LLAMA Documentation

Release 0.6

Alexander Matthes, Bernhard Manfred Gruber

Feb 29, 2024

USER DOCUMENTATION

LLAMA is a cross-platform C++17/C++20 header-only template library for the abstraction of data layout and memory access. It separtes the view of the algorithm on the memory and the real data layout in the background. This allows for performance portability in applications running on heterogeneous hardware with the very same code.

CHAPTER

ONE

INSTALLATION

1.1 Getting LLAMA

The most recent version of LLAMA can be found at GitHub.

```
git clone https://github.com/alpaka-group/llama
cd llama
```

All examples use CMake and the library itself provides a `llama-config.cmake` to be found by CMake. Although LLAMA is a header-only library, it provides installation capabilities via CMake.

1.2 Dependencies

1.2.1 LLAMA library

At its core, using the LLAMA library requires:

- cmake 3.18.3 or higher
- Boost 1.74.0 or higher
- libfmt 6.2.1 or higher (optional) for support to dump mappings as SVG/HTML

1.2.2 Tests

Building the unit tests additionally requires:

- Catch2 3.0.1 or higher

1.2.3 Examples

To build all examples of LLAMA, the following additional libraries are needed:

- libfmt 6.2.1 or higher
- Alpaka 1.0 or higher
- xsimd 9.0.1 or higher
- ROOT
- tinyobjloader 2.0.0-rc9 or higher

1.3 Build tests and examples

As LLAMA is using CMake the tests and examples can be easily built with:

```
mkdir build
cd build
cmake .. -DBUILD_TESTING=ON -DLLAMA_BUILD_EXAMPLES=ON
ccmake .. // optionally change configuration after first run of cmake
cmake --build .
```

This will search for all dependencies and create a build system for your platform. If necessary dependencies are not found, the corresponding examples will be disabled. After the initial call to *cmake*, *ccmake* can be used to add search paths for missing libraries and to deactivate building tests and examples.

1.4 Install LLAMA

To install LLAMA on your system, you can run (with privileges):

```
cmake --install .
```

CHAPTER

TWO

INTRODUCTION

2.1 Motivation

Current hardware architectures are heterogeneous and it seems they will get even more heterogeneous in the future. A central challenge of today's software development is portability between theses hardware architectures without leaving performance on the table. This often requires separate code paths depending on the target system. But even then, sometimes projects last for decades while new architectures rise and fall, making it is dangerous to settle for a specific data structure.

Performance portable parallelism to exhaust multi-, manycore and GPU hardware is addressed in recent developments like alpaka or Kokkos.

However, efficient use of a system's memory and cache hierarchies is crucial as well and equally heterogeneous. General solutions or frameworks seem not to exist yet. First attempts are AoS/SoA container libraries like SoAx or Intel's SDLT, Kokkos's views or C++23's std::mdspan.

Let's consider an example. Accessing structural data in a struct of array (SoA) manner is most of the times faster than array of structs (AoS):

```
// Array of Struct   |   // Struct of Array
struct               |   struct
{                    |   {
    float r, g, b;   |       float r[64][64], g[64][64], b[64][64];
    char a;          |       char a[64][64];
} image[64][64];     |   } image;
```

Even this small decision between SoA and AoS has a quite different access style in code, `image[x][y].r` vs. `image.r[x][y]`. So the choice of layout is already quite infectious on the code we use to access a data structure. For this specific example, research and ready to use libraries already exist (E.g. SOAContainer or Intel's SDLT).

But there are more useful mappings than SoA and AoS, such as:

- blocking of memory (like partly using SoA inside an AoS approach)
- strided access of data (e.g. odd indexes after each other)
- padding
- separating frequently accessed data from the rest (hot/cold data separation)
- ...

Moreover, software is often using various heterogeneous memory architectures such as RAM, VRAM, caches, memory-mapped devices or files, etc. A data layout optimized for a specific CPU may be inefficient on a GPU or only slowly transferable over network. A single layout – not optimal for each architecture – is very often a trade-off. An optimal layout is highly dependent on the architecture, the scaling of the problem and of course the chosen algorithm.

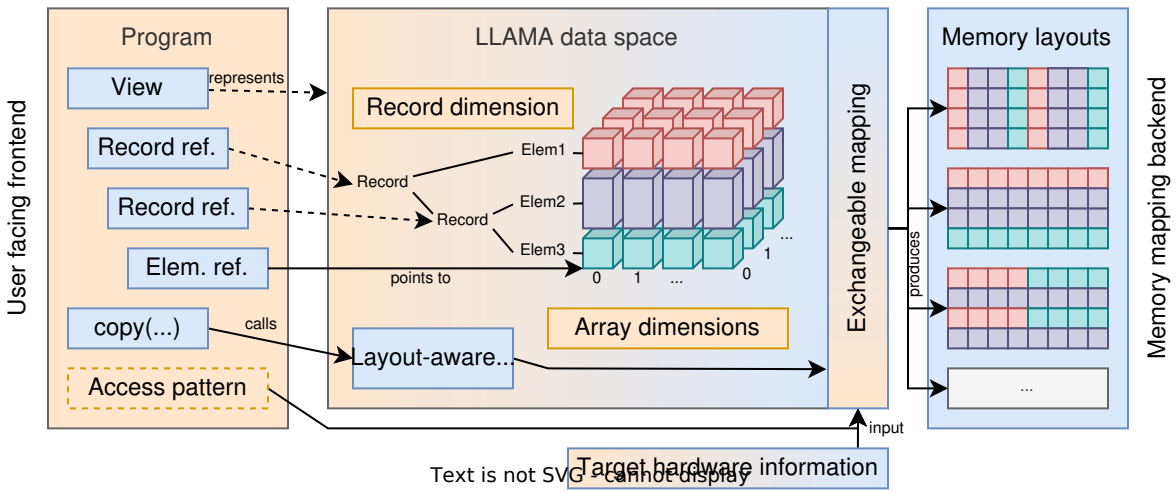
Furthermore, third party libraries may expect specific memory layouts at their interface, into which custom data structures need to be converted.

2.2 Goals

LLAMA tries to achieve the following goals:

- Allow users to express a generic data structure independently of how it is stored. Consequently, algorithms written against this data structure's interface are not bound to the data structure's layout in memory. This requires a data layout independent way to access the data structure.
- Provide generic facilities to map the user-defined data structure into a performant data layout. Also allowing specialization of this mapping for specific data structures by the user. A data structure's mapping is set and resolved statically at compile time, thus guaranteeing the same performance as manually written versions of a data structure.
- Enable efficient, high throughput copying between different data layouts of the same data structure, which is a necessity in heterogeneous systems. This requires meta data on the data layout. Deep copies are the focus, although LLAMA should include the possibility for zero copies and in-situ transformation of data layouts. Similar strategies could be adopted for message passing and copies between file systems and memory. (WIP)
- To be compatible with many architectures, other software packages, compilers and third party libraries, LLAMA tries to stay within C++17/C++20. No separate description files or language is used.
- LLAMA should work well with auto vectorization approaches of modern compilers, but also support explicit vectorization on top of LLAMA.

2.3 Concept

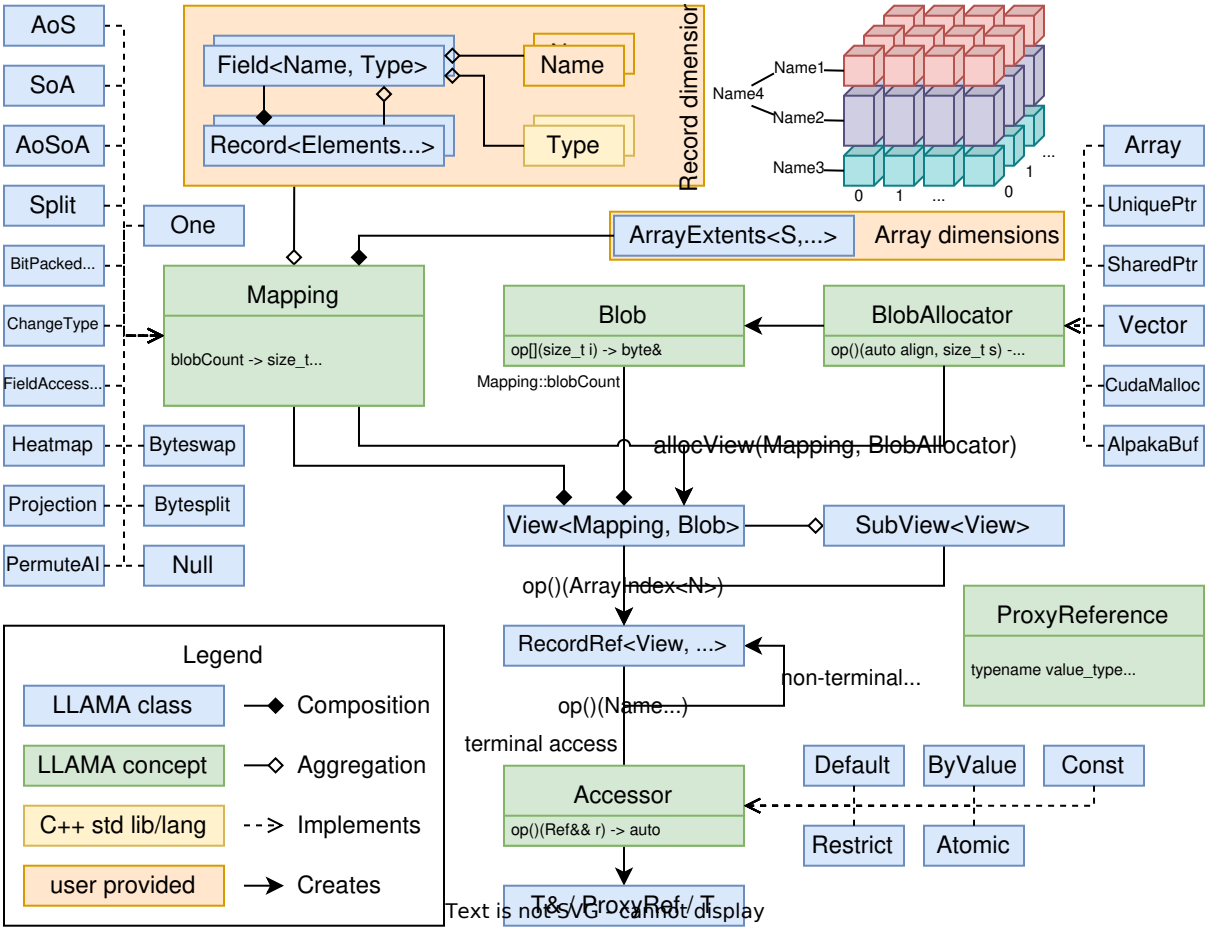


LLAMA separates the data structure access and physical memory layout by an opaque abstract data type called data space. The data space is an hypercubic index set described by the record dimension and one or more array dimensions. The record dimension consists of a hierarchy of names and describes nested, structured data, much like a `struct` in C++. The array dimensions are zero-based integral ranges. Programs are written against this abstract data space and thus formulated independent of the physical manifestation of the data space. Programs can refer to subparts of the data space via record references or real l-value references. The data space is materialized via a mapping that describes how the index set of the data space is embedded into a physical memory. This mapping is exchangeable at compile time and

can be augmented with additional information from the programs access pattern and target hardware information. Due to a mapping encapsulating the full knowledge of a memory layout, LLAMA supports layout aware copies between instances of the same data space but with different mappings.

2.4 Library overview

The following diagram gives an overview over the components of LLAMA:



The core data structure of LLAMA is the *View*, which holds the memory for the data and provides methods to access the data space. In order to create a view, a *Mapping* is needed which is an abstract concept. LLAMA offers many kinds of mappings and users can also provide their own mappings. Mappings are constructed from a *record dimension*, containing tags, and *array dimensions*. In addition to a mapping defining the memory layout, an array of *Blobs* is needed for a view, supplying the actual storage behind the view. A blob is any object representing a contiguous chunk of memory, byte-wise addressable using `operator[]`. A suitable Blob array is either directly provided by the user or built using a *BlobAllocator* when a view is created by a call to *allocView*. A blob allocator is again an abstract concept and any object returning a blob of a requested size when calling `operator()`. LLAMA comes with a set of predefined blob allocators and users can again provider their own.

Once a view is created, the user can navigate on the data managed by the view. On top of a view, a *SubView* can be created, offering access to a subspace of the array dimensions. Elements of the array dimensions, called records, are accessed on both, View and SubView, by calling `operator()` with an array index as instance of `ArrayIndex`. This access returns a *RecordRef*, allowing further access using the tags from the record dimension, until eventually a reference to actual data in memory is returned.

2.5 Example use cases

This library is designed and written by the software development for experiments group (EP-SFT) at CERN, by the group for computational radiation physics (CRP) at HZDR and CASUS. While developing, we have some in house and partner applications in mind. These example use cases are not the only targets of LLAMA, but drove the development and the feature set.

One of the major projects in EP-SFT is the ROOT data analysis framework for data analysis in high-energy physics. A critical component is the fast transfer of petabytes of filesystem data taken from CERN's detectors into an efficient in-memory representation for subsequent analysis algorithms. This data are particle interaction events, each containing a series of variable size attributes. A typical analysis involves column selection, cuts, filters, computation of new attributes and histograms. The data in ROOT files is stored in columnar blocks and significant effort is made to make the data flow and aggregation as optimal as possible. LLAMA will supply the necessary memory layouts for an optimal analysis and automate the data transformations from disk into these layouts.

The CRP group works on a couple of simulation codes, e.g. PIConGPU, the fastest particle in cell code running on GPUs. Recent development efforts furthermore made the open source project ready for other many core and even classic CPU multi core architectures using the library alpaka. The similar namings of alpaka and LLAMA are no coincidence. While alpaka abstracts the parallelization of computations, LLAMA abstracts the memory access. To get the best out of computational resources, accelerating data structures and a mix of SoA and AoS known to perform well on GPUs is used. The goal is to abstract these data structures with LLAMA to be able to change them fast for different architectures.

Image processing is another big, emerging task of the group and partners. Both, post processing of diffraction images as well as live analysis of high rate data sources, will be needed in the near future. As with the simulation codes, the computation devices, the image sensor data format and the problem size may vary and a fast and easy adaption of the code is needed.

The shipped examples of LLAMA try to showcase the implemented feature in the intended usage.

CHAPTER

THREE

DIMENSIONS

As mentioned in the section before, LLAMA distinguishes between the array and the record dimensions. The most important difference is that the array dimensions are defined at compile or *run time* whereas the record dimension is defined fully at *compile time*. This allows to make the problem size itself a run time value but leaves the compiler room to optimize the data access.

3.1 Array dimensions

The array dimensions form an N-dimensional array with N itself being a compile time value. The extent of each dimension can be a compile time or runtime values.

A simple definition of three array dimensions of the extents $128 \times 256 \times 32$ looks like this:

```
llama::ArrayExtents extents{128, 256, 32};
```

The template arguments are deduced by the compiler using Class Template Argument Deduction (CTAD). The full type of `extents` is `llama::ArrayExtents<int, llama::dyn, llama::dyn, llama::dyn>`.

By explicitly specifying the template arguments, we can mix compile time and runtime extents, where the constant `llama::dyn` denotes a dynamic extent:

```
llama::ArrayExtents<int, llama::dyn, 256, llama::dyn> extents{128, 32};
```

The template argument list specifies the integral type used for index calculations and the order and nature (compile vs. runtime) of the extents. Choosing the right index type depends on the possible magnitude of values occurring during index calculations (e.g. `int` only allows a maximum flat index space and blob size of `INT_MAX`), as well as target specific optimization aspects (e.g. `size_t` consuming more CUDA registers than *unsigned int*). An instance of `llama::ArrayExtents` can then be constructed with as many runtime extents as `llama::dyn`s specified in the template argument list.

