Bernhard Manfred Gruber

Object layout in C++

```
struct Position {
   float x;
   float y;
};
Position pos[8];
```

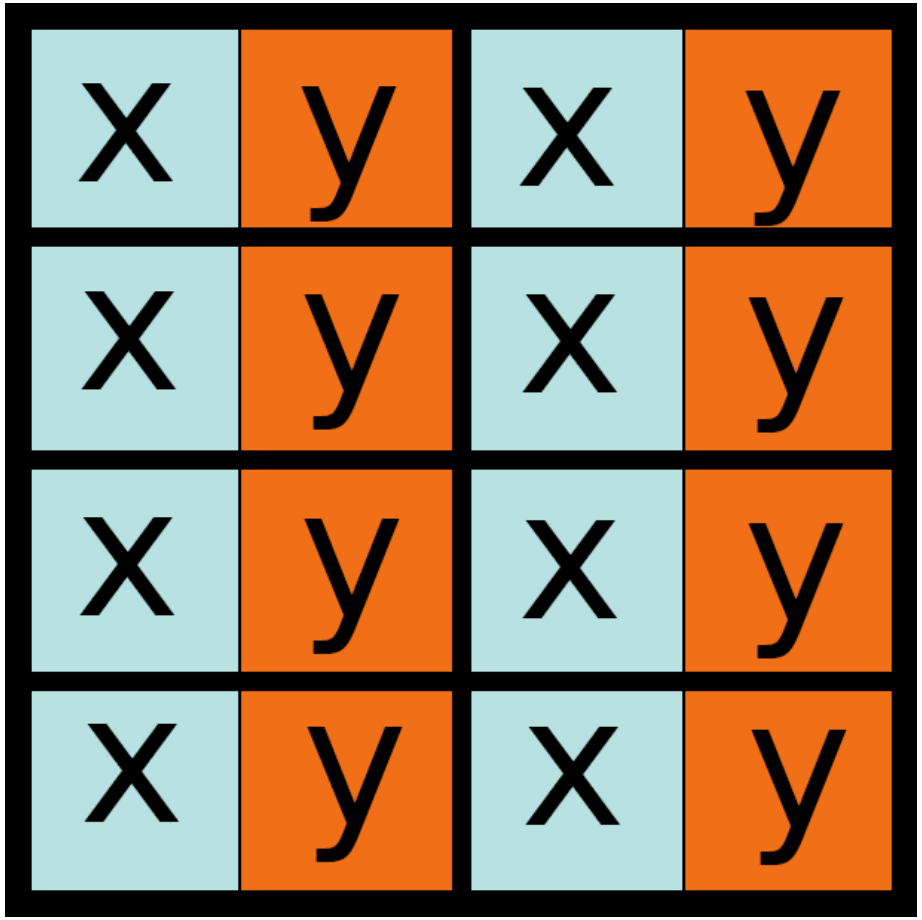
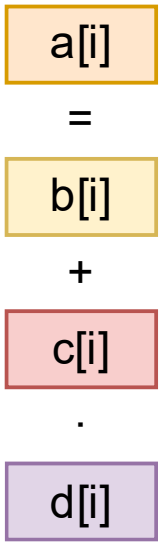


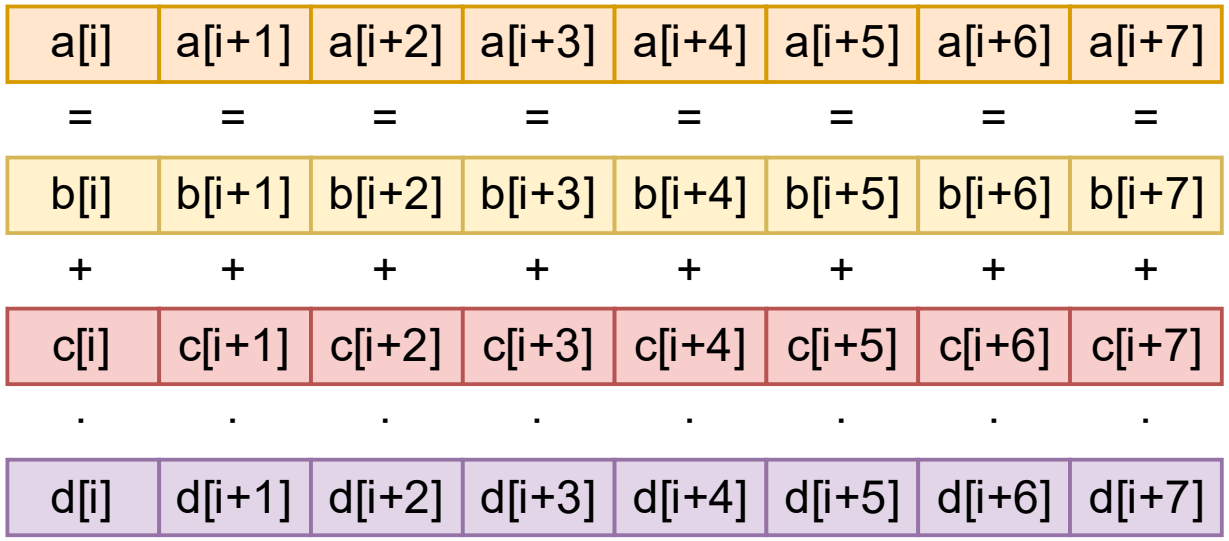
Image from LLAMA presentation of A. Matthes

Vectorization

a[i] = b[i] + c[i] · d[i];



a[i:8] = b[i:8] + c[i:8] · d[i:8];



