4856, 2020

Value-based: P0993, Value-based Reflection , 2018

Value-based typeful: P0953, constexpr reflexpr, 2017-2019

Value-based monotype: P1240, Scalable Reflection in C++, 2018-2022

Combination: P1733, User-friendly and Evolution-friendly Reflection: A Compromise, 2019

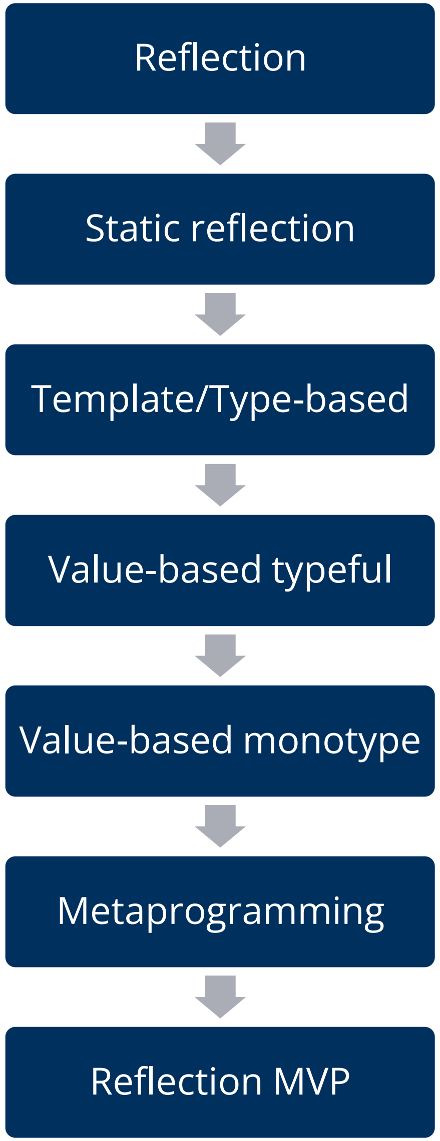
Vision paper: P2237, Metaprogramming, 2020

Syntax: P2320, The Syntax of Static Reflection, 2021

Comparison: P2560, Comparing value- and type-based reflection, 2022

C++26 MV: P2996, Reflection for C++26, 2023

C++26 feature freeze Q1 2025



Reflection in C++

Reflection = reflection proper (+ meta programming) + code generation, need all three

Reflection today: severely limited to some tools and hacks round __PRETTY_FUNCTION__

Code generation today: not possible, workarounds: macros, TMP, external generators

Meta programming: innovations around constexpr and template meta programming

Biggest impact IMO: value-based reflection will democratize metaprogramming

Type-based meta programming (TMP) will become obsolete

(Homogeneous) values + algorithms are much easier to approach than templates
— compare mp_list<Args...> with std::vector<std::meta::info>

This "side effect" of reflection may become its main feature!

People in the future will laugh at us for having done TMP!

Hopefully

Example: SoA offset computation

Offset of the I-th member of a struct

```
template <typename I>
constexpr auto roundUpToMultiple(I n, I mult) {
    return ((n + mult - 1) / mult) * mult;
}
```

```
template <typename Struct, int I>
consteval auto getMemberOffset() -> std::size_t {
  using namespace std::meta;
  return offset_of(nonstatic_data_members_of(^Struct)[I]);
}
```

```
}();
```

```
template <typename Struct, int I>
constexpr auto getMemberOffset() -> std::size_t {
    using Tuple = decltype(boost::pfr::structure_to_tuple(std::declval<Struct>()));
    return offsetOf<Tuple, I>;
}
```

Reflection will make metaprogramming ordinary programming

§ 2.3 List of Types to List of Sizes

Here, `sizes` will be a `std::array<std::size_t, 3>` initialized with `{sizeof(int), sizeof(float), sizeof(double)}`:

```
constexpr std::array types = {^int, ^float, ^double};
constexpr std::array sizes = []{
  std::array<std::size_t, types.size()> r;
  std::ranges::transform(types, r.begin(), std::meta::size_of);
  return r;
}();
```

Meta programming will by 10x simpler

Compare this to the following type-based approach, which produces the same array `sizes`:

```
template<class...> struct list {};

using types = list<int, float, double>;

constexpr auto sizes = []<template<class...> class L, class... T>(L<T...>) {
    return std::array<std::size_t, sizeof...(T)>{{ sizeof(T)... }};
}(types{});
```

From: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2996r0.html#list-of-types-to-list-of-sizes

Back to data layouts: we want familiar syntax

A code using SoA should look the same as with vector<T> or [md]span<T>

returns Event&

```
auto& event = view[i]; // with AoS layout

if(event.h1.isMuon)) continue;
event.h3.isMuon = false;
```

Struct of values (SoV)

```
struct H {
  double px;
  double py;
  double pz;
  double probK;
  double probPi;
  int isMuon;
};
struct Event {
  H h1;
  H h2;
  H h3;
};
```

Back to data layouts: we want familiar syntax

A code using SoA should look the same as with vector<T> or [md]span<T>

returns EventRef by value

```
auto&& event = view[i]; // with SoA layout

if(event.h1.isMuon)) continue;
event.h3.isMuon = false;

Event e1 = view[i]; // cannot convert :(
auto  e2 = view[i]; // still a ref :(


e1.h1.px = 32;
view[i] = e1; // no conversion
```

Struct of values (SoV)

```
struct H {
  double px;
  double py;
  double pz;
  double probK;
  double probPi;
  int isMuon;
};
struct Event {
  H h1;
  H h2;
  H h3;
};
```

Refl. →

Struct of references (SoR)

```
struct HRef {
  double& px;
  double& py;
  double& pz;
  double& probK;
  double& probPi;
  int& isMuon;
};
struct EventRef {
  HRef h1;
  HRef h2;
  HRef h3;
};
```