By setting a specific value for all template arguments, the array extents are fully determined at compile time.

```
llama::ArrayExtents<int, 128, 256, 32> extents{};
```

This is important if such extents are later embedded into other LLAMA objects such as mappings or views, where they should not occupy any additional memory.

```
llama::ArrayExtents<int, 128, 256, 32> extents{};
static_assert(std::is_empty_v<decltype(extents)>);

struct S : llama::ArrayExtents<int, 128, 256, 32> { char c; } s;
static_assert(sizeof(s) == sizeof(char)); // empty base optimization eliminates storage
```

To later described indices into the array dimensions described by a `llama::ArrayExtents`, an instance of `llama::ArrayIndex` is used:

```
llama::ArrayIndex i{2, 3, 4};
// full type of i: llama::ArrayIndex<int, 3>
```

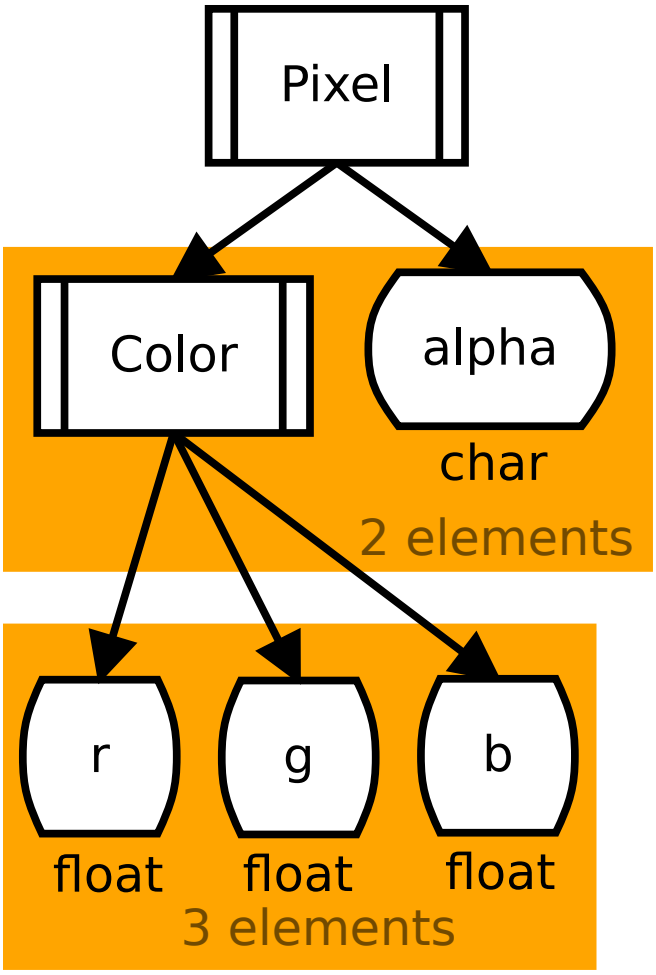
Contrary to `llama::ArrayExtents` which can store a mix of compile and runtime values, `llama::ArrayIndex` only stores runtime indices, so it is templated on the number of dimensions. This might change at some point in the future, if we find sufficient evidence that a design similar to `llama::ArrayExtents` is also useful for `llama::ArrayIndex`.

3.2 Record dimension

The record dimension is a tree structure completely defined at compile time. Nested C++ structs, which the record dimension tries to abstract, they are trees too. Let's have a look at this simple example struct for storing a pixel value:

```
struct Pixel {
    struct {
        float r
        float g
        float b;
    } color;
    char alpha;
};
```

This defines this tree



Unfortunately with C++ it is not possible yet to "iterate" over a struct at compile time and extract member types and names, as it would be needed for LLAMA's mapping (although there are proposals to provide such a facility). For now LLAMA needs to define such a tree itself using two classes, `llama::Record` and `llama::Field`. `llama::Record` is a compile time list of `llama::Field`. `llama::Field` has a name and a fundamental type **or** another `llama::Record` list of child `llama::Fields`. The name of a `llama::Field` needs to be C++ type as well. We recommend creating empty tag types for this. These tags serve as names when describing accesses later. Furthermore, these tags also enable a semantic binding even between two different record dimensions.

To make the code easier to read, the following shortcuts are defined:

- `llama::Record` → `llama::Record`
- `llama::Field` → `llama::Field`

A record dimension itself is just a `llama::Record` (or a fundamental type), as seen here for the given tree:

```
struct color {};
struct alpha {};
struct r {};
struct g {};
struct b {};

using RGB = llama::Record<
    llama::Field<r, float>,
```

(continues on next page)

(continued from previous page)

```
    llama::Field<g, float>,
    llama::Field<b, float>
>;
using Pixel = llama::Record<
    llama::Field<color, RGB>,
    llama::Field<alpha, char>
>;
```

Arrays of compile-time extent are also supported as arguments to `llama::Field`. Such arrays are expanded into a `llama::Record` with multiple `llama::Field`s of the same type. E.g. `llama::Field<Tag, float[4]>` is expanded into

```
llama::Field<Tag, llama::Record<
    llama::Field<llama::RecordCoord<0>, float>,
    llama::Field<llama::RecordCoord<1>, float>,
    llama::Field<llama::RecordCoord<2>, float>,
    llama::Field<llama::RecordCoord<3>, float>
>>
```

CHAPTER

FOUR

VIEW

The view is the main data structure a LLAMA user will work with. It takes coordinates in the array and record dimensions and returns a reference to a record in memory which can be read from or written to. For easier use, some useful operations such as += are overloaded to operate on all record fields inside the record dimension at once.

4.1 View allocation

A view is allocated using the helper function `allocView`, which takes a *mapping* and an optional *blob allocator*.

```
using Mapping = ...; // see next section about mappings
Mapping mapping(extents); // see section about dimensions
auto view = allocView(mapping); // optional blob allocator as 2nd argument
```

The *mapping* and *blob allocator* will be explained later. For now, it is just important to know that all those run time and compile time parameters come together to create the view.

4.2 Data access

LLAMA tries to have an array of struct like interface. When accessing an element of the view, the array part comes first, followed by tags from the record dimension.

In C++, runtime values like the array dimensions coordinates are normal function parameters whereas compile time values such as the record dimension tags are usually given as template arguments. However, compile time information can be stored in a type, instantiated as a value and then passed to a function template deducing the type again. This trick allows to pass both, runtime and compile time values as function arguments. E.g. instead of calling `f<MyType>()` we can call `f(MyType{})` and let the compiler deduce the template argument of `f`.

This trick is used in LLAMA to specify the access to a value of a view. An example access with the dimensions defined in the *dimensions section* could look like this:

```
view(1, 2, 3)(color{}, g{}) = 1.0;
```

It is also possible to access the array dimensions with one compound argument like this:

```
const llama::ArrayIndex pos{1, 2, 3};
view(pos)(color{}, g{}) = 1.0;
// or
view({1, 2, 3})(color{}, g{}) = 1.0;
```

The values `color{}` and `g{}` are not used and just serve as a way to specify the template arguments. Alternatively, an addressing with integral record coordinates is possible like this:

```
view(1, 2, 3)(llama::RecordCoord<0, 1>{}) = 1.0; // color.g
```

These record coordinates are zero-based, nested indices reflecting the nested tuple-like structure of the record dimension.

Notice that the `operator()` is invoked twice in the last example and that an intermediate object is needed for this to work. This object is a `llama::RecordRef`.

4.3 Accessors

An Accessor is a callable that a view invokes on the mapped memory reference returned from a mapping. Accessors can be specified when a view is created or changed later.

```
auto view  = llama::allocView(mapping, llama::bloballoc::Vector{},
                 llama::accessor::Default{});
auto view2 = llama::withAccessor(view,
                 llama::accessor::Const{}); // view2 is a copy!
```

Switching an accessor changes the type of a view, so a new object needs to be created as a copy of the old one. To prevent the blobs to be copied, either use a corresponding blob allocator, or shallow copy the view before changing its accessor.

```
auto view3 = llama::withAccessor(std::move(view),
                 llama::accessor::Const{}); // view3 contains blobs of view now
auto view4 = llama::withAccessor(llama::shallowCopy(view3),
                 llama::accessor::Const{}); // view4 shares blobs with view3
```

4.4 SubView

Sub views can be created on top of existing views, offering shifted access to a subspace of the array dimensions.

```
auto view = ...;
llama::SubView subView{view, {10, 20, 30}};
subView(1, 2, 3)(color{}, g{}) = 1.0; // accesses record {11, 22, 33}
```

CHAPTER

FIVE

RECORDREF

During a view accesses like `view(1, 2, 3)(color{}, g{})` an intermediate object is needed for this to work. This object is a `llama::RecordRef`.

```
using Pixel = llama::Record<
    llama::Field<color, llama::Record<
        llama::Field<r, float>,
        llama::Field<g, float>,
        llama::Field<b, float>
    >>,
    llama::Field<alpha, char>
>;
// ...

auto vd = view(1, 2, 3);

vd(color{}, g{}) = 1.0;
// or:
auto vdColor = vd(color{});
float& g = vdColor(g{});
g = 1.0;
```

Supplying the array dimensions coordinate to a view access returns such a `llama::RecordRef`, storing this array dimensions coordinate. This object models a reference to a record in the N-dimensional array dimensions space, but as the fields of this record may not be contiguous in memory, it is not a native l-value reference.

Accessing subparts of a `llama::RecordRef` is done using `operator()` and the tag types from the record dimension.

If an access describes a final/leaf element in the record dimension, a reference to a value of the corresponding type is returned. Such an access is called terminal. If the access is non-terminal, i.e. it does not yet reach a leaf in the record dimension tree, another `llama::RecordRef` is returned, binding the tags already used for navigating down the record dimension.

A `llama::RecordRef` can be used like a real local object in many places. It can be used as a local variable, copied around, passed as an argument to a function (as seen in the nbody example), etc. In general, `llama::RecordRef` is a value type that represents a reference, similar to an iterator in C++ (`llama::One` is a notable exception).

5.1 One

`llama::One<RecordDim>` is a shortcut to create a scalar `llama::RecordRef`. This is useful when we want to have a single record instance e.g. as a local variable.

```
llama::One<Pixel> pixel;
pixel(color{}, g{}) = 1.0;
auto pixel2 = pixel; // independent copy
```

Technically, `llama::One` is a `llama::RecordRef` which stores a scalar `llama::View` inside, using the mapping `llama::mapping::One`. This also has the consequence that a `llama::One` is now a value type with deep-copy semantic.

5.2 Arithmetic and logical operators

`llama::RecordRef` overloads several operators:

```
auto record1 = view(1, 2, 3);
auto record2 = view(3, 2, 1);

record1 += record2;
record1 *= 7.0; //for every element in the record dimension

foobar(record2);

//With this somewhere else:
template<typename RecordRef>
void foobar(RecordRef vr)
{
    vr = 42;
}
```

The assignment operator (=) and the arithmetic, non-bitwise, compound assignment operators (=, +=, -=, *=, /=, %=) are overloaded. These operators directly write into the corresponding view. Furthermore, the binary, non-bitwise, arithmetic operators (+, -, *, /, %) are overloaded too, but they return a temporary object on the stack (i.e. a `llama::One`).

These operators work between two record references, even if they have different record dimensions. Every tag existing in both record dimensions will be matched and operated on. Every non-matching tag is ignored, e.g.

```
using RecordDim1 = llama::Record<
    llama::Record<llama::Field<pos
        llama::Field<x, float>
    >>,
    llama::Record<llama::Field<vel
        llama::Field <x, double>
    >>,
    llama::Field <x, int>
>;

using RecordDim2 = llama::Record<
    llama::Record<llama::Field<pos
        llama::Field<x, double>
```

(continues on next page)

(continued from previous page)

```
    >>,
    llama::Record<llama::Field<mom
        llama::Field<x, double>
    >>
>;

// Let assume record1 using RecordDim1 and record2 using RecordDim2.

record1 += record2;
// record2.pos.x will be added to record1.pos.x because
// of pos.x existing in both record dimensions although having different types.

record1(vel{}) *= record2(mom{});
// record2.mom.x will be multiplied to record2.vel.x as the first part of the
// record dimension coord is explicit given and the same afterwards
```

The discussed operators are also overloaded for types other than `llama::RecordRef` as well so that `record1 *= 7.0` will multiply 7 to every element in the record dimension. This feature should be used with caution!

The comparison operators ==, !=, <, <=, > and >= are overloaded too and return `true` if the operation is true for **all** pairs of fields with equal tag. Let's examine this deeper in an example:

```
using A = llama::Record <
    llama::Field < x, float >,
    llama::Field < y, float >
>;

using B = llama::Record<
    llama::Field<z, double>,
    llama::Field<x, double>
>;

bool result;

llama::One<A> a1, a2;
llama::One<B> b;

a1(x{}) = 0.0f;
a1(y{}) = 2.0f;

a2 = 1.0f; // sets x and y to 1.0f

b(x{}) = 1.0f;
b(z{}) = 2.0f;

result = a1 < a2;
//result is false, because a1.y > a2.y

result = a1 > a2;
//result is false, too, because now a1.x > a2.x

result = a1 != a2;
//result is true
```

(continues on next page)

(continued from previous page)

```
result = a2 == b;
//result is true, because only the matching "x" matters
```

A partial addressing of a record reference like `record1(color{}) *= 7.0` is also possible. `record1(color{})` itself returns a new record reference with the first record dimension coordinate (`color`) being bound. This enables e.g. to easily add a velocity to a position like this:

```
using Particle = llama::Record<
    llama::Field<pos, llama::Record<
        llama::Field<x, float>,
        llama::Field<y, float>,
        llama::Field<z, float>
    >>,
    llama::Field<vel, llama::Record<
        llama::Field<x, double>,
        llama::Field<y, double>,
        llama::Field<z, double>
    >>,
>;

// Let record be a record reference with the record dimension "Particle".

record(pos{}) += record(vel{});
```

5.3 Tuple interface

A struct in C++ can be modelled by a `std::tuple` with the same types as the struct's members. A `llama::RecordRef` behaves like a reference to a struct (i.e. the record) which is decomposed into it's members. We can therefore not form a single reference to such a record, but references to the individual members. Organizing these references inside a `std::tuple` in the same way the record is represented in the record dimension gives us an alternative to a `llama::RecordRef`. Mind that creating such a `std::tuple` already invokes the mapping function, regardless of whether an actual memory access occurs through the constructed reference later. However, such dead address computations are eliminated by most compilers during optimization.

```
auto record = view(1, 2, 3);
std::tuple<std::tuple<float&, float&, float&>, char&> = record.asTuple();
std::tuple<float&, float&, float&, char&> = record.asFlatTuple();
auto [r, g, b, a] = record.asFlatTuple();
```

Additionally, if the user already has types supporting the C++ tuple interface, `llama::RecordRef` can integrate with these using the `load()`, `loadAs<T>()` and `store(T)` functions.

```
struct MyPixel {
    struct {
        float r, g, b;
    } color;
    char alpha;
};
// implement std::tuple_size<MyPixel>, std::tuple_element<MyPixel> and get(MyPixel)
```

(continues on next page)

(continued from previous page)

```
auto record = view(1, 2, 3);

MyPixel p1 = record.load(); // constructs MyPixel from 3 float& and 1 char&
auto p2 = record.loadAs<MyPixel>(); // same

p1.alpha = 255;
record.store(p1); // tuple-element-wise assignment from p1 to record.asFlatTuple()
```

Keep in mind that the load and store functionality always reads/writes all elements referred to by a `llama::RecordRef`.

5.4 Structured bindings

A `llama::RecordRef` implements the C++ tuple interface itself to allow destructuring:

```
auto record = view(1, 2, 3);
auto [color, a] = record; // color is another RecordRef, a is a char&, 1 call to mapping␣
↪function
auto [r, g, b] = color; // r, g, b are float&, 3 calls to mapping function
```

Contrary to destructuring a tuple generated by calling `asTuple()` or `asFlatTuple()`, the mapping function is not invoked for other instances of `llama::RecordRef` created during the destructuring. The mapping function is just invoked to form references for terminal accesses.