Vectorization

- SIMD (Single Instruction Multiple Data)
  - Use data parallelism
- Improves CPU single core performance
- Manual vectorization using assembler or intrinsics
- Most modern compilers can auto vectorize if
  - the memory access pattern is favorable
  - … (a LOT more conditions)
- On GPUs, threads are grouped (32/64 threads)
  - SIMT (Single Instruction Multiple Threads)
  - If vectors of threads access vectors of data => coalescing => fast

Memory layout: AoS (Array of Structs)

```
struct Position {
   float x;
   float y;
};
Position pos[8];
```

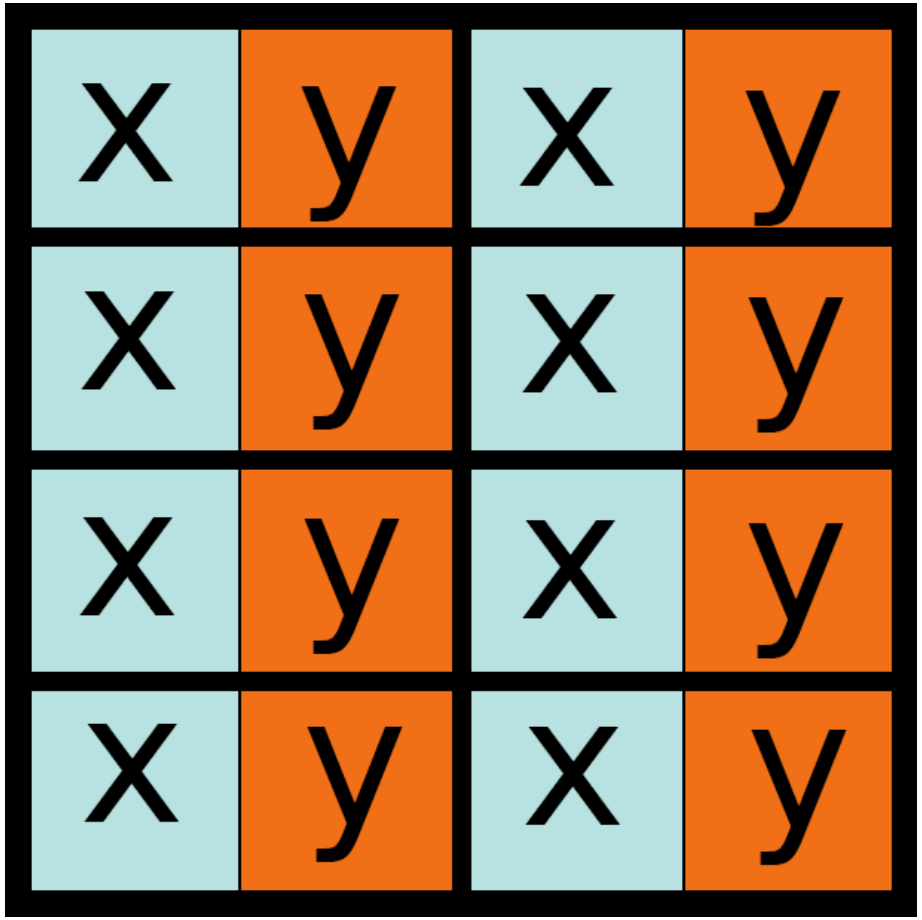


Image from LLAMA presentation of A. Matthes

Memory layout: SoA (Struct of Arrays)

```
struct Position {
   float x[8];
   float y[8];
};
Position pos;
```

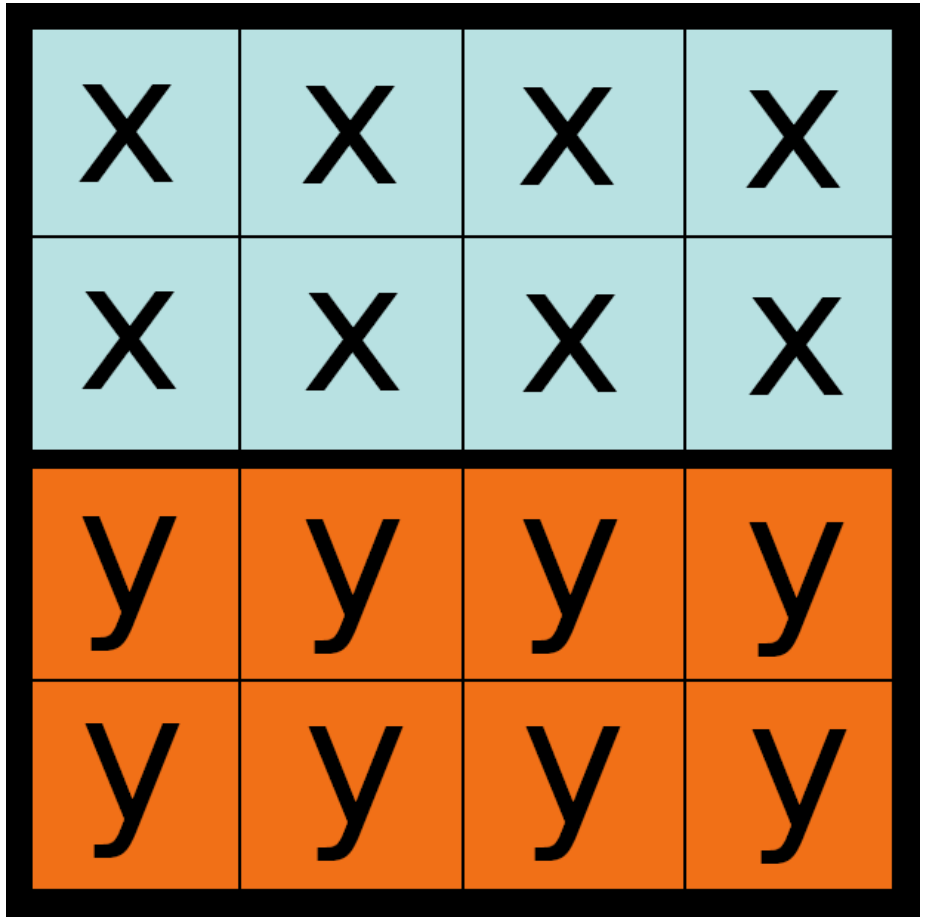


Image from LLAMA presentation of A. Matthes

Memory layout: Blocking

```
struct Position {
   float x[4];
   float y[4];
};
Position pos[2];
```