Back to data layouts: we want familiar syntax

A code using SoA should look the same as with vector<T> or [md]span<T>

```
auto&& event = view[i]; // with SoA layout

if(event.h1.isMuon)) continue;
event.h3.isMuon = false;

Event e1 = view[i]; // conversion op. :)
auto  e2 = view[i]; // still a ref :(

e1.h1.px = 32;
view[i] = e1; // converting assign. :)
```

Needs language feature, P0672

Struct of values (SoV)

```
struct H {
  double px;
  double py;
  double pz;
  double probK;
  double probPi;
  int isMuon;
};
struct Event {
  H h1;
  H h2;
  H h3;
};
```

Refl. →

Struct of references (SoR)

```
struct HRef {
  double& px;
  double& py;
  double& pz;
  double& probK;
  double& probPi;
  int& isMuon;
  HRef& operator=(const H&) {...}
  operator H() const {...}
};
struct EventRef {
  HRef h1;
  HRef h2;
  HRef h3;
  EventRef& op=(const Event&) {.}
  operator Event() const {...}
};
```

Back to data layouts: we want familiar syntax

A code using SoA should look the same as with vector<T> or [md]span<T>

A struct of references has drawbacks

Must compute all references upfront (eager)

Even if they will not be used

However: compilers are good at removing dead code

If you track accesses -> wrong results

Want fine-grained lazy access

Need proxy reference also for elements

Drawback: stores at least View* + array index

But again: optimizers are good these days

This is what LLAMA does for instrumentation, bitpacking, accessors

Struct of proxies (SoP)

```
struct HRef {
  ProxyRef<double> px;
  ProxyRef<double> py;
  ProxyRef<double> pz;
  ProxyRef<double> probK;
  ProxyRef<double> probPi;
  ProxyRef<int> isMuon;
  ...
};
struct EventRef {
  HRef h1;
  HRef h2;
  HRef h3;
  ...
};
```

Interaction between SoV and SoR/SoP

Conversions not enough, we have operators too!

Reflection: Copy impl. from SoV to SoR/SoP

This isn't a problem in ordinary AoS C++:

T& implicitly decays to T when necessary

T binds to a T&

Can call a member function on T or T&

SoV and SoR/SoP: explicit impl. required

But probably automatable -> reflection!

Alternative: language support to customize T&

See also: Andrei Alexandrescu, [Reflection in C++ - Past, Present, and Hopeful Future](), CppCon 2022

```
struct Vec3 {
  double x;
  double y;
  double z;
  Vec3& operator+=(const Vec3& other);
};
Vec3 operator+(Vec3 a, Vec3 b); // ok

struct Vec3Ref {
  double& x;
  double& y;
  double& z;
  Vec3Ref& operator=(const Vec3&) {...}
  operator Vec3() const {...}
  Vec3Ref& operator+=(const Vec3& other);
};

Vec3 v = points[i] + points[j]; // 2 conv
points[i] += points[j]; // 1 conv, Vec3Ref::+=
```

Maybe beyond P2996 and P1240, but in P2237

Refl.

Member functions on proxy references

In AoS world, we can call v.planar() on a Vec3& v

How to call planar() on a Vec3Ref?

Option #1: copy member functions as well (reflection)

Option #2: free function (big rewrite, works today)

Option #3: smart references? (language evolution)

Option #4: deducing this + delegation (combination)

If we cannot touch Vec3, only #1 works efficiently

Needs reflection!

```
struct Vec3 {
  double x;
  double y;
  double z;
  Vec2 planar() const { return {x, y}; }
};

Vec2 planar(auto&& v) { return {v.x, v.y};  } // #2

struct Vec3Ref {
  double& x;
  double& y;
  double& z;
  Vec3Ref& operator=(const Vec3&) {...}
  operator Vec3() const {...}
  Vec2 planar() { return {x, y}; } // #1
};
```

Member functions on proxy references

Smart references

Would require language evolution

P0416: Operator Dot

P0352: Smart References through Delegation

Both need to manifest a Vec3 to produce a Vec3&

Requires copying entire Vec3 data

Involving all side effects of copy construction

Again, optimizers may be smart …

```
struct Vec3 {
  double x;
  double y;
  double z;
  Vec2 planar() const { return {x, y}; }
};

struct Vec3Ref { // P0416
  ...
  Vec3& operator.() { ... } // Vec3& points to?
};
view[i].planar(); // calls:
view[i].operator.().planar();

struct Vec3Ref : using Vec3 { // P0352
  ...
  operator Vec3&() { ... } // Vec& points to?
};
view[i].planar(); // calls:
view[i].operator Vec3&().planar();
```

Member functions on proxy references
Deducing this and delegation

Deducing this (C++23) + P0352

Only loads actually needed data

Through the proxy reference

Maybe sometime in the future :)

Pattern: either share code between SoV and SoR/SoP via templates, or copy impl. with reflection

```
struct Vec3 {
  double x;
  double y;
  double z;
  Vec2 planar(this auto&& self) const { // C++23
    return {self.x, self.y};
  }
};
struct Vec3Ref : using Vec3 { // P0352
  double& x;
  double& y;
  double& z;
  ...
};
view[i].planar(); // calls:
Vec3::planar<Vec3Ref>(view[i]);

std_vector[i].planar(); // calls:
Vec3::planar<Vec3&>(view[i]);
```

Other languages

Reflection and meta programming are the foundations for SoA and other data layouts

Rust: proc macros

Token stream -> token stream

SoA crate: soa-derive

Rust will generate custom SoA vector CheeseVec and "generate the same functions that a Vec<Cheese> would have, and a few helper structs: CheeseSlice, CheeseSliceMut, CheeseRef and CheeseRefMut corresponding respectivly to &[Cheese], &mut [Cheese], &Cheese and &mut Cheese."

Very similar to what we have seen before!

Btw: worth reading the source code!