CHAPTER

SIX

ITERATION

6.1 Array dimensions iteration

The array dimensions span an N-dimensional space of integral indices. Sometimes we just want to quickly iterate over all coordinates in this index space. This is what `llama::ArrayIndexRange` is for, which is a range in the C++ sense and offers the `begin()` and `end()` member functions with corresponding iterators to support STL algorithms or the range-for loop.

```
llama::ArrayIndexRange range{llama::ArrayIndex{3, 3}};

std::for_each(range.begin(), range.end(), [](llama::ArrayIndex<2> ai) {
    // ai is {0, 0}, {0, 1}, {0, 2}, {1, 0}, {1, 1}, {1, 2}, {2, 0}, {2, 1}, {2, 2}
});

for (auto ai : range) {
    // ai is {0, 0}, {0, 1}, {0, 2}, {1, 0}, {1, 1}, {1, 2}, {2, 0}, {2, 1}, {2, 2}
}
```

6.2 Record dimension iteration

The record dimension is iterated using `llama::forEachLeafCoord`. It takes a record dimension as template argument and a callable with a generic parameter as argument. This function's `operator()` is then called for each leaf of the record dimension tree with a record coord as argument. A polymorphic lambda is recommended to be used as a functor.

```
llama::forEachLeafCoord<Pixel>([&](auto rc) {
    // rc is RecordCoord <0, 0 >{}, RecordCoord <0, 1>{}, RecordCoord <0, 2>{} and␣
↪RecordCoord <1>{}
});
```

Optionally, a subtree of the record dimension can be chosen for iteration. The subtree is selected either via a *RecordCoord* or a series of tags.

```
llama::forEachLeafCoord<Pixel>([&](auto rc) {
    // rc is RecordCoord <0, 0 >{}, RecordCoord <0, 1>{} and RecordCoord <0, 2>{}
}, color{});

llama::forEachLeafCoord<Pixel>([&](auto rc) {
    // rc is RecordCoord <0, 1>{}
}, color{}, g{});
```

6.3 View iterators

Iterators on views of any dimension are supported and open up the standard library for use in conjunction with LLAMA:

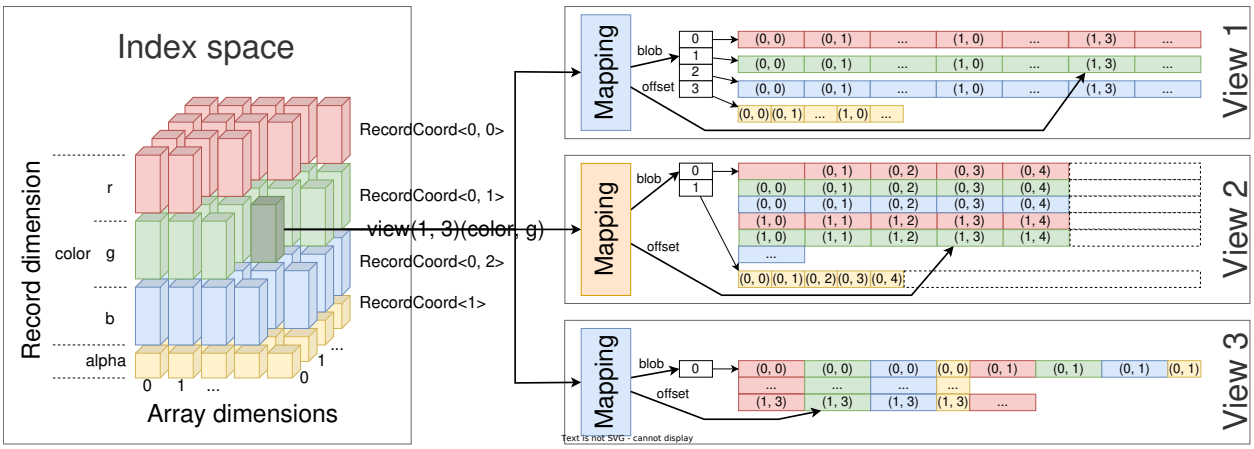
```
using Pixel = ...;
using ArrayExtents = llama::ArrayExtents<std::size_t, llama::dyn>;
// ...
auto view = llama::allocView(mapping);
// ...

// range for
for (auto vd : view)
    vd(color{}, r{}) = 1.0f;

auto view2 = llama::allocView (...); // with different mapping

// layout changing copy
std::copy(begin(aosView), end(aosView), begin(soaView));

// transform into other view
std::transform(begin(view), end(view), begin(view2), [](auto vd) { return vd(color{}) *␣
↪2; });

// accumulate using One as accumulator and destructure result
const auto [r, g, b] = std::accumulate(begin(view), end(view), One<RGB>{},
    [](auto acc, auto vd) { return acc + vd(color{}); });

// C++20:
for (auto x : view | std::views::transform([](auto vd) { return vd(x{}); }) |␣
↪std::views::take(2))
    // ...
```

CHAPTER

SEVEN

MAPPINGS

One of the core tasks of LLAMA is to map an address from the array and record dimensions to some address in the allocated memory space. This is particularly challenging if the compiler shall still be able to optimize the resulting memory accesses (vectorization, reordering, aligned loads, etc.). The compiler needs to **understand** the semantic of the mapping at compile time. Otherwise the abstraction LLAMA provides will perform poorly. Thus, mappings are compile time parameters to LLAMA's views (and e.g. not hidden behind a virtual dispatch). LLAMA provides several ready-to-use mappings, but users are also free to supply their own mappings.



LLAMA supports and uses different classes of mapping that differ in their usage:

7.1 Physical mappings

A physical mapping is the primary form of a mapping. Mapping a record coordinate and array dimension index through a physical mapping results in a blob number and offset. This information is then used either by a view or subsequent mapping and, given a blob array, can be turned into a physical memory location, which is provided as l-value reference to the mapped field type of the record dimension.

7.2 Computed mappings

A computed mapping may invoke a computation to map a subset of the record dimension. The fields of the record dimension which are mapped using a computation, are called computed fields. A computed mapping does not return a blob number and offset for computed fields, but rather a reference to memory directly. However, this reference is not an l-value reference but a *proxy reference*, since this reference needs to encapsulate computations to be performed when reading or writing through the reference. For non-computed fields, a computed mapping behaves like a physical mapping. A mapping with only computed fields is called a fully computed mapping, otherwise a partially computed mapping.

7.3 Meta mappings

A meta mapping is a mapping that builds on other mappings. Examples are altering record or array dimensions before passing the information to another mapping or modifying the blob number and offset returned from a mapping. A meta mapping can also instrument or trace information on the accesses to another mapping. Meta mappings are orthogonal to physical and computed mappings.

7.3.1 Concept

A LLAMA mapping is used to create views as detailed in the *allocView API section* and views consult the mapping when resolving accesses. The view requires each mapping to fulfill at least the following concept:

```
template <typename M>
concept Mapping = requires(M m) {
    typename M::ArrayExtents;
    typename M::RecordDim;
    { m.extents() } -> std::same_as<typename M::ArrayExtents>;
    { +M::blobCount } -> std::same_as<std::size_t>;
    requires isConstexpr<M::blobCount>;
    { m.blobSize(std::size_t{}) } -> std::same_as<typename M::ArrayExtents::value_type>;
};
```

That is, each mapping type needs to expose the types `ArrayExtents` and `RecordDim`. Each mapping also needs to provide a getter *extents()* to retrieve the runtime value of the `ArrayExtents` held by the mapping, and provide a `static constexpr` member variable `blobCount`. Finally, the member function `blobSize(i)` gives the size in bytes of the `i`th block of memory needed for this mapping using the value type of the array extents. `i` is in the range of `0` to `blobCount - 1`.

Additionally, a mapping needs to be either a physical or a computed mapping. Physical mappings, in addition to being mappings, need to fulfill the following concept:

```
template <typename M>
concept PhysicalMapping = Mapping<M> && requires(M m, typename M::ArrayIndex ai,␣
↪RecordCoord<> rc) {
    { m.blobNrAndOffset(ai, rc) } -> std::same_as<NrAndOffset<typename␣
↪M::ArrayExtents::value_type>>;
};
```

That is, they must provide a member function callable as `blobNrAndOffset(ai, rc)` that implements the core mapping logic, which is translating an array index `ai` and record coordinate `rc` into a value of `llama::NrAndOffset`, containing the blob number of offset within the blob where the value should be stored. The integral type used for computing blob number and offset should be the value type of the array extents.

7.3.2 AoS

LLAMA provides a family of AoS (array of structs) mappings based on a generic implementation. AoS mappings keep the data of a single record close together and therefore maximize locality for accesses to an individual record. However, they do not vectorize well in practice.

```
llama::mapping::AoS<ArrayExtents, RecordDim> mapping{extents};
llama::mapping::AoS<ArrayExtents, RecordDim, false> mapping{extents}; // pack fields␣
↪(violates alignment)
llama::mapping::AoS<ArrayExtents, RecordDim, false
    llama::mapping::LinearizeArrayIndexLeft> mapping{extents}; // pack fields, column␣
↪major
```

By default, the array dimensions spanned by `ArrayExtents` are linearized using `llama::mapping::LinearizeArrayIndexRight`. LLAMA provides the aliases `llama::mapping::AlignedAoS` and `llama::mapping::PackedAoS` for convenience.

7.3.3 SoA

LLAMA provides a family of SoA (struct of arrays) mappings based on a generic implementation. SoA mappings store the attributes of a record contiguously and therefore maximize locality for accesses to the same attribute of multiple records. This layout auto-vectorizes well in practice.

```
llama::mapping::SoA<ArrayExtents, RecordDim> mapping{extents};
llama::mapping::SoA<ArrayExtents, RecordDim, true> mapping{extents}; // separate blob␣
↪for each attribute
llama::mapping::SoA<ArrayExtents, RecordDim, true,
    llama::mapping::LinearizeArrayIndexLeft> mapping{extents}; // separate blob for each␣
↪attribute, column major
```

By default, the array dimensions spanned by `ArrayExtents` are linearized using `llama::mapping::LinearizeArrayIndexRight` and the layout is mapped into a single blob. LLAMA provides the aliases `llama::mapping::SingleBlobSoA` and `llama::mapping::MultiBlobSoA` for convenience.

7.3.4 AoSoA

There are also combined AoSoA (array of struct of arrays) mappings. Since the mapping code is more complicated, compilers currently fail to auto vectorize view access. We are working on this. The AoSoA mapping has a mandatory additional parameter specifying the number of elements which are blocked in the inner array of AoSoA.

```
llama::mapping::AoSoA<ArrayExtents, RecordDim, 8> mapping{extents}; // inner array has 8␣
↪values
llama::mapping::AoSoA<ArrayExtents, RecordDim, 8,
    llama::mapping::LinearizeArrayIndexLeft> mapping{extents}; // inner array has 8␣
↪values, column major
```

By default, the array dimensions spanned by `ArrayExtents` are linearized using `llama::mapping::LinearizeArrayIndexRight`.

LLAMA also provides a helper `llama::mapping::maxLanes` which can be used to determine the maximum vector lanes which can be used for a given record dimension and vector register size. In this example, the inner array a size of N so even the largest type in the record dimension can fit N times into a vector register of 256bits size (e.g. AVX2).

```
llama::mapping::AoSoA<ArrayExtents, RecordDim,
    llama::mapping::maxLanes<RecordDim, 256>> mapping{extents};
```

7.3.5 One

The One mapping is intended to map all coordinates in the array dimensions onto the same memory location. This is commonly used in `llama::One`, but also offers interesting applications in conjunction with the `llama::mapping::Split` mapping.

7.3.6 Split

The Split mapping is a meta mapping. It transforms the record dimension and delegates mapping to other mappings. Using a record coordinate, a tag list, or a list of record coordinates or a list of tag lists, a subtree of the record dimension is selected and mapped using one mapping. The remaining record dimension is mapped using a second mapping.

```
llama::mapping::Split<ArrayExtents, RecordDim,
    llama::RecordCoord<1>, llama::mapping::SoA, llama::mapping::PackedAoS>
        mapping{extents}; // maps the subtree at index 1 as SoA, the rest as packed AoS
```

Split mappings can be nested to map a record dimension into even fancier combinations.

7.3.7 Heatmap

The Heatmap mapping is a meta mapping that wraps over an inner mapping and counts all accesses made to all bytes. A script for gnuplot visualizing the heatmap can be extracted.

```
auto anyMapping = ...;
llama::mapping::Heatmap mapping{anyMapping};
...
mapping.writeGnuplotDataFileBinary(view.blobs(), std::ofstream{"heatmap.data",␣
↪std::ios::binary});
std::ofstream{"plot.sh"} << mapping.gnuplotScriptBinary;
```

7.3.8 FieldAccessCount

The FieldAccessCount mapping is a meta mapping that wraps over an inner mapping and counts all accesses made to the fields of the record dimension. A report is printed to stdout when requested. The mapping adds an additional blob to the blobs of the inner mapping used as storage for the access counts.

```
auto anyMapping = ...;
llama::mapping::FieldAccessCount mapping{anyMapping};
...
mapping.printFieldHits(view.blobs()); // print report with read and writes to each field
```

The FieldAccessCount mapping uses proxy references to instrument reads and writes. If this is problematic, it can also be configured to return raw C++ references. In that case, only the number of memory location computations can be traced, but not how often the program reads/writes to those locations. Also, the data type used to count accesses is configurable.

```
auto anyMapping = ...;
llama::mapping::FieldAccessCount<decltype(anyMapping), std::size_t, false> mapping
↪{anyMapping};
```

7.3.9 Null

The Null mappings is a fully computed mapping that maps all elements to nothing. Writing data through a reference obtained from the Null mapping discards the value. Reading through such a reference returns a default constructed object. A Null mapping requires no storage and thus its `blobCount` is zero.

```
llama::mapping::Null<ArrayExtents, RecordDim> mapping{extents};
```

7.3.10 Bytesplit

The Bytesplit mapping is a computed meta mapping that wraps over an inner mapping. It transforms the record dimension by replacing each field type by a byte array of the same size before forwarding the record dimension to the inner mapping.

```
template <typename RecordDim, typename ArrayExtents>
using InnerMapping = ...;

llama::mapping::Bytesplit<ArrayExtents, RecordDim, InnerMapping>
        mapping{extents};
```

7.3.11 Byteswap

The Byteswap mapping is a computed meta mapping that wraps over an inner mapping. It swaps the bytes of all values when reading/writing.

```
template <typename RecordDim, typename ArrayExtents>
using InnerMapping = ...;

llama::mapping::Byteswap<ArrayExtents, RecordDim, InnerMapping>
        mapping{extents};
```

7.3.12 ChangeType

The ChangeType mapping is a computed meta mapping that allows to change data types of several fields in the record dimension before and mapping the adapted record dimension with a further mapping.

```
template <typename RecordDim, typename ArrayExtents>
using InnerMapping = ...;

using ReplacementMap = mp_list<
    mp_list<int, short>,
    mp_list<double, float>
>;
llama::mapping::ChangeType<ArrayExtents, RecordDim, InnerMapping, ReplacementMap>
        mapping{extents};
```

In this example, all fields of type **int** in the record dimension will be stored as **short**, and all fields of type **double** will be stored as **float**. Conversion between the data types is done on loading and storing through a proxy reference returned from the mapping.