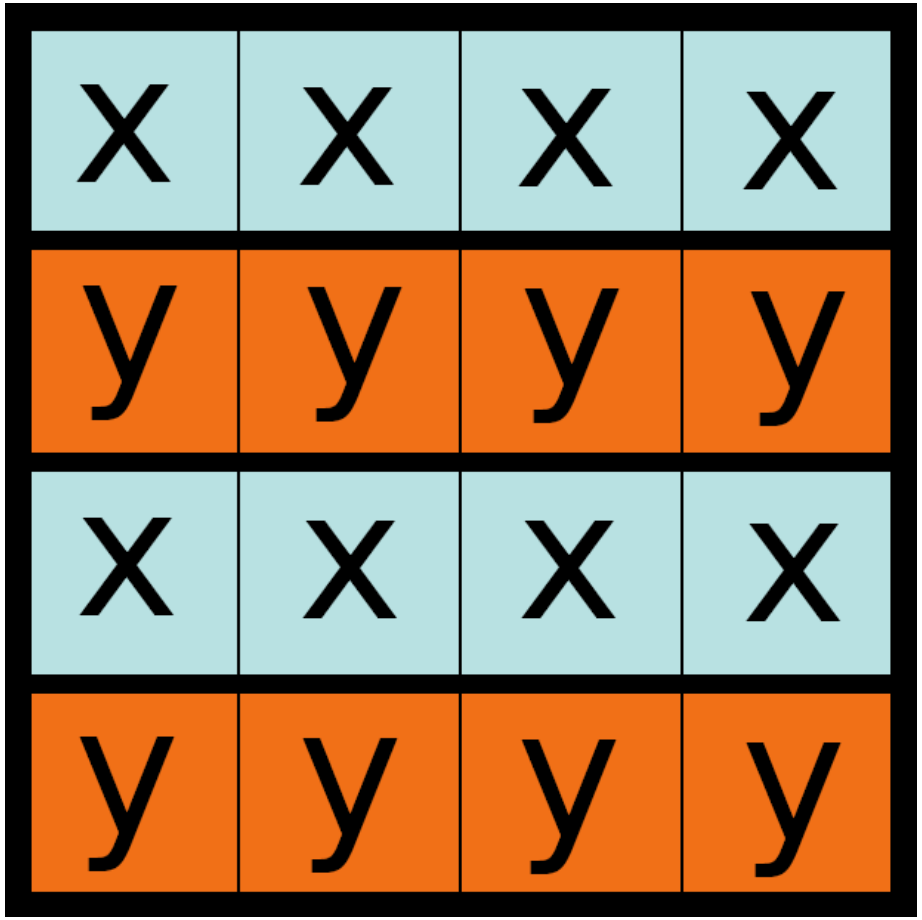


Image from LLAMA presentation of A. Matthes

Padding

```
struct Position {
   float x[2];
   float dummy1;
   float y[2];
   float dummy2;
};
Position pos[4];
```