Zig: MultiArrayList (SoA) in standard library

ISPC: native language support for SoA

Jai: had native language support for SoA

Later replaced by meta programming

```
#[derive(StructOfArray)]
pub struct Cheese {
    pub smell: f64,
    pub color: (f64, f64, f64),
    pub with_mushrooms: bool,
    pub name: String,
}
// generates:
pub struct CheeseVec {
    pub smell: Vec<f64>,
    pub color: Vec<(f64, f64, f64)>,
    pub with_mushrooms: Vec<bool>,
    pub name: Vec<String>,
}
```

Conclusion & outlook

Reflection gets us 80% to where we want and should be our first aim

Reflection MVP is a solid foundation and covers a lot

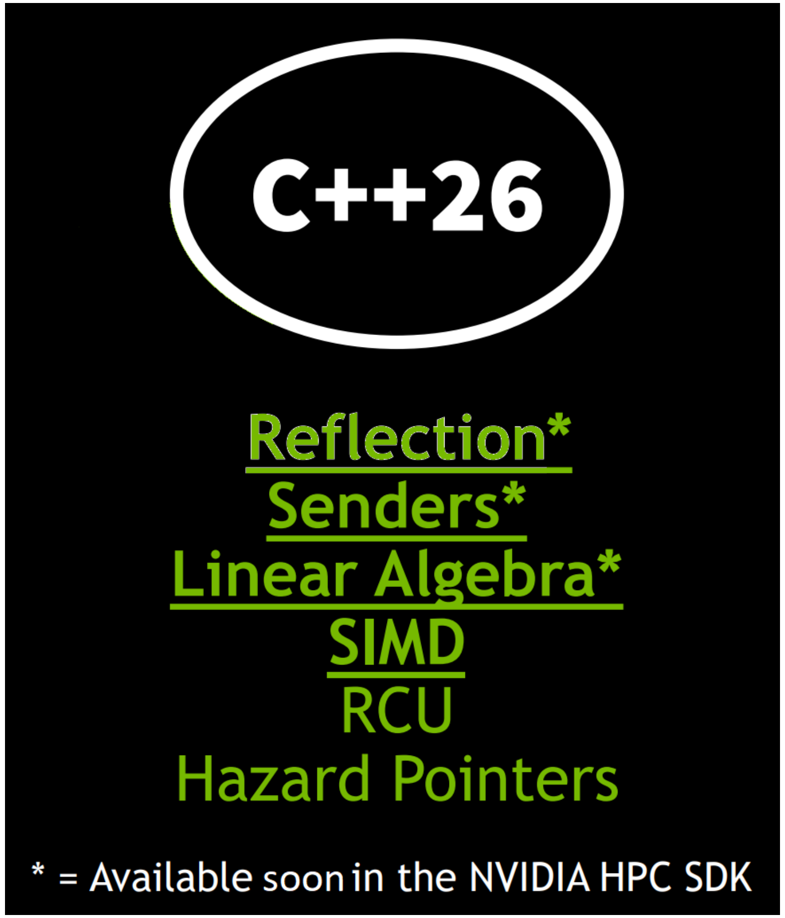
-> should land in C++26

Need impls. to evaluate, **including** for accelerators
— **CUDA**, HIP, SYCL

Then: more powerful code injection, like in P2237
— E.g. to extend existing class bodies

Later: better meta programming to glue reflection + injection

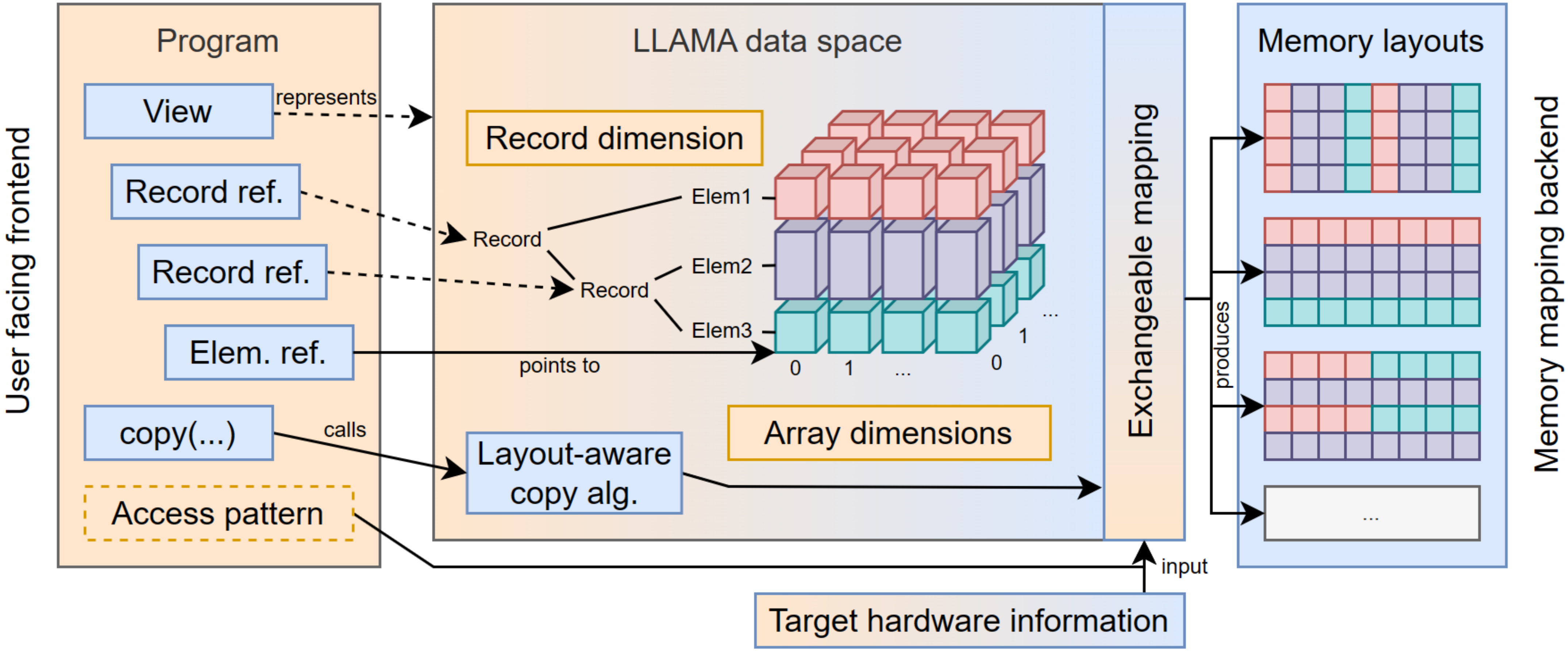
Eventually: std::soa_vector<T>, std::soa_span<T>