7.3.13 Projection

The Projection mapping is a computed meta mapping that allows to apply a function on load/store from/two selected fields in the record dimension. These functions are allowed to change the data type of fields in the record dimension. The modified record dimension is then mapped with a further mapping.

```
template <typename RecordDim, typename ArrayExtents>
using InnerMapping = ...;

struct Sqrt {
    static auto load(float v) -> double {
        return std::sqrt(v);
    }

    static auto store(double d) -> float {
        return static_cast<float>(d * d);
    }
};

using ReplacementMap = mp_list<
    mp_list<double, Sqrt>,
    mp_list<RecordCoord<0, 1>, Sqrt>
>;
llama::mapping::ChangeType<ArrayExtents, RecordDim, InnerMapping, ReplacementMap>
        mapping{extents};
```

In this example, all fields of type **double**, and the field at coordinate RecordCoord<0, 1>, in the record dimension will store the product with itself as **float**. The load/store functions are called on loading and storing through a proxy reference returned from the mapping.

7.3.14 BitPackedIntAoS/BitPackedIntSoA

The BitPackedIntSoA and BitPackedIntAoS mappings are fully computed mappings that bitpack integral values to reduce size and precision. The bits are stored as array of structs and struct of arrays, respectively. The number of bits used per integral is configurable. All field types in the record dimension must be integral.

```
unsigned bits = 7;
llama::mapping::BitPackedIntSoA<ArrayExtents, RecordDim>
        mapping{bits, extents}; // use 7 bits for each integral in RecordDim
```

7.3.15 BitPackedFloatAoS/BitPackedFloatSoA

The BitPackedFloatAoS and BitPackedFloatSoA mappings are fully computed mapping that bitpack floating-point values to reduce size and precision. The bits are stored as array of structs and struct of arrays, respectively. The number of bits used to store the exponent and mantissa is configurable. All field types in the record dimension must be floating-point. These mappings require the C++ implementation to use IEEE 754 floating-point formats.

```
unsigned exponentBits = 4;
unsigned mantissaBits = 7;
llama::mapping::BitPackedFloatSoA<ArrayExtents, RecordDim>
        mapping{exponentBits, mantissaBits, extents}; // use 1+4+7 bits for each␣
↪floating-point in RecordDim
```

7.3.16 PermuteArrayIndex

The PermuteArrayIndex mapping is a meta mapping that wraps over an inner mapping. It permutes the array indices before passing the index information to the inner mapping.

```
using InnerMapping = ...;

llama::mapping::PermuteArrayIndex<InnerMapping, 2, 0, 1> mapping{extents};
auto view = llama::allocView(mapping);
view(1, 2, 3); // will pass {3, 1, 2} to inner mapping
```

7.3.17 Dump visualizations

Sometimes it is hard to image how data will be laid out in memory by a mapping. LLAMA can create a graphical representation of a mapping instance as SVG image or HTML document:

```
std::ofstream{filename + ".svg" } << llama::toSvg (mapping);
std::ofstream{filename + ".html"} << llama::toHtml(mapping);
```

CHAPTER

EIGHT

PROXY REFERENCES

8.1 The story of `std::vector<bool>`

When we want to refer to an object of type `T` somewhere in memory, we can form a reference to that object using the language built-in reference `T&`. This also holds true for containers, which often maintain larger portions of memory containing many objects of type `T`. Given an index, we can obtain a reference to one such `T` living in memory:

```
std::vector<T> obj(100);
T& ref = obj[42];
```

The reference `ref` of type `T&` refers to an actual object of type `T` which is truly manifested in memory.

Sometimes however, we choose to store the value of a `T` in a different way in memory, not as an object of type `T`. The most prominent example of such a case is `std::vector<bool>`, which uses bitfields to store the values of the booleans, thus decreasing the memory required for the data structure. However, since `std::vector<bool>` does not store objects of type `bool` in memory, we can now longer form a `bool&` to one of the vectors elements:

```
std::vector<bool> obj(100);
bool& ref = obj[42]; // compile error
```

The proposed solution in this case is to replace the `bool&` by an object representing a reference to a `bool`. Such an object is called a proxy reference. Because some standard containers may use proxy references for some contained types, when we write generic code, it is advisable to use the corresponding `reference` alias provided by them, or to use a forwarding reference:

```
std::vector<T> obj(100);
std::vector<T>::reference ref1 = obj[42]; // works for any T including bool
auto&&                    ref2 = obj[42]; // binds to T& for real references,
                                          // or proxy references returned by value
```

Although `std::vector<bool>` is notorious for this behavior of its references, more such data structures exist (e.g. `std::bitset`) or started to appear in recent C++ standards and its proposals. E.g. in the area of text encodings, or the zip range adaptors.

8.2 Working with proxy references

A proxy reference is usually a value-type with reference semantic. Thus, a proxy reference can be freely created, copied, moved and destroyed. Their sole purpose is to give access to a value they refer to. They usually encapsulate a reference to some storage and computations to be performed when writing or reading through the proxy reference. Write access to a referred value of type `T` is typically given via an assignment operator from `T`. Read access is given by a (non-explicit) conversion operator to `T`.

```
std::vector<bool> v(100);
auto&& ref = v[42];

ref = true;    // write: invokes std::vector<bool>::reference::operator=(bool)
bool b1 = ref; // read:  invokes std::vector<bool>::reference::operator bool()

auto  ref2 = ref;  // takes a copy of the proxy reference (!!!)
auto& ref3 = ref2; // references (via the language build-in l-value reference) the proxy␣
↪reference ref2

for (auto&& ref : v) {
   bool b = ref;
   ref = !b;
   ...
}
```

Mind, that we explicitly state `bool` as the type of the resulting value on access. If we use `auto` instead, we would take a copy of the reference object, not the value.

8.3 Proxy references in LLAMA

By handing out references to contained objects on access, LLAMA views are similar to standard C++ containers. For references to whole records, LLAMA views hand out record references. Although a record reference models a reference to a "struct" (= record) in memory, this struct is not physically manifested in memory. This allows mappings the freedom to arbitrarily arrange how the data for a struct is stored. A record reference in LLAMA is thus a proxy reference to a "struct".

```
auto view = llama::allocView(mapping);
auto rr1 = view(1, 2, 3); // rr1 is a RecordRef, a proxy reference (assuming this access␣
↪is not terminal)
auto rr2 = rr1(color{});  // same here
```

An exception to this are the `load()` and `store()` member functions of a record reference. We might change this in the future.

```
Pixel p = rr.load(); // read access
rr.store(p);         // write access
```

Similarly, some mappings choose a different in-memory representation for the field types in the leaves of the record dimension. Examples are the `Bytesplit`, `ChangeType`, `BitPackedIntSoa` or `BitPackedFloatSoa` mappings. These mappings even return a proxy reference for terminal accesses:

```
auto&& ref = rr(color{}, r{}); // may be a float& or a proxy reference object, depending␣
↪on the mapping
```

Thus, when you want to write truly generic code with LLAMA's views, please keep these guidelines in mind:

- Each non-terminal access on a view returns a record reference, which is a value-type with reference semantic.
- Each terminal access on a view may return an l-value reference or a proxy reference. Thus use `auto&&` to handle both cases.
- Explicitly specify the type of copies of individual fields you want to make from references obtains from a LLAMA view. This avoids accidentally coping a proxy reference.

8.4 Concept

Proxy references in LLAMA fulfill the following concept:

```
template <typename R>
concept ProxyReference = std::is_copy_constructible_v<R> && std::is_copy_assignable_v<R>
    && requires(R r) {
    typename R::value_type;
    { static_cast<typename R::value_type>(r) } -> std::same_as<typename R::value_type>;
    { r = typename R::value_type{} } -> std::same_as<R&>;
} && AdlTwoStepSwappable<R>;
```

That is, a proxy reference can be copied, which should make the original and the copy refer to the same element. It can be assigned to another proxy reference, which should transfer the referred value, not where the proxy reference is referring to! A proxy references provides a member type `value_type`, which indicates the type of the values which can be loaded and stored through the proxy reference. Furthermore, a proxy reference can be converted to its value type (thus calling `operator value_type ()`) or assigned an instance of its value type. Finally, two proxy references can be swapped using the ADL two-step idiom, swapping their referred values:

```
using std::swap;
swap(pr1, pr2);
```

8.5 Arithmetic on proxy references and ProxyRefOpMixin

An additional feature of normal references in C++ is that they can be used as operands for certain operators:

```
auto&& ref = ...;
T = ref + T(42); // works for normal and proxy references
ref++;           // normally, works only for normal references
ref *= 2;        // -||-
                 // both work in LLAMA due to llama::ProxyRefOpMixin
```

Proxy references cannot be used in compound assignment and increment/decrement operators unless they provide overloads for these operators. To cover this case, LLAMA provides the CRTP mixin `llama::ProxyRefOpMixin`, which a proxy reference type can inherit from, to supply the necessary operators. All proxy reference types in LLAMA inherit from `llama::ProxyRefOpMixin` to supply the necessary operators. If you define your own computed mappings returning proxy references, make sure to inherit your proxy reference types from `llama::ProxyRefOpMixin`.

8.6 Member functions and proxy references

Given a class with a member function:

```
struct Rng {
    double next();
    RngState state() const;

private:
    RngState m_state;
};
```

We can naturally call a member function of that class on a reference to an instance in memory in C++:

```
std::vector<Rng> v = ...;
Rng& rng = v[i]; // reference to Rng instance
RngState s = rng.state();
double n = rng.next();
```

However, this is not possible with proxy references:

```
using RecordDim = Rng;
auto v = llama::allocView(m); // where the mapping m uses proxy references
auto&& rng = v[i];            // proxy reference to Rng instance
RngState s = rng.state();     // compilation error
double n = rng.next();        // no member function state()/next() in proxy reference␣
↪class
```

We can workaround this limitation for `const` member functions by materializing the proxy reference into a temporary value:

```
auto&& rng = v[i]; // proxy reference to Rng instance
RngState s = (static_cast<Rng>(rng)).state();
double n = (static_cast<Rng>(rng)).next(); // silent error: updates temporary, not␣
↪instance at rng!
```

This invokes the conversion operator of the proxy reference and we call the member function on a temporary. However, for mutating member functions, the best possible solution so far is to load the instance into a local copy, call the mutating member function, and store back the local copy.

```
auto&& rng = v[i];     // proxy reference to Rng instance
Rng rngCopy = rng;     // local copy
double n = rng.next(); // modify local copy
rng = rngCopy;         // store back modified instance
```

This is also how `llama::ProxyRefOpMixin` is implemented.

In order to allow `rng` to forward the call `.next()` to a different object than itself, C++ would require a frequently discussed, but not standardized, extension: smart references.

8.7 Implementing proxy references

A good explanation on how to implement proxy references is given here. In addition to that, proxy references used with LLAMA should inherit from `llama::ProxyRefOpMixin` and satisfy the concept `llama::ProxyReference`.

CHAPTER

NINE

BLOBS

When a *view* is created, it needs to be given an array of blobs. A blob is an object representing a contiguous region of memory where each byte is accessible using the subscript operator. The number of blobs and the alignment/size of each blob is a property determined by the mapping used by the view. All this is handled by `llama::allocView()`, but I needs to be given a blob allocator to handle the actual allocation of each blob.

```
auto blobAllocator = ...;
auto view = llama::allocView(mapping, blobAllocator);
```

Every time a view is copied, it's array of blobs is copied too. Depending on the type of blobs used, this can have different effects. If e.g. `std::vector<std::byte>` is used, the full storage will be copied. Contrary, if a `std::shared_ptr<std::byte[]>` is used, the storage is shared between each copy of the view.

9.1 Blob allocators

A blob allocator is a callable which returns an appropriately sized blob given a desired compile-time alignment and runtime allocation size in bytes. Choosing the right compile-time alignment has implications on the read/write speed on some CPU architectures and may even lead to CPU exceptions if data is not properly aligned. A blob allocator is called like this:

```
auto blobAllocator = ...;
auto blob = blobAllocator(std::integral_constant<std::size_t, FieldAlignment>{}, size);
```

There is a number of a built-in blob allocators:

9.1.1 Vector

`llama::bloballoc::Vector` is a blob allocator creating blobs of type `std::vector<std::byte>`. This means every time a view is copied, the whole memory is copied too. When the view is moved, no extra allocation or copy operation happens.

9.1.2 Shared pointer

`llama::bloballoc::SharedPtr` is a blob allocator creating blobs of type `std::shared_ptr<std::byte[]>`. These blobs will be shared between each copy of the view and only destroyed then the last view is destroyed.

9.1.3 Unique pointer

`llama::bloballoc::UniquePtr` is a blob allocator creating blobs of type `std::unique_ptr<std::byte[], ...>`. These blobs will be uniquely owned by a single view, so the view cannot be copied, only moved.

9.1.4 Array

When working with small amounts of memory or temporary views created frequently, it is usually beneficial to store the data directly inside the view, avoiding a heap allocation.

`llama::bloballoc::Array` addresses this issue and creates blobs of type `llama::Array<std::byte, N>`, where `N` is a compile time value passed to the allocator. These blobs are copied every time their view is copied. `llama::One` uses this facility. In many such cases, the extents of the array dimensions are also known at compile time, so they can be specified in the template argument list of `llama::ArrayExtents`.

Creating a small view of 4×4 may look like this:

```
using ArrayExtents = llama::ArrayExtents<int, 4, 4>;
constexpr ArrayExtents extents{};

using Mapping = /* a simple mapping */;
auto blobAllocator = llama::bloballoc::Array<
    extents[0] * extents[1] * llama::sizeOf<RecordDim>::value
>;
auto miniView = llama::allocView(Mapping{extents}, blobAllocator);

// or in case the mapping is constexpr and produces just 1 blob:
constexpr auto mapping = Mapping{extents};
auto miniView = llama::allocView(mapping, llama::bloballoc::Array<mapping.blobSize(0)>{}
↪);
```

For N-dimensional one-record views a shortcut exists, returning a view with just one record on the stack:

```
auto tempView = llama::allocScalarView<N, RecordDim>();
```

9.1.5 CudaMalloc

`llama::bloballoc::CudaMalloc` is a blob allocator for creating blobs of type `std::unique_ptr<std::byte[], ...>`. The memory is allocated using `cudaMalloc` and the unique ptr destroys it using `cudaFree`. This allocator is automatically available if the `<cuda_runtime.h>` header is available.

9.1.6 AlpakaBuf

`llama::bloballoc::AlpakaBuf` is a blob allocator for creating alpaka buffers as blobs. This allocator is automatically available if the <alpaka/alpaka.hpp> header is available.

```
auto view = llama::allocView(mapping, llama::bloballoc::AlpakaBuf{alpakaDev});
```

Using this blob allocator is essentially the same as:

```
auto view = llama::allocView(mapping, [&alpakaDev](auto align, std::size_t size){
    return alpaka::allocBuf<std::byte, std::size_t>(alpakaDev, size);
});
```

You may want to use the latter version in case the buffer creation is more complex.

9.2 Non-owning blobs

If a view is needed based on already allocated memory, the view can also be directly constructed with an array of blobs, e.g. an array of `std::byte*` pointers or `std::span<std::byte>` to the existing memory regions. Everything works here as long as it can be subscripted by the view like `blob[offset]`. One needs to be careful though, since now the ownership of the blob is decoupled from the view. It is the responsibility of the user now to ensure that the blobs outlive the views based on them.

9.2.1 Alpaka

LLAMA features some examples using alpaka for the abstraction of computation parallelization. Alpaka has its own memory allocation functions for different memory regions (e.g. host, device and shared memory). Additionally there are some cuda-inherited rules which make e.g. sharing memory regions hard (e.g. no possibility to use a `std::shared_ptr` on a GPU).

Alpaka creates and manages memory using buffers. A pointer to the underlying storage of a buffer can be obtained, which may be used for a LLAMA view:

```
auto buffer = alpaka::allocBuf<std::byte, std::size_t>(dev, size);
auto view = llama::View<Mapping, std::byte*>{mapping, {alpaka::getPtrNative(buffer)}};
```

This is an alternative to the `llama::bloballoc::AlpakaBuf` blob allocator, if the user wants to decouple buffer allocation and view creation.