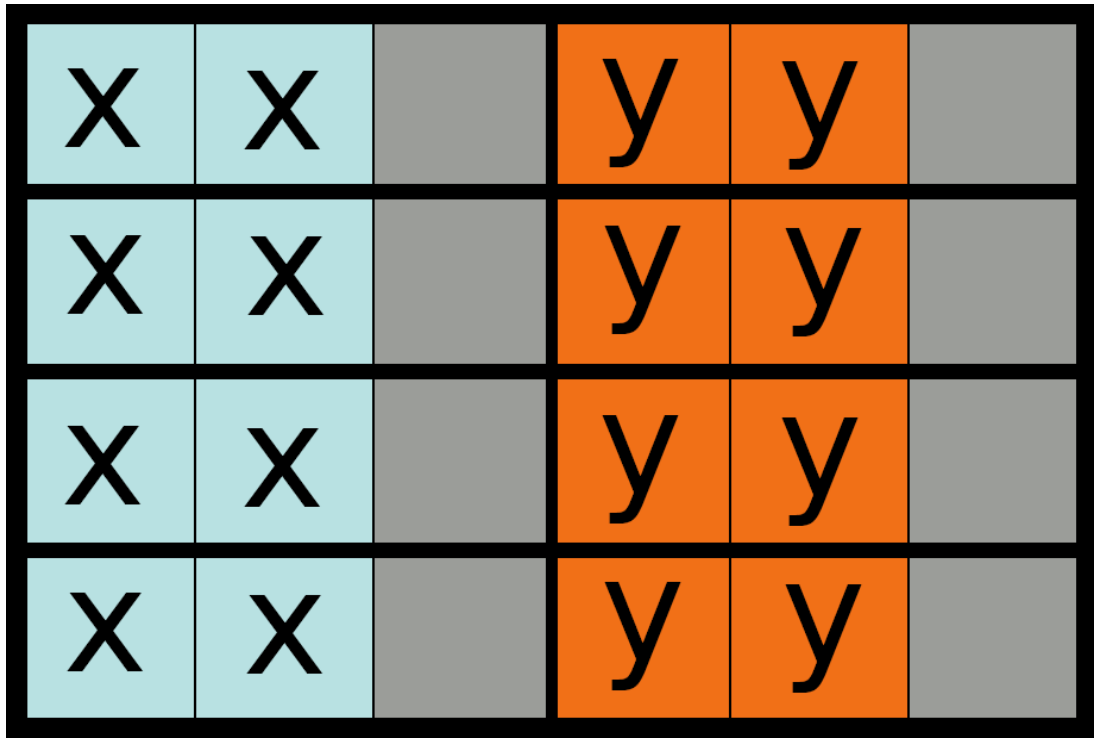


Image from LLAMA presentation of A. Matthes

Problem

- Program representation of data is tied to physical memory layout
  - When we define data
    - `struct P { float x;     float y;     } p[64];`
    - `struct P { float x[64]; float y[64]; } p;`
  - When we access data
    - `p[i].x = 3.14f; func(p[i]);`
    - `p.x[i] = 3.14f; func(p.x[i], p.y[i]);`
  - Same algorithm needs to be written multiple times for different data layouts
- Different hardware needs different data layouts to perform well

- What if memory layout could be separated from access in program?
  - And not just AoS and SoA, any mapping!
  - Data layout is a bijection between program and physical memory representation

LLAMA

- Low Level Abstraction of Memory Access
- “Splitting of algorithmic view of data and the actual mapping in the background so that different layouts may be chosen without touching the algorithm at all”
- Header only, portable, C++11/17 library
- Works with CUDA >= 9/11
- Designed to integrate with alpaka, but orthogonal
- Eventually relies on compiler’s auto-vectorizer

Example – N-body simulation

- Timestep simulation of particles with mass moving in space and attracting each other

For n timesteps:

    For each particle p1:

        For each particle p2:

            Update velocity of p2 based on the influence of p1, given a timestep t

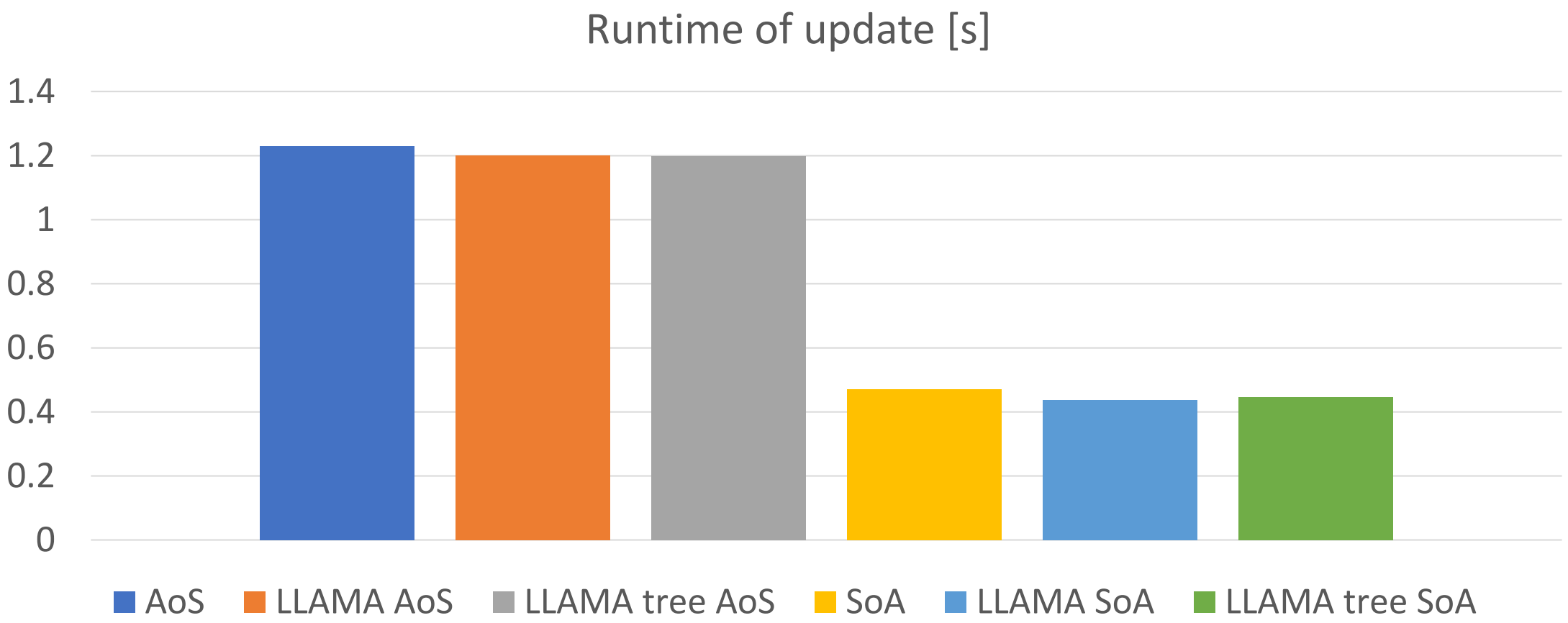
    For each particle p

        Updated position of p depending on its velocity

Example – N-body simulation

- SoA: https://godbolt.org/z/LnynGc
- AoS : https://godbolt.org/z/M53MZK
- LLAMA : https://godbolt.org/z/5k-ZaN