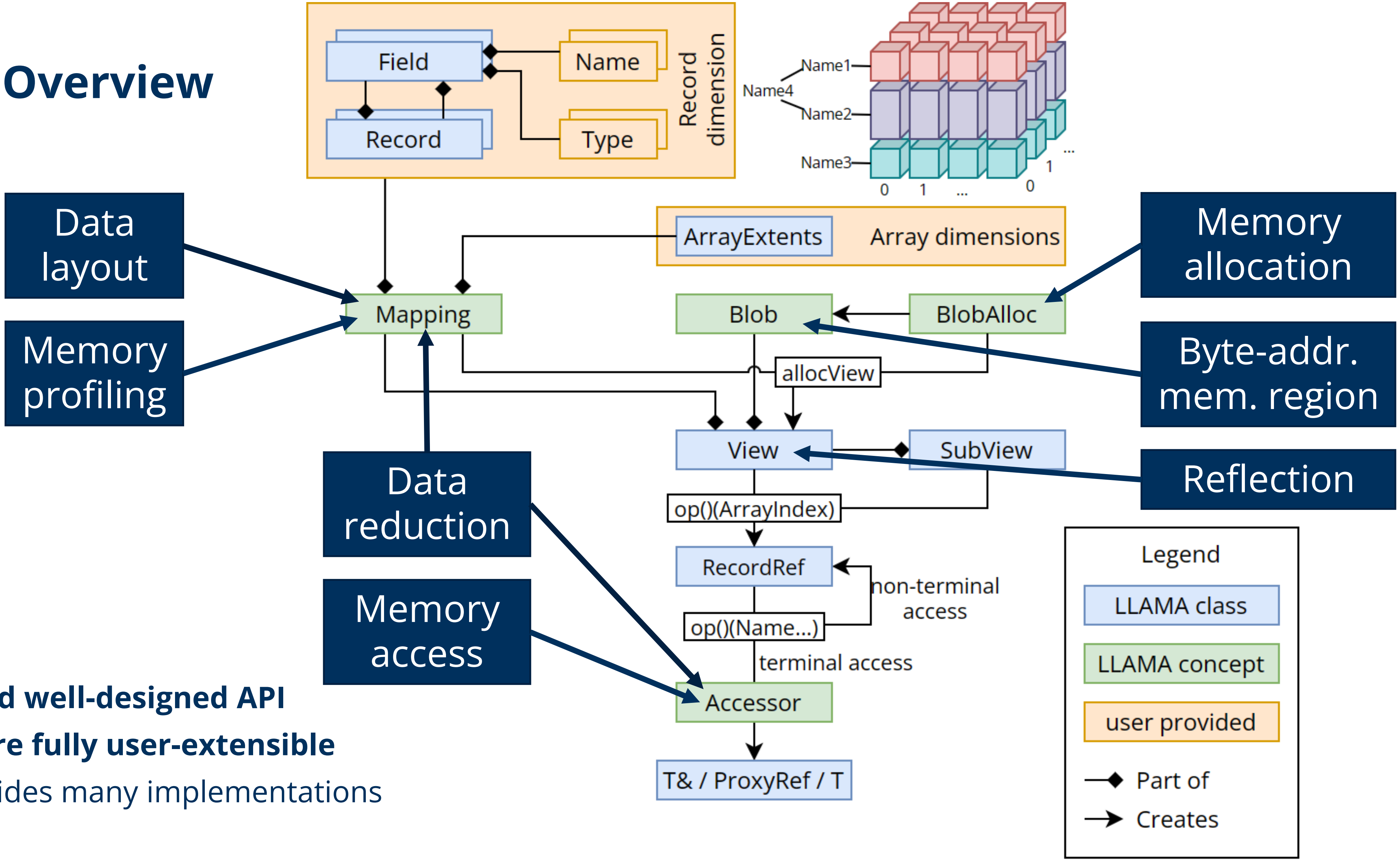
Maybe in NVIDIA's next keynote. Adapted from here

Backup slides

LLAMA concept



LLAMA
Library Overview



Minimal and well-designed API

Concepts are fully user-extensible

LLAMA provides many implementations

LHCB B2HHH analysis layout 4 definition

```
using Mapping = llama::mapping::Split<
    llama::ArrayExtentsDynamic<RE::NTupleSize_t, 1>,
    RecordDim,
    mp_list<mp_list<H1isMuon>, mp_list<H2isMuon>, mp_list<H3isMuon>>,
    llama::mapping::AlignedAoS,
    llama::mapping::BindSplit<
        mp_list<mp_list<H1ProbK>, mp_list<H2ProbK>>,
        llama::mapping::AlignedAoS,
        llama::mapping::AlignedAoS,
        true>::fn,
    true>;
```