Shared memory is created by alpaka using a special function returning a reference to a shared variable. To allocate storage for LLAMA, we can allocate a shared byte array using alpaka and then pass the address of the first element to a LLAMA view.

```
auto& sharedMem = alpaka::declareSharedVar<std::byte[sharedMemSize], __COUNTER__>(acc);
auto view = llama::View<Mapping, std::byte*>{mapping, {&sharedMem[0]}};
```

9.3 Shallow copy

The type of a view's blobs determine part of the semantic of the view. It is sometimes useful to strip this type information from a view and create a new view reusing the same memory as the old one, but using a plain referrential blob type (e.g. a `std::byte*`). This is what `llama::shallowCopy` is for.

This is especially useful when passing views with more complicated blob types to accelerators. E.g. views using the `llama::bloballoc::CudaMalloc` allocator:

E.g. views using alpaka buffers as blobs:

CHAPTER

TEN

COPYING BETWEEN VIEWS

Especially when working with hardware accelerators such as GPUs, or offloading to many-core processors, explicit copy operations call for large, contiguous memory chunks to reach good throughput.

Copying the contents of a view from one memory region to another if mapping and size are identical is trivial. However, if the mapping differs, a direct copy of the underlying memory is wrong. In many cases only field-wise copy operations are possible.

There is a small class of remaining cases where the mapping is the same, but the size or shape of the view is different, or the record dimension differ slightly, or the mappings are very related to each other. E.g. when both mappings use SoA, but one time with, one time without padding, or a specific field is missing on one side. Or two AoSoA mappings with a different inner array length. In those cases an optimized copy procedure is possible, copying larger chunks than mere fields.

Four solutions exist for this problem:

1. Implement specializations for specific combinations of mappings, which reflect the properties of these. However, for every new mapping a new specialization is needed.

2. A run time analysis of the two views to find contiguous memory chunks. The overhead is probably big, especially if no contiguous memory chunks are identified.

3. A black box compile time analysis of the mapping function. All current LLAMA mappings are `constexpr` and can thus be run at compile time. This would allow to observe a mappings behavior from exhaustive sampling of the mapping function at compile time.

4. A white box compile time analysis of the mapping function. This requires the mapping to be formulated transparently in a way which is fully consumable via meta-programming, probably at the cost of read- and maintainability. Potentially upcoming C++ features in the area of statement reflection could improve these a lot.

Copies between different address spaces, where elementary copy operations require calls to external APIs, pose an additional challenge and profit especially from large chunk sizes. A good approach could use smaller intermediate views to shuffle a chunk from one mapping to the other and then perform a copy of that chunk into the other address space, potentially overlapping shuffles and copies in an asynchronous workflow.

The async copy example tries to show an asynchronous copy/shuffle/compute workflow. This example applies a blurring kernel to an RGB-image, but also may work only on two or one channel instead of all three. Not used channels are not allocated and especially not copied.

For the moment, LLAMA implements a generic, field-wise copy algorithm with faster specializations for combinations of SoA and AoSoA mappings.

```
auto srcView = llama::allocView(srcMapping);
auto dstView = llama::allocView(dstMapping);
llama::copy(srcView, dstView); // use best copy strategy
```

Internally, `llama::copy` will choose a copy strategy depending on the source and destination mapping. This choice is done via template specializations of the `llama::Copy` class template. Users can add specializations of `llama::Copy` to provide additional copy strategies:

```
// provide special copy from AoS -> UserDefinedMapping
template <typename ArrayExtents, typename RecordDim, bool Aligned, typename␣
↪LinearizeArrayIndex>
struct Copy<
    llama::mapping::AoS<ArrayExtents, RecordDim, Aligned, LinearizeArrayIndex>,
    UserDefinedMapping<ArrayExtents, RecordDim>>
{
    template <typename SrcBlob, typename DstBlob>
    void operator()(
        const View<mapping::AoS<ArrayExtents, RecordDim, Aligned, LinearizeArrayIndex>,␣
↪SrcBlob>& srcView,
        View<mapping::SoA<ArrayExtents, RecordDim, DstSeparateBuffers,␣
↪LinearizeArrayIndex>, DstBlob>& dstView,
        std::size_t threadId, std::size_t threadCount) {
        ...
    }
};

llama::copy(srcView, dstView); // can delegate to above specialization now
```

LLAMA also allows direct access to its two copy implementations, which is mainly used for benchmarking them:

```
llama::fieldWiseCopy(srcView, dstView); // explicit field-wise copy
llama::aosoaCommonBlockCopy(srcView, dstView); // explicit SoA/AoSoA copy
```

CHAPTER

ELEVEN

SIMD

Single instruction, multiple data (SIMD) is a data parallel programming paradigm where an operation is simultaneously performed on multiple data elements.

There is really only one goal to using SIMD and that is: performance. SIMD improves performance by allowing the CPU to crunch more data with each instruction, thus increasing throughput. This influenced some of the API decisions LLAMA has taken, because there is no point in providing an API that cannot be performant. NB: The use of SIMD technology can also improve energy efficiency, but, arguably, this also stems from improved performance.

Many hardware architectures provide dedicated instruction sets (such as AVX2 on x86, or SVE2 on ARM) to perform basic operations such as addition, type-conversion, square-root, etc. on a vector of fundamental types (.e.g `int` or `float`). Such instructions are typically accessibly in C++ via compiler intrinsic functions.

11.1 SIMD libraries

Since compiler intrinsics tend to be hard to use and inflexible (e.g. cannot just switch a code between e.g. AVX2 and AVX512), several SIMD libraries have been developed over time.

Here is a non-exhaustive list of some active SIMD libraries we are aware of:

- EVE (C++20)
- xsimd (C++11)
- std::simd (experimental, GCC >= 11)
- Kokkos SIMD (upcoming in Kokkos 3.7, used to be developed here)

11.2 SIMD interaction with LLAMA

SIMD is primarily a technique for expressing computations. These computations mainly occur between registers but may have optional memory operands. SIMD operations involving memory usually only load or store a vector of N elements from or to the memory location. Thus, whether a code uses SIMD or not is at first glance independent of LLAMA. The only link between SIMD programming and data layouts provided by LLAMA is transferring of N-element vectors between memory and registers instead of scalar values.

Since LLAMA's description and use of record data is rather unwieldy and lead to the creation of `llama::One`, a similar construct for SIMD versions of records, called `llama::Simd`, further increases the usability of the API.

11.3 SIMD library integration with LLAMA

In order for LLAMA to make use of a third-party SIMD library, the class template `llama::SimdTraits` has to be specialized for the SIMD types of the SIMD library. Each specialization `llama::SimdTraits<Simd>` must provide:

- an alias `value_type` to indicate the element type of the Simd.
- a `static constexpr size_t lanes` variable holding the number of SIMD lanes of the Simd.
- a `static auto loadUnaligned(const value_type* mem) -> Simd` function, loading a Simd from the given memory address.
- a `static void storeUnaligned(Simd simd, value_type* mem)` function, storing the given Simd to a given memory address.
- a `static auto gather(const value_type* mem, std::array<int, lanes> indices) -> Simd` function, gathering values into a Simd from the memory addresses identified by `mem + indices * sizeof(value_type)`.
- a `static void scatter(Simd simd, value_type* mem, std::array<int, lanes> indices)` function, scattering the values from a Simd to the memory addresses identified by `mem + indices * sizeof(value_type)`.

For an example integration of *xsimd::batch<T, A>* with LLAMA, see the nbody example. For an example integration of *std::experimental::simd<T, Abi>* with LLAMA, see the *simd.cpp* unit tests.

LLAMA already provides a specialization of `llama::SimdTraits` for the built-in scalar arithmetic types. In that sense, these types are SIMD types from LLAMA's perspective and can be used with the SIMD API in LLAMA.

11.4 LLAMA SIMD API

SIMD codes deal with vectors of N elements. This assumption holds as long as the code uses the same element type for all SIMD vectors. The moment different element types are mixed, all bets are off, and various trade-offs can be made. For this reason, LLAMA does not automatically choose a vector length and this number needs to be provided by the user. A good idea is to query your SIMD library for a suitable size:

```
constexpr auto N = stdx::native_simd<T>::size();
```

Alternatively, LLAMA provides a few constructs to find a SIMD vector length for a given record dimension:

```
constexpr auto N1 = llama::simdLanesWithFullVectorsFor<RecordDim, stdx::native_simd>;
constexpr auto N2 = llama::simdLanesWithLeastRegistersFor<RecordDim, stdx::native_simd>;
```

`llama::simdLanesWithFullVectorsFor` ensures that the vector length is large enough to even fully fill at least one SIMD vector of the smallest field types of the record dimension. So, if your record dimension contains e.g. `double`, `int` and `uint16_t`, then LLAMA will choose a vector length were a `stdx::native_simd<uint16_t>` is full. The SIMD vectors for `double` and `int` would then we larger then a full vector, so the chosen SIMD library needs to support SIMD vector lengths larger than the native length. E.g. the `stdx::fixed_size_simd<T, N>` type allows `N` to be larger than the native vector size.

`llama::simdLanesWithLeastRegistersFor` ensures that the smallest number of SIMD registers is needed and may thus only partially fill registers for some data types. So, given the same record dimension, LLAMA would only fill the SIMD vectors for the largest data type (`double`). The other SIMD vectors would only be partially filled, so the chosen SIMD library needs to support SIMD vector lengths smaller than the native length.

After choosing the SIMD vector length, we can allocate SIMD registers for `N` elements of each record dimension field using `llama::SimdN`:

```
llama::SimdN<RecordDim, N, stdx::fixed_size_simd> s;
```

We expect `llama::SimdN` to be also used in heterogeneous codes where we want to control the vector length at compile time. A common use case would be to have a SIMD length in accord with the available instruction set on a CPU, and a SIMD length of 1 on a GPU. In the latter case, it is important that the code adapts itself to not make use of types from a third-party SIMD library, as these cannot usually be compiled for GPU targets. Therefore, for an `N` of 1, LLAMA will not use SIMD types:

SimdN<T, N>		N > 1 | N == 1	
T	record dim	One<Simd<T>>	One<T>
	scalar	Simd<T>	T

Alternatively, there is also a version without an enforced SIMD vector length:

```
llama::Simd<RecordDim, stdx::native_simd> s;
```

Mind however, that with `llama::Simd`, LLAMA does not enforce a vector width. This choice is up to the behavior of the SIMD type. Thus, the individual SIMD vectors (one per record dimension field) may have different lengths.

`llama::SimdN` and `llama::Simd` both make use of the helpers `llama::SimdizeN` and `llama::Simdize` to create SIMD versions of a given record dimension:

```
using RecordDimSimdN = llama::SimdizeN<RecordDim, N, stdx::fixed_size_simd>;
using RecordDimSimd  = llama::Simdize <RecordDim,    stdx::native_simd>;
```

Eventually, whatever SIMD type is built or used by the user, LLAMA needs to be able to query its lane count in a generic context. This is what `llama::simdLanes` is for.

T	llama::simdLanes<T>
scalar (std::is_arithmetic)	1
llama::One<T>	1
llama::SimdN<T, N, ...>	N
llama::Simd<T, ...>	llama::simdLanes<First<T>> // if equal for all fields // otherwise: compile error
otherwise	llama::SimdTraits<T>::lanes

Use `llama::simdLanes` in generic code which needs to handle scalars, third-party SIMD vectors (via. `llama::SimdTraits`, record references, `llama::One` and LLAMA built SIMD types.

Loading and storing data between a SIMD vector and a llama view is done using `llama::loadSimd` and `llama::storeSimd`:

```
llama::loadSimd(view(i), s);
llama::storeSimd(s, view(i));
```

Both functions take a `llama::Simd` and a reference into a LLAMA view as arguments. Depending on the mapping of the view, different load/store instructions will be used. E.g. `llama::mapping::SoA` will allow SIMD loads/stores, whereas `llama::mapping::AoS` will resort to scalar loads/stores (which the compiler sometimes optimizes into SIMD gather/scatter).

Since `llama::Simd` is a special version of `llama::One`, ordinary navigation to sub records and arithmetic can be performed:

```
llama::SimdN<Vec3, N, stdx::fixed_size_simd> vel; // SIMD with N lanes holding 3 values
llama::loadSimd(view(i)(Vel{}), vel);

s(Pos{}) += vel; // 3 SIMD adds performed between llama::Simd vel and sub-record␣
↪llama::Simd of s
llama::storeSimd(s(Pos{}), view(i)(Pos{})); // store subpart of llama::Simd into view
```

CHAPTER

TWELVE

MACROS

As LLAMA tries to stay independent from specific compiler vendors and extensions, C preprocessor macros are used to define some directives for a subset of compilers but with a unified interface for the user. Some macros can even be overwritten from the outside to enable interoperability with libraries such as alpaka.

12.1 Offloading

We frequently have to deal with dialects of C++ which allow/require do specify to which target a function is compiled. To support his use, every method which can be used on offloading devices (e.g. GPUs) uses the `LLAMA_FN_HOST_ACC_INLINE` macro as attribute. By default it is defined as:

```
#ifndef LLAMA_FN_HOST_ACC_INLINE
    #define LLAMA_FN_HOST_ACC_INLINE inline
#endif
```

When working with cuda it should be globally defined as something like `__host__ __device__ inline`. Please specify this as part of your CXX flags globally. When LLAMA is used in conjunction with alpaka, please define it as `ALPAKA_FN_ACC __forceinline__` (with CUDA) or `ALPAKA_FN_ACC inline`.

12.2 Data (in)dependence

Compilers usually cannot assume that two data regions are independent of each other if the data is not fully visible to the compiler (e.g. a value completely lying on the stack or the compiler observing the allocation call). One solution in C is the `restrict` keyword which specifies that the memory pointed to by a pointer is not aliased by anything else. However this does not work for more complex data structures containing pointers, and easily fails in other scenarios as well.

Another solution are compiler specific `#pragma`s which tell the compiler that **each** memory access through a pointer inside a loop can be assumed to not interfere with other accesses through other pointers. The usual goal is to allow vectorization. Such `#pragma`s are handy and work with more complex data types, too. LLAMA provides a macro called `LLAMA_INDEPENDENT_DATA` which can be put in front of loops to communicate the independence of memory accesses to the compiler.

CHAPTER

THIRTEEN

API

Users should just include `llama.hpp` and all functionality should be available. All basic functionality of the library is in the namespace `llama` or sub namespaces.

13.1 Useful helpers

template<typename **T**>

struct **NrAndOffset**

template<typename **T**>
constexpr auto `llama::`**structName**(*T* = {}) -> std::string_view

template<typename **FromT**, typename **ToT**>

using `llama::`**CopyConst** = std::conditional_t<std::is_const_v<*FromT*>, const *ToT*, *ToT*>

Alias for ToT, adding `const` if FromT is const qualified.

template<typename **Derived**, typename **ValueType**>

struct **ProxyRefOpMixin**

CRTP mixin for proxy reference types to support all compound assignment and increment/decrement operators.

template<typename **T**>
inline auto `llama::`**decayCopy**(*T* &&valueOrRef) -> typename internal::ValueOf<*T*>::type

Pulls a copy of the given value or reference. Proxy references are resolved to their value types.

template<typename **Reference**, typename = void>

struct **ScopedUpdate** : public internal::ValueOf::type

Scope guard type. *ScopedUpdate* takes a copy of a value through a reference and stores it internally during construction. The stored value is written back when *ScopedUpdate* is destroyed. *ScopedUpdate* tries to act like the stored value as much as possible, exposing member functions of the stored value and acting like a proxy reference if the stored value is a primitive type.

13.1.1 Array

template<typename **T**, std::size_t **N**>

struct **Array**

Array class like `std::array` but suitable for use with offloading devices like GPUs.