Example – N-body simulation



Datum Domain

```
struct Particle {
    struct Pos {
        float x;
        float y;
        float z;
    } pos;
    struct Vel {
        float x;
        float y;
        float z;
    } vel;
    float mass;
};
```

```
namespace tag {
  struct Pos {}; struct Vel {};
  struct X {}; struct Y {}; struct Z {};
  struct Mass {};
}
using Particle = llama::DS<
  llama::DE<tag::Pos, llama::DS<
    llama::DE<tag::X, float>,
    llama::DE<tag::Y, float>,
    llama::DE<tag::Z, float>
  >>,
  llama::DE<tag::Vel, llama::DS<
    llama::DE<tag::X, float>,
    llama::DE<tag::Y, float>,
    llama::DE<tag::Z, float>
  >>,
  llama::DE<tag::Mass, float>
>;
```

Datum Domain

- Tags used as names for elements
- A tree of
  - llama::DS = DatumStruct
  - llama::DE = DatumElement
  - llama::DA = DatumArray
- DatumElements
  - Use tag types as names
  - Leave elements carry final type used for computation

```
namespace tag {
  struct Pos {};
  struct X {}; struct Y {}; struct Z {};
  struct Momentum {}; struct Weight {};
}
using Particle = llama::DS<
    llama::DE<tag::Pos, llama::DS<
        llama::DE<tag::X, float>,
        llama::DE<tag::Y, float>,
        llama::DE<tag::Z, float>>>,
    llama::DE<tag::Momentum, llama::DS<
        llama::DE<tag::Z, double>,
        llama::DE<tag::X, double>>>,
    llama::DE<tag::Weight, int>,
    llama::DE<tag::Options,
              llama::DA<bool, 4>>>;
```

User Domain

- Same principle as alpaka grid workdiv, CUDA grid, OpenCL NDRange
- Compile-time dimensionality of the problem
- Runtime size of the problem

```
using UserDomain = llama::UserDomain<1>;
const UserDomain userDomain{PROBLEM_SIZE};
```

Mapping

- Mapping is a compile time bijection that maps the datum domain to a linear byte array (and back)
- Depends on datum and user domain (compile + runtime info)
- LLAMA provides AoS, SoA and tree mappings

```
using Mapping = llama::mapping::AoS<UserDomain, Particle>;
Mapping mapping(userDomain);
```

Factory

- Creates view using mapping and an allocator
- LLAMA provides std::vector, std::shared_ptr and stack-based allocator

```
using Factory = llama::Factory<Mapping,
  llama::allocator::Vector<>>;
auto view = Factory::allocView(mapping);
```

View

- Holds storage of data
- Provides a rich interface for accessing datum elements

```
const UserDomain pos{0};
float& x = view(pos).access<0, 0>();
float& x = view(pos).access<0>().access<0>();
float& x = view(pos).access<0>().access<tag::X>();
float& x = view(pos).access<tag::Pos>().access<0>();
float& x = view(pos).access<tag::Pos, tag::X>();
float& x = view(pos).access<tag::Pos>().access<tag::X>();
float& x = view[pos](tag::Pos{}, tag::X{});
float& x = view[pos](tag::Pos{})(tag::X{});
float& x = view[pos].access<tag::Pos>()(tag::X{});
float& x = view[pos](tag::Pos{}).access<tag::X>();
float& x = view[pos](llama::DatumCoord<0, 0>{});
float& x = view[pos](llama::DatumCoord<0>{})(llama::DatumCoord<0>{});
```

VirtualDatum

- Every access on a view that does not resolve to a leave in the datum domain tree returns a VirtualDatum
- Supports further access
- Supports many operators (fun!)

```
llama::VirtualDatum<decltype(view)> datum = view(pos);
llama::VirtualDatum<decltype(view), llama::DatumCoord<0>> pos
  = datum.access<st::Pos>();
float& x = pos.access<st::X>();
```

Single element view on stack

- Sometimes we just want a single datum instance
  - On the stack, like a local variable

```
auto datum = llama::stackVirtualDatumAlloc<Particle>();
datum(tag::Pos{}, tag::X{}) = 14.0f;
datum(tag::Pos{}, tag::Y{}) = 15.0f;
datum(tag::Pos{}, tag::Z{}) = 16.0f;
datum(tag::Momentum{}) = 0;
datum(tag::Weight{}) = 500.0f;
datum(tag::Options{}).access<0>() = true;
datum(tag::Options{}).access<1>() = false;
datum(tag::Options{}).access<2>() = true;
datum(tag::Options{}).access<3>() = false;

view({3}) = datum;
```