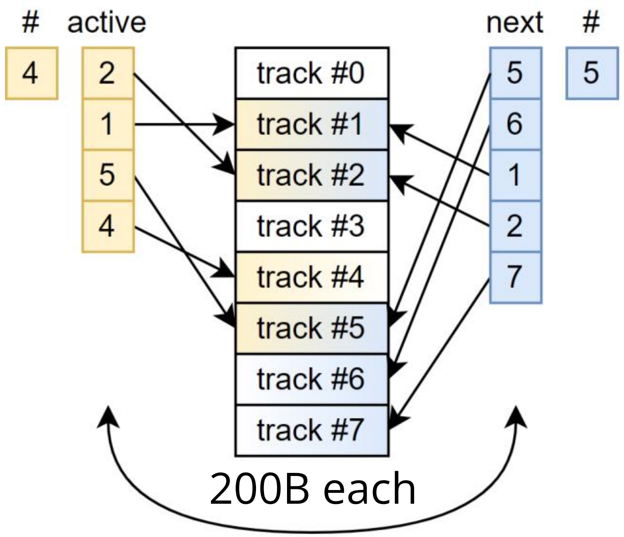
LHCB B2HHH analysis layout 8 definition

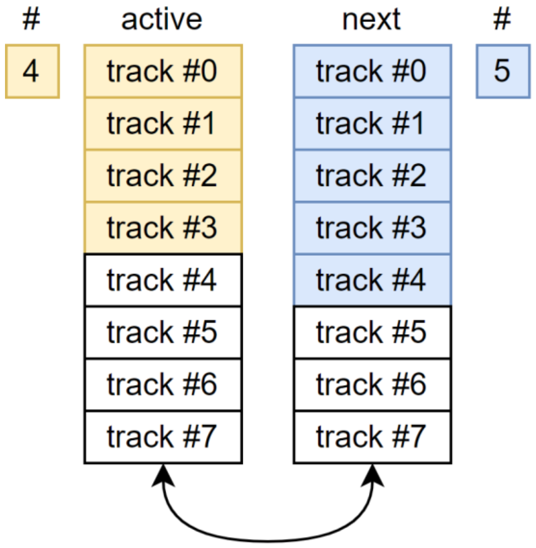
```
using Mapping = llama::mapping::Split<
    llama::ArrayExtentsDynamic<RE::NTupleSize_t, 1>,
    RecordDim,
    mp_list<mp_list<H1isMuon>, mp_list<H2isMuon>, mp_list<H3isMuon>>,
    llama::mapping::BindBitPackedIntAoS<llama::Constant<1>, llama::mapping::SignBit::Discard>::fn,
    llama::mapping::BindSplit<
        mp_list<mp_list<H1ProbK>, mp_list<H2ProbK>>,
        llama::mapping::BindChangeType<llama::mapping::BindAoS<>::fn, mp_list<mp_list<double, float>>>::fn,
        llama::mapping::BindBitPackedFloatAoS<llama::Constant<6>, llama::Constant<16>>::template fn,
        true>::fn,
    true>;
```