Template Parameters

- **T** – type if array elements.
- **N** – rank of the array.

template<typename **T**, std::size_t **N**>
inline constexpr auto `llama::`**pushFront**([[maybe_unused]] *Array*<*T*, *N*> a, *T* v) -> *Array*<*T*, *N* + 1>

template<typename **T**, std::size_t **N**>
inline constexpr auto `llama::`**pushBack**([[maybe_unused]] *Array*<*T*, *N*> a, *T* v) -> *Array*<*T*, *N* + 1>

Warning: doxygenfunction: Unable to resolve function "llama::popFront" with arguments (Array<T, N>) in doxygen xml output for project "LLAMA" from directory: ./doxygen/xml. Potential matches:

```
- template<typename ...Elements> constexpr auto popFront(const Tuple<Elements...> &
↪tuple)
- template<typename T, std::size_t N> constexpr auto popFront([[maybe_unused]] Array
↪<T, N> a)
```

template<typename **T**, std::size_t **N**>
inline constexpr auto `llama::`**popBack**([[maybe_unused]] *Array*<*T*, *N*> a)

template<typename **T**, std::size_t **N**>
inline constexpr auto `llama::`**product**(*Array*<*T*, *N*> a) -> *T*

13.1.2 Tuple

template<typename ...**Elements**>

struct **Tuple**

template<std::size_t **I**, typename ...**Elements**>
inline constexpr auto `llama::`**get**(*Tuple*<*Elements*...> &tuple) -> auto&

template<typename **Tuple1**, typename **Tuple2**>
inline constexpr auto `llama::`**tupleCat**(const *Tuple1* &t1, const *Tuple2* &t2)

template<std::size_t **Pos**, typename **Tuple**, typename **Replacement**>
inline constexpr auto `llama::`**tupleReplace**(*Tuple* &&tuple, *Replacement* &&replacement)

Creates a copy of a tuple with the element at position Pos replaced by replacement.

template<typename ...**Elements**, typename **Functor**>
inline constexpr auto `llama::`**tupleTransform**(const *Tuple*<*Elements*...> &tuple, const *Functor* &functor)

Applies a functor to every element of a tuple, creating a new tuple with the result of the element transformations. The functor needs to implement a template `operator()` to which all tuple elements are passed.

template<typename ...**Elements**>

inline constexpr auto `llama::popFront`(const *Tuple*<*Elements*...> &tuple)

Returns a copy of the tuple without the first element.

13.2 Array dimensions

template<typename **T** = std::size_t, *T*... **Sizes**>

struct **ArrayExtents** : public llama::*Array*<std::size_t, ((*Sizes* == dyn) + ... + 0)>

ArrayExtents holding compile and runtime indices. This is conceptually equivalent to the std::extent of std::mdspan (

See also:

: https://wg21.link/P0009) including the changes to make the size_type controllable (

See also:

: https://wg21.link/P2553).

Subclassed by *llama::ArrayIndexRange< ArrayExtents >*

template<typename **SizeType**, std::size_t **N**>

using `llama::ArrayExtentsDynamic` = *ArrayExtentsNCube*<*SizeType*, *N*, dyn>

N-dimensional *ArrayExtents* where all values are dynamic.

template<typename **SizeType**, std::size_t **N**, *SizeType* **Extent**>

using `llama::ArrayExtentsNCube` = decltype(internal::makeArrayExtents<*SizeType*, *Extent*>(std::make_index_sequence<*N*>{}))

N-dimensional *ArrayExtents* where all N extents are Extent.

template<typename **T**, std::size_t **Dim**>

struct **ArrayIndex** : public llama::*Array*<*T*, *Dim*>

Represents a run-time index into the array dimensions.

Template Parameters

Dim – Compile-time number of dimensions.

template<typename **ArrayExtents**>

struct **ArrayIndexIterator**

Iterator supporting *ArrayIndexRange*.

template<typename **ArrayExtents**>

struct **ArrayIndexRange** : private llama::*ArrayExtents*<T, Sizes>

Range allowing to iterate over all indices in an *ArrayExtents*.

template<typename **SizeType**, *SizeType*... **Sizes**, typename **Func**>
inline void `llama::forEachArrayIndex`(*ArrayExtents*<*SizeType*, *Sizes*...> extents, *Func* &&func)

13.3 Record dimension

template<typename ...**Fields**>

struct **Record**

A type list of *Field*s which may be used to define a record dimension.

template<typename **Tag**, typename **Type**>

struct **Field**

Record dimension tree node which may either be a leaf or refer to a child tree presented as another *Record*.

Template Parameters

- **Tag** – Name of the node. May be any type (struct, class).
- **Type** – Type of the node. May be one of three cases. 1. another sub tree consisting of a nested *Record*. 2. an array of static size of any type, in which case a *Record* with as many *Field* as the array size is created, named *RecordCoord* specialized on consecutive numbers I. 3. A scalar type different from *Record*, making this node a leaf of this type.

struct **NoName**

Anonymous naming for a *Field*.

template<typename **Field**>

using `llama::`**GetFieldTag** = mp_first<*Field*>

Get the tag from a *Field*.

template<typename **Field**>

using `llama::`**GetFieldType** = mp_second<*Field*>

Get the type from a *Field*.

template<typename **RecordDim**, typename **RecordCoord**, bool **Align** = false>

constexpr std::size_t `llama::`**offsetOf** = flatOffsetOf<*FlatRecordDim*<*RecordDim*>, *flatRecordCoord*<*RecordDim*, *RecordCoord*>, *Align*>

The byte offset of an element in a record dimension if it would be a normal struct.

Template Parameters

- **RecordDim** – *Record* dimension tree.
- **RecordCoord** – *Record* coordinate of an element inrecord dimension tree.

template<typename **T**, bool **Align** = false, bool **IncludeTailPadding** = true>

constexpr std::size_t `llama::`**sizeOf** = sizeof(*T*)

The size of a type T.

template<typename **T**>

constexpr std::size_t `llama::`**alignOf** = alignof(*T*)

The alignment of a type T.

template<typename **RecordDim**, typename **RecordCoord**>

using `llama::GetTags` = typename internal::GetTagsImpl<*RecordDim*, *RecordCoord*>::type

Get the tags of all *Field*s from the root of the record dimension tree until to the node identified by *RecordCoord*.

template<typename **RecordDim**, typename **RecordCoord**>

using `llama::GetTag` = typename internal::GetTagImpl<*RecordDim*, *RecordCoord*>::type

Get the tag of the *Field* at a *RecordCoord* inside the record dimension tree.

template<typename **RecordDimA**, typename **RecordCoordA**, typename **RecordDimB**, typename **RecordCoordB**>

constexpr auto `llama::hasSameTags`

Is true if, starting at two coordinates in two record dimensions, all subsequent nodes in the record dimension tree have the same tag.

Template Parameters

- **RecordDimA** – First record dimension.
- **RecordCoordA** – *RecordCoord* based on RecordDimA along which the tags are compared.
- **RecordDimB** – second record dimension.
- **RecordCoordB** – *RecordCoord* based on RecordDimB along which the tags are compared.

template<typename **RecordDim**, typename ...**TagsOrTagList**>

using `llama::GetCoordFromTags` = typename internal::GetCoordFromTagsImpl<*RecordDim*, *RecordCoord*<>, *TagsOrTagList*...>::type

Converts a series of tags, or a list of tags, navigating down a record dimension into a *RecordCoord*. A *RecordCoord* will be passed through unmodified.

template<typename **RecordDim**, typename ...**RecordCoordOrTags**>

using `llama::GetType` = typename internal::GetTypeImpl<*RecordDim*, *RecordCoordOrTags*...>::type

Returns the type of a node in a record dimension tree identified by a given *RecordCoord* or a series of tags.

template<typename **RecordDim**>

using `llama::FlatRecordDim` = typename internal::FlattenRecordDimImpl<*RecordDim*>::type

Returns a flat type list containing all leaf field types of the given record dimension.

template<typename **RecordDim**, typename **RecordCoord**>

constexpr std::size_t `llama::flatRecordCoord` = 0

The equivalent zero based index into a flat record dimension (*FlatRecordDim*) of the given hierarchical record coordinate.

template<typename **RecordDim**>

using `llama::LeafRecordCoords` = typename internal::LeafRecordCoordsImpl<*RecordDim*, *RecordCoord*<>>::type

Returns a flat type list containing all record coordinates to all leaves of the given record dimension.

template<typename **RecordDim**, template<typename> typename **FieldTypeFunctor**>

using `llama::TransformLeaves` = TransformLeavesWithCoord<*RecordDim*, internal::MakePassSecond<*FieldTypeFunctor*>::template fn>

Creates a new record dimension where each new leaf field's type is the result of applying FieldTypeFunctor to the original leaf field's type.

template<typename RecordDimA,
typename RecordDimB> llama::MergedRecordDims = typename decltype(internal::mergeRecordDimsImpl(mp_ident
mp_identity< RecordDimB >{}))::type

Creates a merged record dimension, where duplicated, nested fields are unified.

template<typename **RecordDim**, typename **Functor**, typename ...**Tags**>
inline constexpr void `llama::forEachLeafCoord`(*Functor* &&functor, *Tags*...)

Iterates over the record dimension tree and calls a functor on each element.

Parameters

- **functor** – Functor to execute at each element of. Needs to have `operator()` with a template parameter for the *RecordCoord* in the record dimension tree.
- **baseTags** – Tags used to define where the iteration should be started. The functor is called on elements beneath this coordinate.

template<typename **RecordDim**, typename **Functor**, std::size_t... **Coords**>
inline constexpr void `llama::forEachLeafCoord`(*Functor* &&functor, *RecordCoord*<*Coords*...> baseCoord)

Iterates over the record dimension tree and calls a functor on each element.

Parameters

- **functor** – Functor to execute at each element of. Needs to have `operator()` with a template parameter for the *RecordCoord* in the record dimension tree.
- **baseCoord** – *RecordCoord* at which the iteration should be started. The functor is called on elements beneath this coordinate.

template<typename **RecordDim**, std::size_t... **Coords**>
constexpr auto `llama::prettyRecordCoord`(*RecordCoord*<*Coords*...> = {}) -> std::string_view

Returns a pretty representation of the record coordinate inside the given record dimension. Tags are interspersed by '.' and arrays are represented using subscript notation ("[123]").

13.4 Record coordinates

template<std::size_t... **Coords**>

struct **RecordCoord**

Represents a coordinate for a record inside the record dimension tree.

Template Parameters
Coords... – the compile time coordinate.

Public Types

using **List** = mp_list_c<std::size_t, *Coords*...>

The list of integral coordinates as `mp_list`.

template<typename **L**>

using `llama::RecordCoordFromList` = internal::mp_unwrap_values_into<*L*, *RecordCoord*>

Converts a type list of integral constants into a *RecordCoord*.

template<typename ...**RecordCoords**>

using `llama::Cat` = *RecordCoordFromList*<mp_append<typename *RecordCoords*::List...>>

Concatenate a set of *RecordCoord*s.

template<typename **RecordCoord**>

using `llama::PopFront` = *RecordCoordFromList*<mp_pop_front<typename *RecordCoord*::List>>

RecordCoord without first coordinate component.

template<typename **First**, typename **Second**>

constexpr auto `llama::recordCoordCommonPrefixIsBigger` =
internal::recordCoordCommonPrefixIsBiggerImpl(*First*{}, *Second*{})

Checks wether the first *RecordCoord* is bigger than the second.

template<typename **First**, typename **Second**>

constexpr auto `llama::recordCoordCommonPrefixIsSame` =
internal::recordCoordCommonPrefixIsSameImpl(*First*{}, *Second*{})

Checks whether two *RecordCoord*s are the same or one is the prefix of the other.

13.5 Views

template<typename **Mapping**, typename **Allocator** = bloballoc::*Vector*, typename **Accessor** = *accessor*::Default>
inline auto `llama::allocView`(*Mapping* mapping = {}, const *Allocator* &alloc = {}, *Accessor* accessor = {}) ->
View<*Mapping*, internal::AllocatorBlobType<*Allocator*, typename *Mapping*::RecordDim>, *Accessor*>

Creates a view based on the given mapping, e.g. *mapping::AoS* or *mapping::SoA*. For allocating the view's underlying memory, the specified allocator callable is used (or the default one, which is *bloballoc::Vector*). The allocator callable is called with the alignment and size of bytes to allocate for each blob of the mapping. Value-initialization is performed for all fields by calling *constructFields*. This function is the preferred way to create a *View*. See also *allocViewUninitialized*.

template<typename **Mapping**, typename **BlobType**, typename **Accessor**>
inline void `llama::constructFields`(*View*<*Mapping*, *BlobType*, *Accessor*> &view)

Value-initializes all fields reachable through the given view. That is, constructors are run and fundamental types are zero-initialized. Computed fields are constructed if they return l-value references and assigned a default constructed value if they return a proxy reference.

template<typename **Mapping**, typename **Allocator** = bloballoc::*Vector*, typename **Accessor** = *accessor*::Default>
inline auto `llama::allocViewUninitialized`(*Mapping* mapping = {}, const *Allocator* &alloc = {}, *Accessor* accessor = {})

Same as *allocView* but does not run field constructors.

template<std::size_t **Dim**, typename **RecordDim**>
inline auto `llama::allocScalarView`() -> decltype(auto)

Allocates a *View* holding a single record backed by a byte array (*bloballoc::Array*).

Template Parameters
Dim – Dimension of the *ArrayExtents* of the *View*.

template<typename **RecordDim**>

using `llama::One` = *RecordRef* <decltype(*allocScalarView*<0, *RecordDim*>()), *RecordCoord*<>, true>

A *RecordRef* that owns and holds a single value.

template<typename **View**, typename **BoundRecordCoord**, bool **OwnView**>
inline auto `llama::copyRecord`(const *RecordRef*<*View*, *BoundRecordCoord*, *OwnView*> &rr)

Returns a *One* with the same record dimension as the given record ref, with values copyied from rr.

template<typename **ViewFwd**, typename **TransformBlobFunc**, typename = std::enable_if_t<isView<std::decay_t<*ViewFwd*>>>>
inline auto `llama::transformBlobs`(*ViewFwd* &&view, const *TransformBlobFunc* &transformBlob)

Applies the given transformation to the blobs of a view and creates a new view with the transformed blobs and the same mapping and accessor as the old view.

template<typename **View**, typename **NewBlobType** = *CopyConst*<std::remove_reference_t<*View*>, std::byte>*, typename = std::enable_if_t<isView<std::decay_t<*View*>>>>
inline auto `llama::shallowCopy`(*View* &&view)

Creates a shallow copy of a view. This copy must not outlive the view, since it references its blob array.

Template Parameters
NewBlobType – The blob type of the shallow copy. Must be a non owning pointer like type.

Returns
A new view with the same mapping as view, where each blob refers to the blob in view.

template<typename **NewMapping**, typename **ViewFwd**, typename = std::enable_if_t<isView<std::decay_t<*ViewFwd*>>>>
inline auto `llama::withMapping`(*ViewFwd* &&view, *NewMapping* newMapping = {})

template<typename **NewAccessor**, typename **ViewFwd**, typename = std::enable_if_t<isView<std::decay_t<*ViewFwd*>>>>
inline auto `llama::withAccessor`(*ViewFwd* &&view, *NewAccessor* newAccessor = {})

13.5.1 Blob allocators

struct **Vector**

Allocates heap memory managed by a `std::vector` for a *View*, which is copied each time a *View* is copied.

struct **SharedPtr**

Allocates heap memory managed by a `std::shared_ptr` for a *View*. This memory is shared between all copies of a *View*.

struct **UniquePtr**

Allocates heap memory managed by a `std::unique_ptr` for a *View*. This memory can only be uniquely owned by a single *View*.

template<std::size_t **BytesToReserve**>

struct **Array**

Allocates statically sized memory for a *View*, which is copied each time a *View* is copied.