VirtualDatum operators

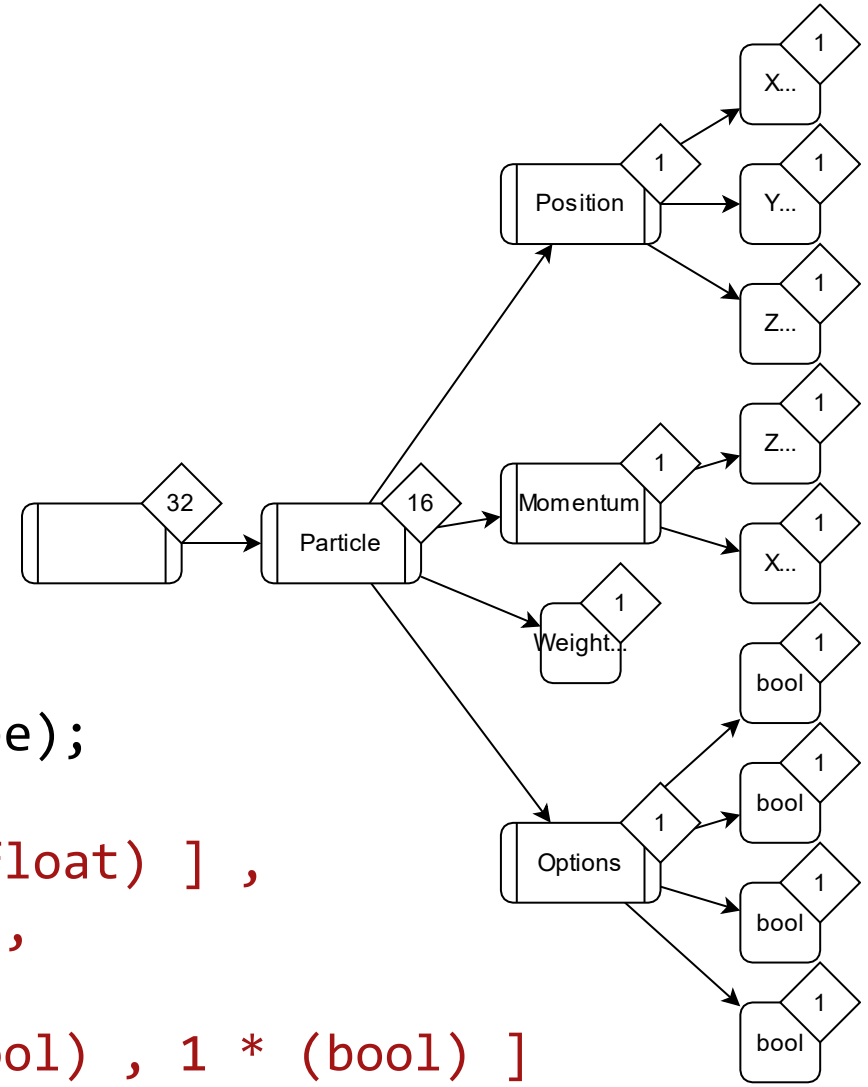
- Most arithmetic and logical operators are supported
- Operations are applied on the subtree a VirtualDatum represents

```
auto datum = llama::stackVirtualDatumAlloc<Particle>();
datum = 2; // broadcast to all elements
datum(tag::Pos{}) = 1; // broadcast to Pos/[X|Y|Z]
datum(tag::Pos{}) > 0; // conjunction of > applied on all elements
datum *= 2; // doubles every element
datum(tag::Pos{}) += datum(tag::Momentum{});
  // applies Pos/[X|Z] += Momentum/[X|Z], Pos/Y is unmodified
```

Tree mapping

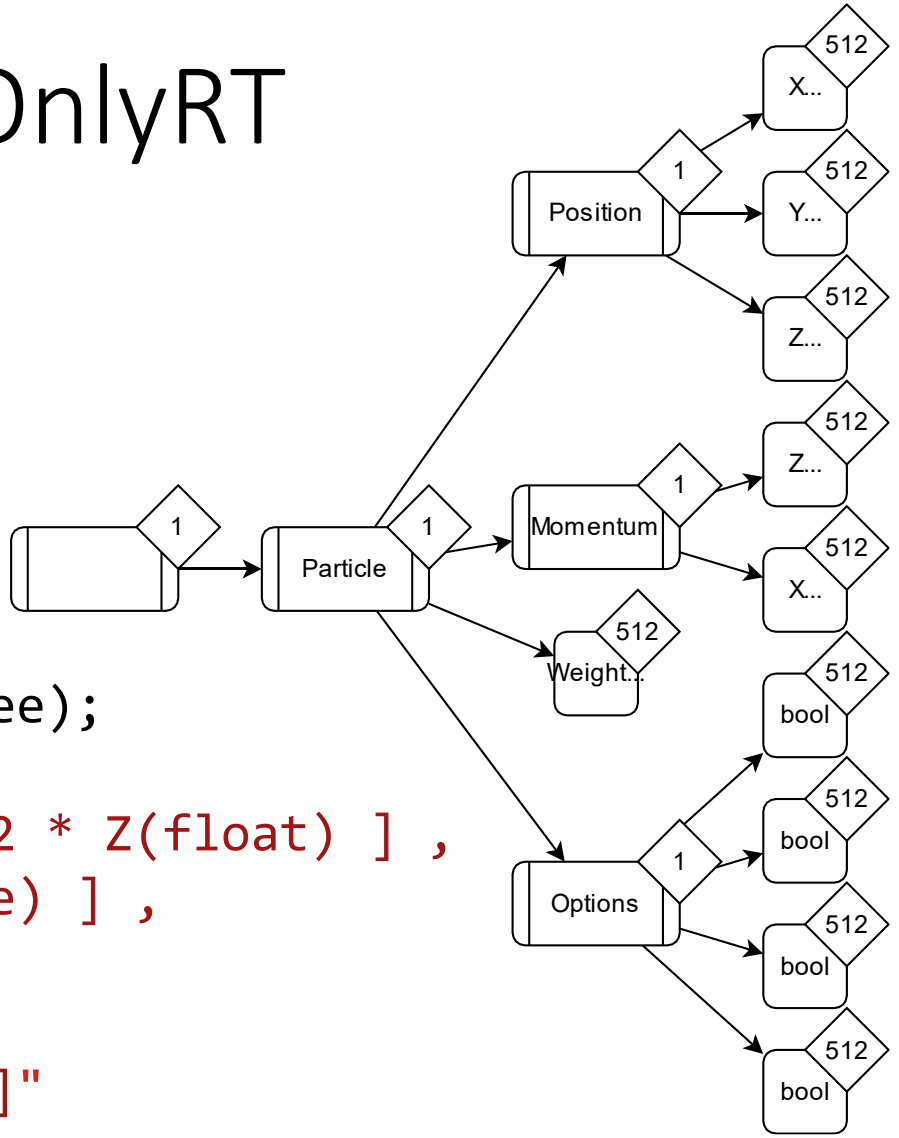
```
using UserDomain = llama::UserDomain<2>;
const UserDomain userDomain{32, 16};
auto operations = llama::Tuple{};
using Mapping = llama::mapping::tree::Mapping<
  UserDomain, Particle, decltype(operations)>;
const Mapping mapping(userDomain, operations);

llama::mapping::tree::toString(mapping.resultTree);
"32 * [ 16 * [
  1 * Pos[ 1 * X(float) , 1 * Y(float) , 1 * Z(float) ] ,
  1 * Momentum[ 1 * Z(double) , 1 * X(double) ] ,
  1 * Weight(int) ,
  1 * Options[ 1 * (bool) , 1 * (bool) , 1 * (bool) , 1 * (bool) ]
] ]"
```