AdePT track data structure

Single sparse buffer

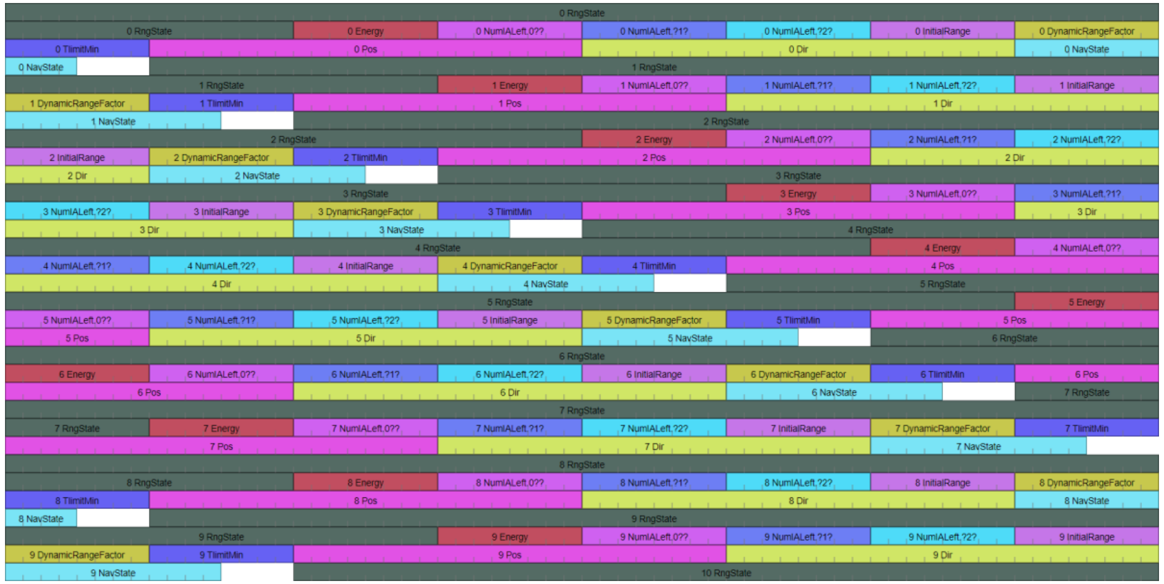
Double dense buffers





Drawback: copies more memory

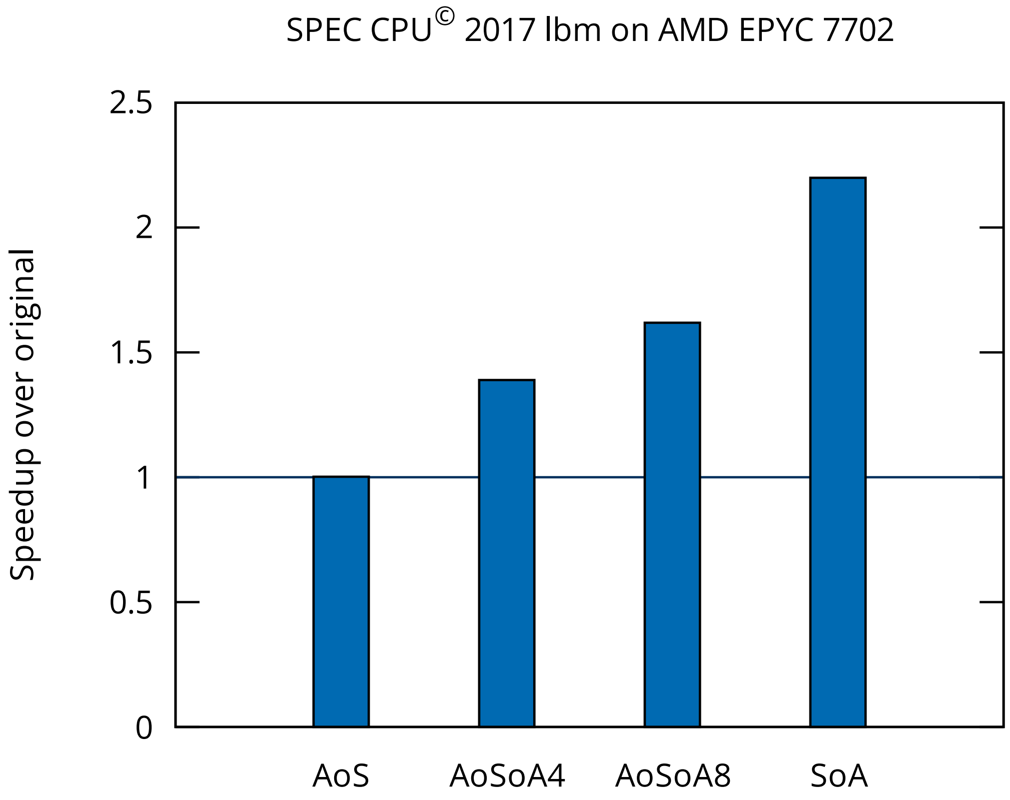
wrapped after 64B



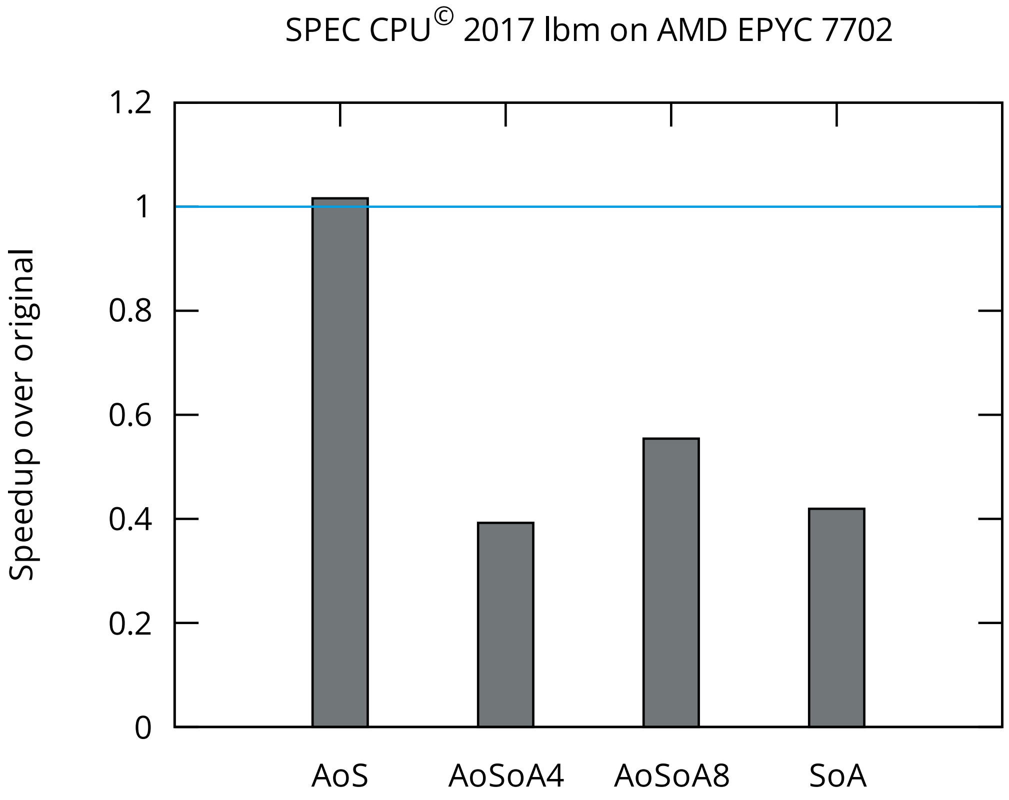
Layout visualization: 1 LOC with LLAMA

SPEC CPU® 2017 lbm

Multi threaded



Single threaded



Example – N-body simulation 1/3

```
using FP = float;
constexpr FP timestep = 0.0001, eps2 = 0.01;
constexpr int steps = 5, problemSize = 64 * 1024;

namespace tag {
  struct Pos{}; struct Vel{}; struct X{}; struct Y{};
  struct Z{}; struct Mass{};
}
using V3 = llama::Record<
  llama::Field<tag::X, FP>,
  llama::Field<tag::Y, FP>,
  llama::Field<tag::Z, FP>>;
using Particle = llama::Record<
  llama::Field<tag::Pos, V3>,
  llama::Field<tag::Vel, V3>,
  llama::Field<tag::Mass, FP>>;
```

Example – N-body simulation 2/3

```
void pPInteraction(auto&& pi, auto&& pj) {
  auto dist = pi(tag::Pos{}) - pj(tag::Pos{});
  dist *= dist;
  const auto distSqr = eps2 +
                       dist(tag::X{}) + dist(tag::Y{}) + dist(tag::Z{});
  const auto distSixth = distSqr * distSqr * distSqr;
  const auto invDistCube = FP{1} / sqrt(distSixth);
  const auto sts = (pj(tag::Mass{}) * timestep) * invDistCube;
  pi(tag::Vel{}) += dist * sts;
}

void update(auto& particles) {
  LLAMA_INDEPENDENT_DATA
  for(std::size_t i = 0; i < problemSize; i++) {
    llama::One<Particle> pi = particles(i);
    for(std::size_t j = 0; j < problemSize; ++j)
      pPInteraction(pi, particles(j));
    particles(i)(tag::Vel{}) = pi(tag::Vel{});
  }
}
```