Template Parameters
BytesToReserve – the amount of memory to reserve.

template<std::size_t **Alignment**>

struct **AlignedArray** : public llama::*Array*<std::byte, *BytesToReserve*>

13.6 Mappings

template<typename **TArrayExtents**, typename **TRecordDim**, FieldAlignment **TFieldAlignment** = FieldAlignment::Align, typename **TLinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*, template<typename> typename **PermuteFields** = *PermuteFieldsInOrder*>
struct **AoS** : public llama::mapping::MappingBase<*TArrayExtents*, *TRecordDim*>

Array of struct mapping. Used to create a *View* via *allocView*.

Template Parameters

- **TFieldAlignment** – If Align, padding bytes are inserted to guarantee that struct members are properly aligned. If Pack, struct members are tightly packed.
- **TLinearizeArrayIndexFunctor** – Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.
- **PermuteFields** – Defines how the record dimension's fields should be permuted. See *PermuteFieldsInOrder*, *PermuteFieldsIncreasingAlignment*, *PermuteFieldsDecreasingAlignment* and *PermuteFieldsMinimizePadding*.

template<typename **ArrayExtents**, typename **RecordDim**, typename **LinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*>
using `llama::mapping::`**AlignedAoS** = *AoS*<*ArrayExtents*, *RecordDim*, FieldAlignment::Align, *LinearizeArrayIndexFunctor*>

Array of struct mapping preserving the alignment of the field types by inserting padding.

See also:

AoS

template<typename **ArrayExtents**, typename **RecordDim**, typename **LinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*>
using `llama::mapping::`**MinAlignedAoS** = *AoS*<*ArrayExtents*, *RecordDim*, FieldAlignment::Align, *LinearizeArrayIndexFunctor*, *PermuteFieldsMinimizePadding*>

Array of struct mapping preserving the alignment of the field types by inserting padding and permuting the field order to minimize this padding.

See also:

AoS

template<typename **ArrayExtents**, typename **RecordDim**, typename **LinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*>
using `llama::mapping::`**PackedAoS** = *AoS*<*ArrayExtents*, *RecordDim*, FieldAlignment::Pack, *LinearizeArrayIndexFunctor*>

Array of struct mapping packing the field types tightly, violating the type's alignment requirements.

See also:

AoS

template<typename **ArrayExtents**, typename **RecordDim**, typename **LinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*>
using `llama::mapping::`**AlignedSingleBlobSoA** = *SoA*<*ArrayExtents*, *RecordDim*, Blobs::Single, SubArrayAlignment::Align, *LinearizeArrayIndexFunctor*>

Struct of array mapping storing the entire layout in a single blob. The starts of the sub arrays are aligned by inserting padding.

See also:

SoA

template<typename **ArrayExtents**, typename **RecordDim**, typename **LinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*>
using `llama::mapping::PackedSingleBlobSoA` = *SoA*<*ArrayExtents*, *RecordDim*, Blobs::Single, SubArrayAlignment::Pack, *LinearizeArrayIndexFunctor*>

Struct of array mapping storing the entire layout in a single blob. The sub arrays are tightly packed, violating the type's alignment requirements.

See also:

SoA

template<typename **ArrayExtents**, typename **RecordDim**, typename **LinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*>
using `llama::mapping::MultiBlobSoA` = *SoA*<*ArrayExtents*, *RecordDim*, Blobs::OnePerField, SubArrayAlignment::Pack, *LinearizeArrayIndexFunctor*>

Struct of array mapping storing each attribute of the record dimension in a separate blob.

See also:

SoA

template<typename **TArrayExtents**, typename **TRecordDim**, typename *TArrayExtents*::value_type **Lanes**, FieldAlignment **TFieldAlignment** = FieldAlignment::Align, typename **TLinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*, template<typename> typename **PermuteFields** = *PermuteFieldsInOrder*>
struct **AoSoA** : public llama::mapping::MappingBase<*TArrayExtents*, *TRecordDim*>

Array of struct of arrays mapping. Used to create a *View* via *allocView*.

Template Parameters

- **Lanes** – The size of the inner arrays of this array of struct of arrays.
- **TFieldAlignment** – If Align, padding bytes are inserted to guarantee that struct members are properly aligned. If Pack, struct members are tightly packed.
- **PermuteFields** – Defines how the record dimension's fields should be permuted. See *PermuteFieldsInOrder*, *PermuteFieldsIncreasingAlignment*, *PermuteFieldsDecreasingAlignment* and *PermuteFieldsMinimizePadding*.

template<typename **RecordDim**, std::size_t **VectorRegisterBits**>

constexpr std::size_t `llama::mapping::maxLanes`

The maximum number of vector lanes that can be used to fetch each leaf type in the record dimension into a vector register of the given size in bits.

template<typename **TArrayExtents**, typename **TRecordDim**, typename **Bits** = typename *TArrayExtents*::value_type, *SignBit* **SignBit** = *SignBit*::Keep, typename **TLinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*, template<typename> typename **PermuteFields** = *PermuteFieldsInOrder*, typename **TStoredIntegral** = internal::StoredUnsignedFor<*TRecordDim*>>
struct **BitPackedIntAoS** : public llama::mapping::internal::BitPackedIntCommon<*TArrayExtents*, *TRecordDim*, typename *TArrayExtents*::value_type, *SignBit*::Keep, *LinearizeArrayIndexRight*, internal::StoredUnsignedFor<*TRecordDim*>>

Array of struct mapping using bit packing to reduce size/precision of integral data types. If your record dimension contains non-integral types, split them off using the *Split* mapping first.

Template Parameters

- **Bits** – If Bits is llama::Constant<N>, the compile-time N specifies the number of bits to use. If Bits is an integral type T, the number of bits is specified at runtime, passed to the constructor and stored as type T. Must not be zero and must not be bigger than the bits of TStoredIntegral.
- **SignBit** – When set to SignBit::Discard, discards the sign bit when storing signed integers. All numbers will be read back positive.
- **TLinearizeArrayIndexFunctor** – Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.
- **PermuteFields** – Defines how the record dimension's fields should be permuted. See \tparam TStoredIntegral Integral type used as storage of reduced precision integers. Must be std::uint32_t or std::uint64_t.

template<typename **TArrayExtents**, typename **TRecordDim**, typename **Bits** = typename *TArrayExtents*::value_type, *SignBit* **SignBit** = *SignBit*::Keep, typename **TLinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*, typename **TStoredIntegral** = internal::StoredUnsignedFor<*TRecordDim*>>
struct **BitPackedIntSoA** : public llama::mapping::internal::BitPackedIntCommon<*TArrayExtents*, *TRecordDim*, typename *TArrayExtents*::value_type, *SignBit*::Keep, *LinearizeArrayIndexRight*, internal::StoredUnsignedFor<*TRecordDim*>>

Struct of array mapping using bit packing to reduce size/precision of integral data types. If your record dimension contains non-integral types, split them off using the *Split* mapping first.

Template Parameters

- **Bits** – If Bits is llama::Constant<N>, the compile-time N specifies the number of bits to use. If Bits is an integral type T, the number of bits is specified at runtime, passed to the constructor and stored as type T. Must not be zero and must not be bigger than the bits of TStoredIntegral.
- **SignBit** – When set to SignBit::Discard, discards the sign bit when storing signed integers. All numbers will be read back positive.
- **TLinearizeArrayIndexFunctor** – Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.
- **TStoredIntegral** – Integral type used as storage of reduced precision integers. Must be std::uint32_t or std::uint64_t.

template<typename **TArrayExtents**, typename **TRecordDim**, typename **ExponentBits** = typename *TArrayExtents*::value_type, typename **MantissaBits** = *ExponentBits*, typename **TLinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*, template<typename> typename **PermuteFields** = *PermuteFieldsInOrder*, typename **TStoredIntegral** = internal::StoredIntegralFor<*TRecordDim*>>
struct **BitPackedFloatAoS** : public llama::mapping::MappingBase<*TArrayExtents*, *TRecordDim*>, public llama::internal::BoxedValue<typename *TArrayExtents*::value_type, 0>, public llama::internal::BoxedValue<typename *TArrayExtents*::value_type, 1>

template<typename **TArrayExtents**, typename **TRecordDim**, typename **ExponentBits** = typename *TArrayExtents*::value_type, typename **MantissaBits** = *ExponentBits*, typename **TLinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*, typename **TStoredIntegral** = internal::StoredIntegralFor<*TRecordDim*>>
struct **BitPackedFloatSoA** : public llama::mapping::MappingBase<*TArrayExtents*, *TRecordDim*>, public llama::internal::BoxedValue<typename *TArrayExtents*::value_type, 0>, public llama::internal::BoxedValue<typename *TArrayExtents*::value_type, 1>

Struct of array mapping using bit packing to reduce size/precision of floating-point data types. The bit layout is [1 sign bit, exponentBits bits from the exponent, mantissaBits bits from the mantissa]+ and tries to follow IEEE 754. Infinity and NAN are supported. If the packed exponent bits are not big enough to hold a number, it will be set to infinity (preserving the sign). If your record dimension contains non-floating-point types, split them off using the *Split* mapping first.

Template Parameters

- **ExponentBits** – If ExponentBits is llama::Constant<N>, the compile-time N specifies the number of bits to use to store the exponent. If ExponentBits is llama::Value<T>, the number of bits is specified at runtime, passed to the constructor and stored as type T. Must not be zero.
- **MantissaBits** – Like ExponentBits but for the mantissa bits. Must not be zero (otherwise values turn INF).
- **TLinearizeArrayIndexFunctor** – Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.
- **TStoredIntegral** – Integral type used as storage of reduced precision floating-point values.

template<typename **TArrayExtents**, typename **TRecordDim**, template<typename, typename> typename **InnerMapping**>
struct **Bytesplit** : private *InnerMapping*<*TArrayExtents*, internal::SplitBytes<*TRecordDim*>>

Meta mapping splitting each field in the record dimension into an array of bytes and mapping the resulting record dimension using a further mapping.

template<typename **RC**, typename **BlobArray**>

struct **Reference** : public llama::*ProxyRefOpMixin*<*Reference*<*RC*, *BlobArray*>, *GetType*<*TRecordDim*, *RC*>>

template<typename **ArrayExtents**, typename **RecordDim**, template<typename, typename> typename **InnerMapping**>
struct **Byteswap** : public llama::mapping::*Projection*<*ArrayExtents*, *RecordDim*, *InnerMapping*, internal::MakeByteswapProjectionMap<*RecordDim*>>

Mapping that swaps the byte order of all values when loading/storing.

template<typename **ArrayExtents**, typename **RecordDim**, template<typename, typename> typename **InnerMapping**, typename **ReplacementMap**>
struct **ChangeType** : public llama::mapping::*Projection*<*ArrayExtents*, *RecordDim*, *InnerMapping*, internal::MakeProjectionMap<*RecordDim*, *ReplacementMap*>>

Mapping that changes the type in the record domain for a different one in storage. Conversions happen during load and store.

Template Parameters

ReplacementMap – A type list of binary type lists (a map) specifiying which type or the type at a *RecordCoord* (map key) to replace by which other type (mapped value).

template<typename **Mapping**, typename *Mapping*::ArrayExtents::value_type **Granularity** = 1, typename **TCountType** = std::size_t>
struct **Heatmap** : private *Mapping*

Forwards all calls to the inner mapping. Counts all accesses made to blocks inside the blobs, allowing to extract a heatmap.

Template Parameters

- **Mapping** – The type of the inner mapping.
- **Granularity** – The granularity in bytes on which to could accesses. A value of 1 counts every byte. individually. A value of e.g. 64, counts accesses per 64 byte block.
- **TCountType** – Data type used to count the number of accesses. Atomic increments must be supported for this type.

Public Functions

template<typename **Blobs**, typename **OStream**>
inline void **writeGnuplotDataFileAscii**(const *Blobs* &blobs, *OStream* &&os, bool trimEnd = true, std::size_t wrapAfterBlocks = 64) const

Writes a data file suitable for gnuplot containing the heatmap data. You can use the script provided by gnuplotScript to plot this data file.

Parameters

- **blobs** – The blobs of the view containing this mapping
- **os** – The stream to write the data to. Should be some form of std::ostream.

Public Static Attributes

static constexpr std::string_view **gnuplotScriptAscii**

An example script for plotting the ASCII heatmap data using gnuplot.

static constexpr std::string_view **gnuplotScriptBinary**

An example script for plotting the binary heatmap data using gnuplot.

template<typename **TArrayExtents**, typename **TRecordDim**>

struct **Null** : public llama::mapping::MappingBase<*TArrayExtents*, *TRecordDim*>

The *Null* mappings maps all elements to nothing. Writing data through a reference obtained from the *Null* mapping discards the value. Reading through such a reference returns a default constructed object.

template<typename **TArrayExtents**, typename **TRecordDim**, FieldAlignment **TFieldAlignment** = FieldAlignment::Align, template<typename> typename **PermuteFields** = *PermuteFieldsMinimizePadding*>
struct **One** : public llama::mapping::MappingBase<*TArrayExtents*, *TRecordDim*>

Maps all array dimension indices to the same location and layouts struct members consecutively. This mapping is used for temporary, single element views.

Template Parameters

- **TFieldAlignment** – If Align, padding bytes are inserted to guarantee that struct members are properly aligned. If false, struct members are tightly packed.
- **PermuteFields** – Defines how the record dimension's fields should be permuted. See *PermuteFieldsInOrder*, *PermuteFieldsIncreasingAlignment*, *PermuteFieldsDecreasingAlignment* and *PermuteFieldsMinimizePadding*.

template<typename **TArrayExtents**, typename **TRecordDim**, template<typename, typename> typename **InnerMapping**, typename **TProjectionMap**>
struct **Projection** : private *InnerMapping*<*TArrayExtents*, internal::ReplaceTypesByProjectionResults<*TRecordDim*, *TProjectionMap*>>

Mapping that projects types in the record domain to different types. Projections are executed during load and store.

Template Parameters

TProjectionMap – A type list of binary type lists (a map) specifing a projection (map value) for a type or the type at a *RecordCoord* (map key). A projection is a type with two functions: struct Proj { static auto load(auto&& fromMem); static auto store(auto&& toMem); };

template<typename **TArrayExtents**, typename **TRecordDim**, Blobs **TBlobs** = Blobs::OnePerField, SubArrayAlignment **TSubArrayAlignment** = *TBlobs* == Blobs::Single ? SubArrayAlignment::Align : SubArrayAlignment::Pack, typename **TLinearizeArrayIndexFunctor** = *LinearizeArrayIndexRight*, template<typename> typename **PermuteFieldsSingleBlob** = *PermuteFieldsInOrder*>
struct **SoA** : public llama::mapping::MappingBase<*TArrayExtents*, *TRecordDim*>

Struct of array mapping. Used to create a *View* via *allocView*. We recommend to use multiple blobs when the array extents are dynamic and an aligned single blob version when they are static.

Template Parameters

- **TBlobs** – If OnePerField, every element of the record dimension is mapped to its own blob.
- **TSubArrayAlignment** – Only relevant when TBlobs == Single, ignored otherwise. If Align, aligns the sub arrays created within the single blob by inserting padding. If the array extents are dynamic, this may add some overhead to the mapping logic.
- **TLinearizeArrayIndexFunctor** – Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.
- **PermuteFieldsSingleBlob** – Defines how the record dimension's fields should be permuted if Blobs is Single. See *PermuteFieldsInOrder*, *PermuteFieldsIncreasingAlignment*, *PermuteFieldsDecreasingAlignment* and *PermuteFieldsMinimizePadding*.

template<typename **TArrayExtents**, typename **TRecordDim**, typename **TSelectorForMapping1**, template<typename...> typename **MappingTemplate1**, template<typename...> typename **MappingTemplate2**, bool **SeparateBlobs** = false>
struct **Split**

Mapping which splits off a part of the record dimension and maps it differently then the rest.