Tree mapping functors - LeafOnlyRT

```
auto operations = llama::Tuple{
  llama::mapping::tree::functor::LeafOnlyRT()};
```



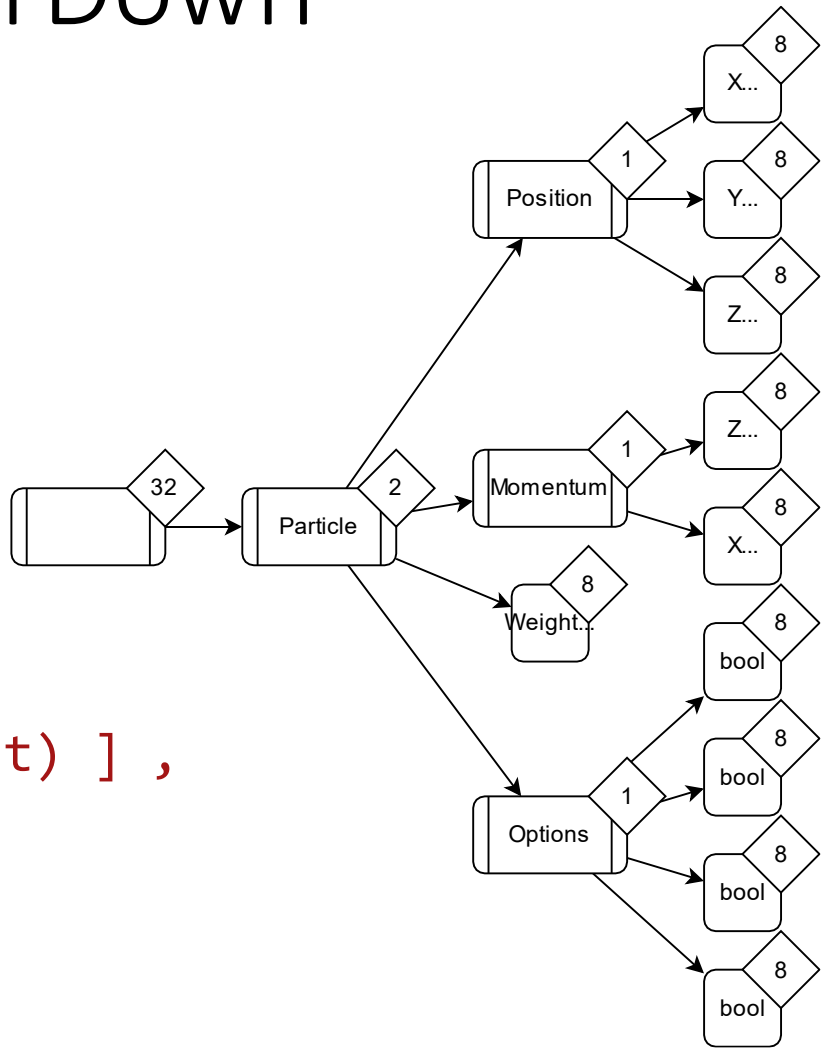

```
llama::mapping::tree::toString(mapping.resultTree);
"1 * [ 1 * [
  1 * Pos[ 512 * X(float) , 512 * Y(float) , 512 * Z(float) ] ,
  1 * Momentum[ 512 * Z(double) , 512 * X(double) ] ,
  512 * Weight(int) ,
  1 * Options[ 512 * (bool) , 512 * (bool) ,
               512 * (bool) , 512 * (bool) ] ] ]"
```

https://godbolt.org/z/4-k3y2

Tree mapping functors - MoveRTDown

```
auto vectorWidth = 8;
using llama::mapping::tree::functor::MoveRTDown;
using llama::mapping::tree::TreeCoord;
auto operations = llama::Tuple{
  MoveRTDown<TreeCoord<0>>{vectorWidth},
  MoveRTDown<TreeCoord<0, 0>>{vectorWidth},
  MoveRTDown<TreeCoord<0, 1>>{vectorWidth},
  MoveRTDown<TreeCoord<0, 3>>{vectorWidth}};

llama::mapping::tree::toString(mapping.resultTree);
"32 * [ 2 * [
  1 * Pos[ 8 * X(float) , 8 * Y(float) , 8 * Z(float) ] ,
  1 * Momentum[ 8 * Z(double) , 8 * X(double) ] ,
  8 * Weight(int) ,
  1 * Options[ 8 * (bool) , 8 * (bool) ,
               8 * (bool) , 8 * (bool) ] ] ]"
```