Example – N-body simulation 3/3

```
void move(auto& particles) {
  LLAMA_INDEPENDENT_DATA
  for(std::size_t i = 0; i < problemSize; i++)
    particles(i)(tag::Pos{}) += particles(i)(tag::Vel{}) * timestep;
}
int main() {
  using ArrayExtents = llama::ArrayExtentsDynamic<std::size_t, 1>;
  using Mapping = llama::mapping::AoS<ArrayExtents, Particle>; // !!!
  auto mapping = Mapping{ArrayExtents{problemSize}};
  auto view = llama::allocViewUninitialized(mapping); // !!!
  for(auto&& p : view) {
    p(tag::Pos{}, tag::X{}) = random();
    // ...
    p(tag::Mass{}) = random();
  }
  for(std::size_t s = 0; s < steps; ++s) {
    update(view);
    move(view);
  }
}
```

Change mapping with this line

Set custom blob alloc. or accessor on this line

Mapping example: AoS implementation

```
template<typename TArrayExtents, typename TRecordDim, bool AlignAndPad = true,
  typename TLinearizeArrayDimsFunctor = LinearizeArrayDimsCpp,
  template<typename> typename FlattenRecordDim = FlattenRecordDimInOrder>
struct AoS : MappingBase<TArrayExtents, TRecordDim> {
  private:
    using Base = MappingBase<TArrayExtents, TRecordDim>;
    using size_type = typename Base::size_type;
  public:
    inline static constexpr bool alignAndPad = AlignAndPad;
    using LinearizeArrayDimsFunctor = TLinearizeArrayDimsFunctor;
    using Flattener = FlattenRecordDim<TRecordDim>;
    inline static constexpr std::size_t blobCount = 1;
    using Base::Base;
    LLAMA_FN_HOST_ACC_INLINE constexpr auto blobSize(size_type) const -> size_type {
      return LinearizeArrayDimsFunctor{}.size(Base::extents())
        * flatSizeOf<typename Flattener::FlatRecordDim, AlignAndPad>;
    }
    template<std::size_t... RecordCoords>
    LLAMA_FN_HOST_ACC_INLINE constexpr auto blobNrAndOffset(
      typename Base::ArrayIndex ai,
      RecordCoord<RecordCoords...> = {}) const -> NrAndOffset<size_type> {
      constexpr std::size_t flatFieldIndex = Flattener::template flatIndex<RecordCoords...>;
      const auto offset = LinearizeArrayDimsFunctor{}(ai, Base::extents())
        * static_cast<size_type>(flatSizeOf<typename Flattener::FlatRecordDim, AlignAndPad>)
        + static_cast<size_type>(flatOffsetOf<typename Flattener::FlatRecordDim, flatFieldIndex,
                                                                                  AlignAndPad>);
      return {size_type{0}, offset};
    }
};
```

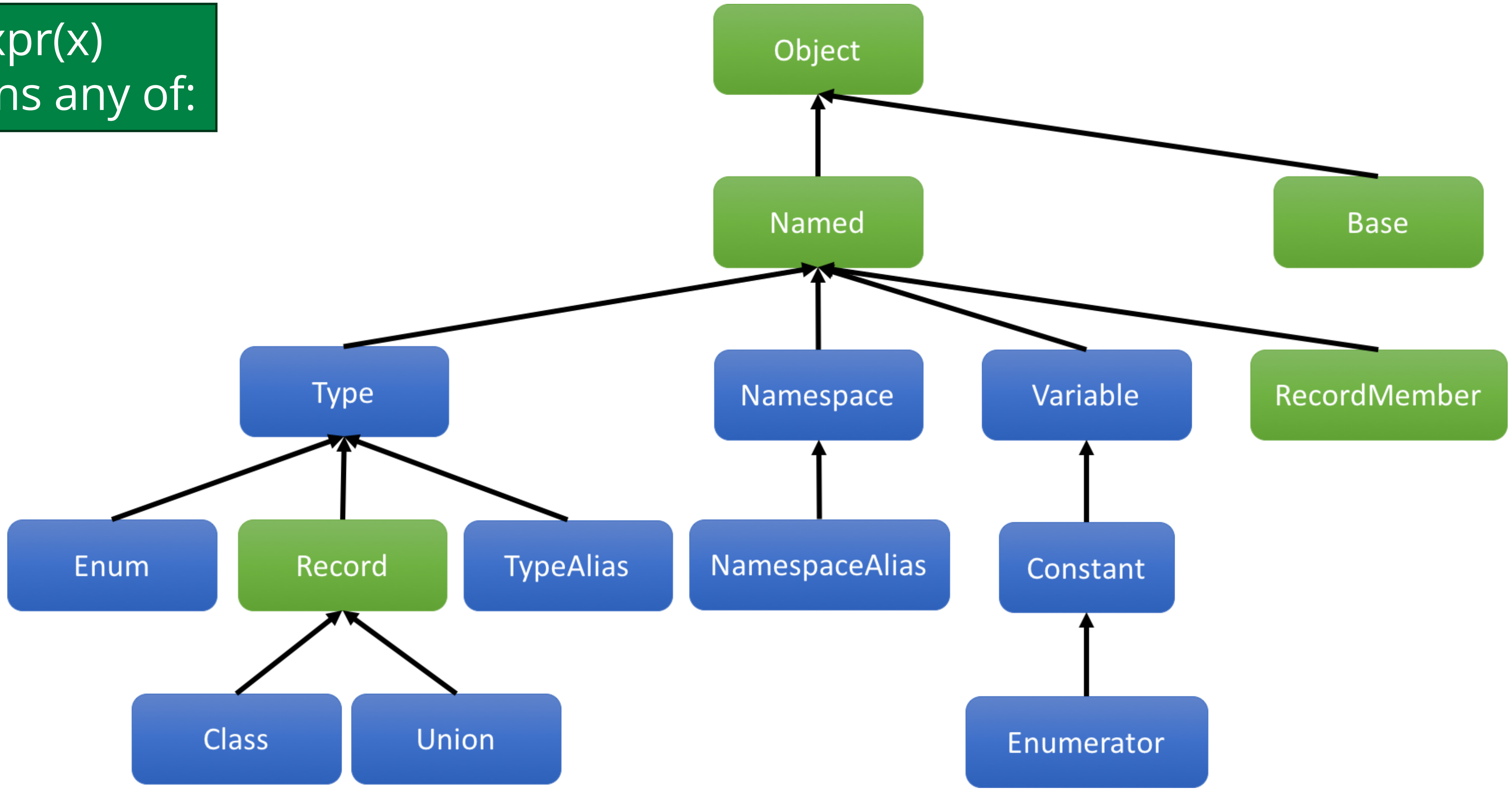
P0367: std::accessor – access types

non-unified memory
read/write qualifiers
non-temporal access
aliasing
sequential access
prefetching
burst mode
pipelined access
DMA

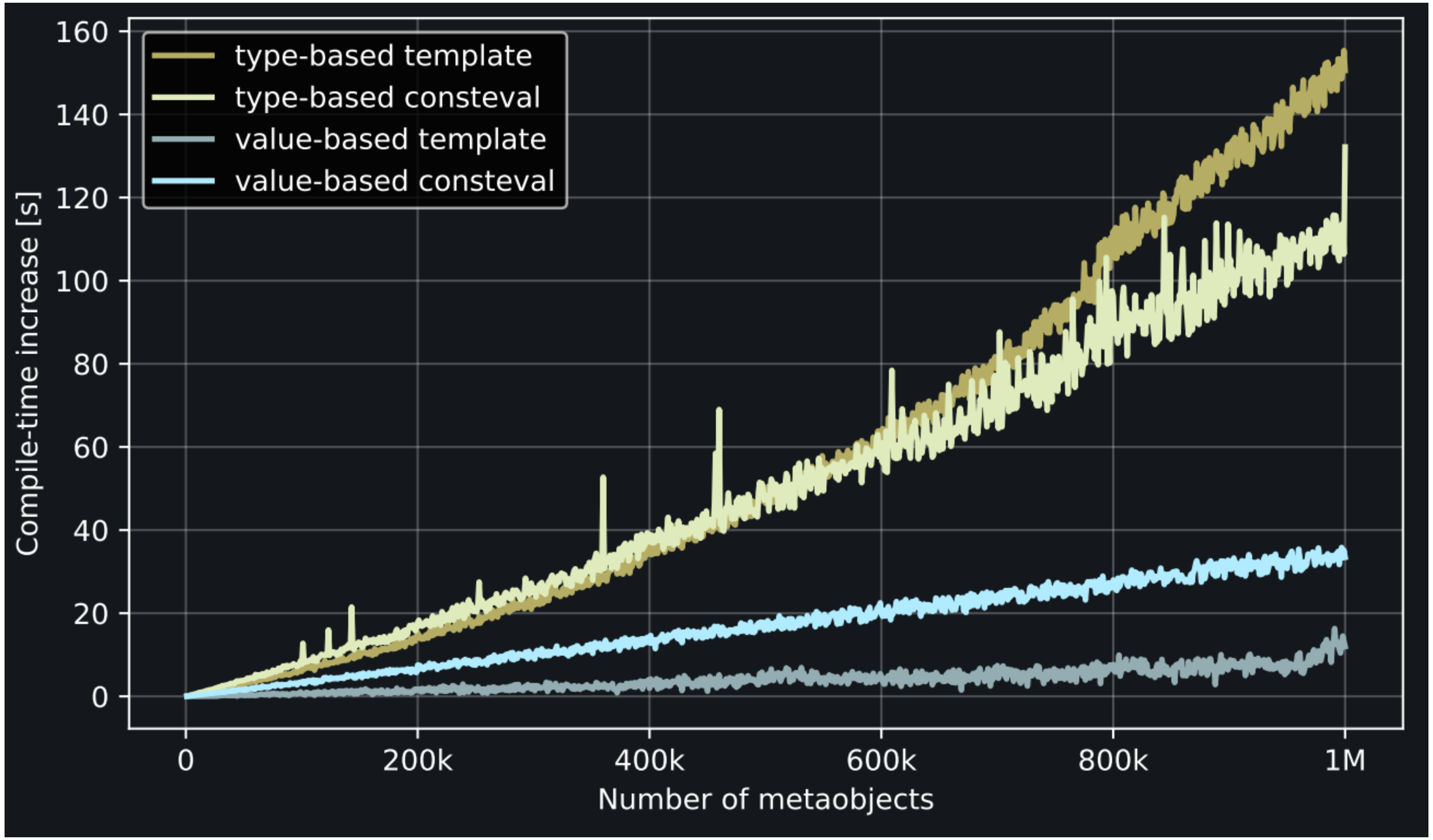
bus type
access width
address mode
translation
modulo addressing
address bit setting
transactional memory
prediction
generic proxy

P0953r2 type hierarchy

reflexpr(x)
returns any of:



P2560: compile time increase per N metaobjects




```
auto t = ^x;

// template <int id>
// void foo(std::meta::TYPE<id>)
foo(t);

// consteval void foo(int id);
// t converts to id
foo(t);

std::meta::info i = ^x;

// template <std::meta::info id>
// void foo();
foo<x>();

// consteval void foo(
//   std::meta::info id);
foo(x);
```