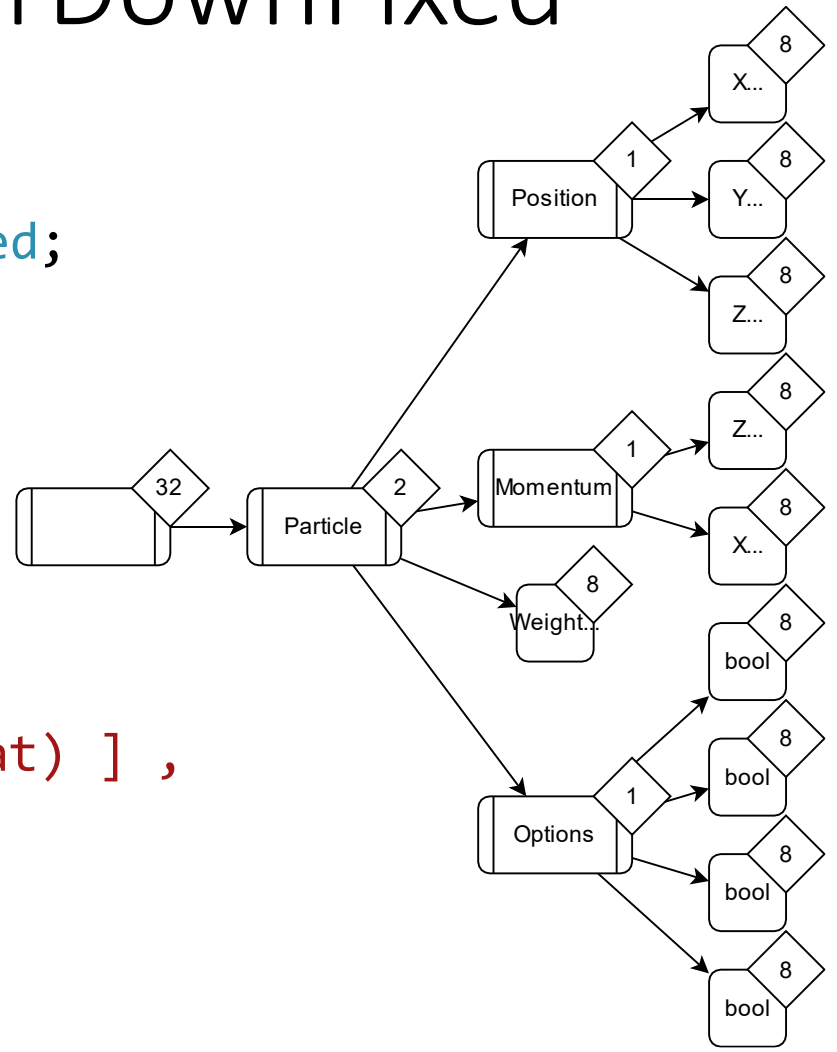


https://godbolt.org/z/VSEK8W

Tree mapping functors - MoveRTDownFixed

```
constexpr auto vectorWidth = 8;
using llama::mapping::tree::functor::MoveRTDownFixed;
using llama::mapping::tree::TreeCoord;
auto operations = llama::Tuple{
  MoveRTDownFixed<TreeCoord<0>, vectorWidth>{},
  MoveRTDownFixed<TreeCoord<0, 0>, vectorWidth>{},
  MoveRTDownFixed<TreeCoord<0, 1>, vectorWidth>{},
  MoveRTDownFixed<TreeCoord<0, 3>, vectorWidth>{};

llama::mapping::tree::toString(mapping.resultTree);
"32 * [ 2 * [
  1 * Pos[ 8 * X(float) , 8 * Y(float) , 8 * Z(float) ] ,
  1 * Momentum[ 8 * Z(double) , 8 * X(double) ] ,
  8 * Weight(int) ,
  1 * Options[ 8 * (bool) , 8 * (bool) ,
               8 * (bool) , 8 * (bool) ] ] ]"
```



https://godbolt.org/z/Fk9zwj

Allocators

- Allocate blobs used as storage underneath a view
- The view stores the blobs created by the allocator
- Theoretically supports multiple blobs
  - View maps data into multiple, disjunct memory regions
- LLAMA supports std::shared_ptr, std::vector and stack-based blobs
- Factories and views can be customized via a template parameter
  - Examples contain an Alpaka allocator that uses a GPU buffer

My critique

- View allocates memory. Allocation is customizable but not separable. LLAMA View should be able to be wrapped over existing memory.
- Huge use of TMP, library is hard to read and maintain.
  - Quite substantial compile times.
  - An external code generator might be faster, easier and more maintainable.
- Data is eventually accessed using indices. No support for iterators or the STL.

- Still, what LLAMA achieves is very impressive!

Observations on the side

- Compilers (g++10) seem to be able to vectorize AoS layouts as well
  1. Vector load some structs from memory
  2. Shuffle struct members in registers
  3. Run fully vectorized computation
  4. Shuffle results back
  5. Vector store result structs to memory
- Example vector addition:
  - AoS: https://godbolt.org/z/fT-VNP
  - SoA: https://godbolt.org/z/53Ngib
  - LLAMA: https://godbolt.org/z/mR7hUo

AoS	SoA	LLAMA SoA
0.211005	0.200032	0.368752
0.212109	0.200078	0.192976
0.211213	0.200037	0.194353
0.217268	0.200422	0.194729
0.216285	0.201179	0.193602
0.206317	0.200432	0.193283
0.206522	0.200459	0.193529
0.205697	0.200131	0.194217
0.204703	0.201265	0.194087
0.216404	0.200670	0.193233

Try it!

- Official repository
  - https://github.com/ComputationalRadiationPhysics/llama
  - C++11
- My fork
  - https://github.com/bernhardmgruber/llama
  - More examples
  - Less code due to intensive refactoring
  - C++17
  - Compiler explorer:
    - #include <https://gist.githubusercontent.com/bernhardmgruber/bdbeae6f6d8eb710637cff61bb369947/raw/8b4fb73eb8e6916e4dea28c14885ea5b6755b942/llama.hpp>