Template Parameters

- **TSelectorForMapping1** – Selects a part of the record dimension to be mapped by MappingTemplate1. Can be a *RecordCoord*, a type list of RecordCoords, a type list of tags (selecting one field), or a type list of type list of tags (selecting one field per sub list). dimension to be mapped differently.
- **MappingTemplate1** – The mapping used for the selected part of the record dimension.
- **MappingTemplate2** – The mapping used for the not selected part of the record dimension.
- **SeparateBlobs** – If true, both pieces of the record dimension are mapped to separate blobs.

template<typename **Mapping**, typename **TCountType** = std::size_t, bool **MyCodeHandlesProxyReferences** = true>
struct **FieldAccessCount** : public *Mapping*

Forwards all calls to the inner mapping. Counts all accesses made through this mapping and allows printing a summary.

Template Parameters

- **Mapping** – The type of the inner mapping.
- **TCountType** – The type used for counting the number of accesses.
- **MyCodeHandlesProxyReferences** – If false, *FieldAccessCount* will avoid proxy references but can then only count the number of address computations

struct **FieldHitsArray** : public llama::*Array*<AccessCounts<CountType>, flatFieldCount<RecordDim>>

Public Functions

inline auto **totalBytes**() const

When MyCodeHandlesProxyReferences is true, return a pair of the total read and written bytes. If false, returns the total bytes of accessed data as a single value.

struct **TotalBytes**

13.6.1 Acessors

struct **Default**

Default accessor. Passes through the given reference.

Subclassed by llama::accessor::internal::StackedLeave< 0, Default >, *llama::View< TMapping, TBlobType, TAccessor >*

struct **ByValue**

Allows only read access and returns values instead of references to memory.

struct **Const**

Allows only read access by qualifying the references to memory with const.

struct **Restrict**

Qualifies references to memory with __restrict. Only works on l-value references.

struct **Atomic**

Accessor wrapping a reference into a std::atomic_ref. Can only wrap l-value references.

template<typename ...**Accessors**>

struct **Stacked** : public llama::accessor::internal::StackedLeave<0, *Default*>

Accessor combining multiple other accessors. The contained accessors are applied in left to right order to the memory location when forming the reference returned from a view.

13.6.2 RecordDim field permuters

template<typename **TFlatRecordDim**>

struct **PermuteFieldsInOrder**

Retains the order of the record dimension's fields.

template<typename **FlatOrigRecordDim**, template<typename, typename> typename **Less**>

struct **PermuteFieldsSorted**

Sorts the record dimension's the fields according to a given predicate on the field types.

Template Parameters

Less – A binary predicate accepting two field types, which exposes a member value. Value must be true if the first field type is less than the second one, otherwise false.

template<typename **FlatRecordDim**>

using `llama::mapping::PermuteFieldsIncreasingAlignment` = *PermuteFieldsSorted*<*FlatRecordDim*, internal::LessAlignment>

Sorts the record dimension fields by increasing alignment of its fields.

template<typename **FlatRecordDim**>

using `llama::mapping::PermuteFieldsDecreasingAlignment` = *PermuteFieldsSorted*<*FlatRecordDim*, internal::MoreAlignment>

Sorts the record dimension fields by decreasing alignment of its fields.

template<typename **FlatRecordDim**>

using `llama::mapping::PermuteFieldsMinimizePadding` = *PermuteFieldsIncreasingAlignment*<*FlatRecordDim*>

Sorts the record dimension fields by the alignment of its fields to minimize padding.

13.6.3 Common utilities

struct **LinearizeArrayIndexRight**

Functor that maps an *ArrayIndex* into linear numbers, where the fast moving index should be the rightmost one, which models how C++ arrays work and is analogous to mdspan's layout_right. E.g. ArrayIndex<3> a; stores 3 indices where a[2] should be incremented in the innermost loop.

Public Functions

template<typename **ArrayExtents**>
inline constexpr auto **operator()**(const typename *ArrayExtents*::Index &ai, const *ArrayExtents* &extents) const -> typename *ArrayExtents*::value_type

Parameters

- **ai** – Index in the array dimensions.
- **extents** – Total size of the array dimensions.

Returns
Linearized index.

struct **LinearizeArrayIndexLeft**

Functor that maps a *ArrayIndex* into linear numbers the way Fortran arrays work. The fast moving index of the *ArrayIndex* object should be the last one. E.g. ArrayIndex<3> a; stores 3 indices where a[0] should be incremented in the innermost loop.

Public Functions

template<typename **ArrayExtents**>
inline constexpr auto **operator()**(const typename *ArrayExtents*::Index &ai, const *ArrayExtents* &extents) const -> typename *ArrayExtents*::value_type

Parameters

- **ai** – Index in the array dimensions.
- **extents** – Total size of the array dimensions.

Returns
Linearized index.

struct **LinearizeArrayIndexMorton**

Functor that maps an *ArrayIndex* into linear numbers using the Z-order space filling curve (Morton codes).

Public Functions

template<typename **ArrayExtents**>
inline constexpr auto **operator()**(const typename *ArrayExtents*::Index &ai, [[maybe_unused]] const *ArrayExtents* &extents) const -> typename *ArrayExtents*::value_type

Parameters

- **ai** – Coordinate in the array dimensions.
- **extents** – Total size of the array dimensions.

Returns
Linearized index.

13.6.4 Dumping

Warning: doxygenfunction: Cannot find function "llama::toSvg" in doxygen xml output for project "LLAMA" from directory: ./doxygen/xml

Warning: doxygenfunction: Cannot find function "llama::toHtml" in doxygen xml output for project "LLAMA" from directory: ./doxygen/xml

13.7 Data access

template<typename **TMapping**, typename **TBlobType**, typename **TAccessor** = accessor::*Default*>

struct **View** : private *TMapping*, private llama::accessor::*Default*

Central LLAMA class holding memory for storage and giving access to values stored there defined by a mapping. A view should be created using *allocView*.

Template Parameters

- **TMapping** – The mapping used by the view to map accesses into memory.
- **TBlobType** – The storage type used by the view holding memory.
- **TAccessor** – The accessor to use when an access is made through this view.

Public Functions

View() = default

Performs default initialization of the blob array.

inline explicit **View**(Mapping mapping, *Array*<BlobType, Mapping::blobCount> blobs = {}, Accessor accessor = {})

Creates a LLAMA *View* manually. Prefer the allocations functions *allocView* and *allocViewUninitialized* if possible.

Parameters

- **mapping** – The mapping used by the view to map accesses into memory.
- **blobs** – An array of blobs providing storage space for the mapped data.
- **accessor** – The accessor to use when an access is made through this view.

inline auto **operator()**(*ArrayIndex* ai) const -> decltype(auto)

Retrieves the *RecordRef* at the given *ArrayIndex* index.

template<typename ...**Indices**, std::enable_if_t<std::conjunction_v<std::is_convertible<*Indices*, size_type>...>, int> = 0>
inline auto **operator()**(*Indices*... indices) const -> decltype(auto)

Retrieves the *RecordRef* at the *ArrayIndex* index constructed from the passed component indices.

inline auto **operator[]**(*ArrayIndex* ai) const -> decltype(auto)

Retrieves the *RecordRef* at the *ArrayIndex* index constructed from the passed component indices.

inline auto **operator[]**(size_type index) const -> decltype(auto)

Retrieves the *RecordRef* at the 1D *ArrayIndex* index constructed from the passed index.

template<typename **TStoredParentView**>

struct **SubView**

Like a *View*, but array indices are shifted.

Template Parameters

TStoredParentView – Type of the underlying view. May be cv qualified and/or a reference type.

Public Types

using **ParentView** = std::remove_const_t<std::remove_reference_t<StoredParentView>>

type of the parent view

Public Functions

inline explicit **SubView**(*ArrayIndex* offset)

Creates a *SubView* given an offset. The parent view is default constructed.

template<typename **StoredParentViewFwd**>
inline **SubView**(*StoredParentViewFwd* &&parentView, *ArrayIndex* offset)

Creates a *SubView* given a parent *View* and offset.

inline auto **operator()**(*ArrayIndex* ai) const -> decltype(auto)

Same as View::operator()(ArrayIndex), but shifted by the offset of this *SubView*.

template<typename ...**Indices**>
inline auto **operator()**(*Indices*... indices) const -> decltype(auto)

Same as corresponding operator in *View*, but shifted by the offset of this *SubView*.

Public Members

const *ArrayIndex* **offset**

offset by which this view's *ArrayIndex* indices are shifted when passed to the parent view.

template<typename **TView**, typename **TBoundRecordCoord**, bool **OwnView**>

struct **RecordRef** : private *TView*::Mapping::ArrayExtents::Index

Record reference type returned by *View* after resolving an array dimensions coordinate or partially resolving a *RecordCoord*. A record reference does not hold data itself, it just binds enough information (array dimensions coord and partial record coord) to retrieve it later from a *View*. Records references should not be created by the user. They are returned from various access functions in *View* and *RecordRef* itself.

Public Types

using **View** = *TView*

View this record reference points into.

using **BoundRecordCoord** = *TBoundRecordCoord*

Record coords into View::RecordDim which are already bound by this *RecordRef*.

using **AccessibleRecordDim** = *GetType*<RecordDim, *BoundRecordCoord*>

Subtree of the record dimension of *View* starting at BoundRecordCoord. If BoundRecordCoord is *RecordCoord*<> (default) AccessibleRecordDim is the same as `Mapping::RecordDim`.

Public Functions

inline **RecordRef**()

Creates an empty *RecordRef*. Only available for if the view is owned. Used by *llama::One*.

template<typename **OtherView**, typename **OtherBoundRecordCoord**, bool **OtherOwnView**>
inline **RecordRef**(const *RecordRef*<*OtherView*, *OtherBoundRecordCoord*, *OtherOwnView*> &recordRef)

Create a *RecordRef* from a different *RecordRef*. Only available for if the view is owned. Used by *llama::One*.

template<typename **T**, typename = std::enable_if_t<!isRecordRef<*T*>>>
inline explicit **RecordRef**(const *T* &scalar)

Create a *RecordRef* from a scalar. Only available for if the view is owned. Used by *llama::One*.

template<std::size_t... **Coord**>
inline auto **operator()**(*RecordCoord*<*Coord*...>) const -> decltype(auto)

Access a record in the record dimension underneath the current record reference using a *RecordCoord*. If the access resolves to a leaf, an l-value reference to a variable inside the *View* storage is returned, otherwise another *RecordRef*.

template<typename ...**Tags**>
inline auto **operator()**(*Tags*...) const -> decltype(auto)

Access a record in the record dimension underneath the current record reference using a series of tags. If the access resolves to a leaf, an l-value reference to a variable inside the *View* storage is returned, otherwise another *RecordRef*.

struct **Loader**

struct **LoaderConst**

13.8 Copying

template<typename **SrcMapping**, typename **SrcBlob**, typename **DstMapping**, typename **DstBlob**>
void **llama::copy**(const *View*<*SrcMapping*, *SrcBlob*> &srcView, *View*<*DstMapping*, *DstBlob*> &dstView,
std::size_t threadId = 0, std::size_t threadCount = 1)

Copy data from source to destination view. Both views need to have the same array and record dimensions, but may have different mappings. The blobs need to be read- and writeable. Delegates to *Copy* to choose an implementation.

Parameters

- **threadId** – Optional. Zero-based id of calling thread for multi-threaded invocations.
- **threadCount** – Optional. Thread count in case of multi-threaded invocation.

template<typename **SrcMapping**, typename **DstMapping**, typename **SFINAE** = void>

struct **Copy**

Generic implementation of *copy* defaulting to *fieldWiseCopy*. LLAMA provides several specializations of this construct for specific mappings. Users are encouraged to also specialize this template with better copy algorithms for further combinations of mappings, if they can and want to provide a better implementation.

template<typename **SrcMapping**, typename **SrcBlob**, typename **DstMapping**, typename **DstBlob**>

void `llama::fieldWiseCopy`(const *View*<*SrcMapping*, *SrcBlob*> &srcView, *View*<*DstMapping*, *DstBlob*> &dstView, std::size_t threadId = 0, std::size_t threadCount = 1)

Field-wise copy from source to destination view. Both views need to have the same array and record dimensions.

Parameters

- **threadId** – Optional. Thread id in case of multi-threaded copy.
- **threadCount** – Optional. Thread count in case of multi-threaded copy.

template<typename **SrcMapping**, typename **SrcBlob**, typename **DstMapping**, typename **DstBlob**>
void `llama::aosoaCommonBlockCopy`(const *View*<*SrcMapping*, *SrcBlob*> &srcView, *View*<*DstMapping*, *DstBlob*> &dstView, std::size_t threadId = 0, std::size_t threadCount = 1)

AoSoA copy strategy which transfers data in common blocks. SoA mappings are also allowed for at most 1 argument.

Parameters

- **threadId** – Optional. Zero-based id of calling thread for multi-threaded invocations.
- **threadCount** – Optional. Thread count in case of multi-threaded invocation.

13.9 SIMD

template<typename **Simd**, typename **SFINAE** = void>

struct **SimdTraits**

Traits of a specific Simd implementation. Please specialize this template for the SIMD types you are going to use in your program. Each specialization SimdTraits<Simd> must provide:

- an alias `value_type` to indicate the element type of the Simd.
- a `static constexpr size_t lanes` variable holding the number of SIMD lanes of the Simd.
- a `static auto loadUnalinged(const value_type* mem) -> Simd` function, loading a Simd from the given memory address.
- a `static void storeUnaligned(Simd simd, value_type* mem)` function, storing the given Simd to a given memory address.
- a `static auto gather(const value_type* mem, std::array<int, lanes> indices) -> Simd` function, gathering values into a Simd from the memory addresses identified by mem + indices * sizeof(value_type).
- a `static void scatter(Simd simd, value_type* mem, std::array<int, lanes> indices)` function, scattering the values from a Simd to the memory addresses identified by mem + indices * sizeof(value_type).

template<typename **Simd**, typename **SFINAE** = void>

constexpr auto `llama::simdLanes` = *SimdTraits*<*Simd*>::lanes

The number of SIMD simdLanes the given SIMD vector or *Simd<T>* has. If Simd is not a structural *Simd* or *SimdN*, this is a shortcut for SimdTraits<Simd>::lanes.

template<typename **RecordDim**, std::size_t **N**, template<typename, auto> typename **MakeSizedSimd**>

using `llama::SimdizeN` = typename internal::SimdizeNImpl<*RecordDim*, *N*, *MakeSizedSimd*>::type

Transforms the given record dimension into a SIMD version of it. Each leaf field type will be replaced by a sized SIMD vector with length N, as determined by MakeSizedSimd. If N is 1, SimdizeN<T, 1, ...> is an alias for T.

template<typename **RecordDim**, template<typename> typename **MakeSimd**>

using `llama::Simdize` = *TransformLeaves*<*RecordDim*, *MakeSimd*>

Transforms the given record dimension into a SIMD version of it. Each leaf field type will be replaced by a SIMD vector, as determined by MakeSimd.

template<typename **RecordDim**, template<typename> typename **MakeSimd**>

constexpr std::size_t `llama::simdLanesWithFullVectorsFor`

Determines the number of simd lanes suitable to process all types occurring in the given record dimension. The algorithm ensures that even SIMD vectors for the smallest field type are filled completely and may thus require multiple SIMD vectors for some field types.

Template Parameters

- **RecordDim** – The record dimension to simdize
- **MakeSimd** – Type function creating a SIMD type given a field type from the record dimension.

template<typename **RecordDim**, template<typename> typename **MakeSimd**>

constexpr std::size_t `llama::simdLanesWithLeastRegistersFor`

Determines the number of simd lanes suitable to process all types occurring in the given record dimension. The algorithm ensures that the smallest number of SIMD registers is needed and may thus only partially fill registers for some data types.

Template Parameters

- **RecordDim** – The record dimension to simdize
- **MakeSimd** – Type function creating a SIMD type given a field type from the record dimension.

template<typename **T**, std::size_t **N**, template<typename, auto> typename **MakeSizedSimd**>

using `llama::SimdN` = typename std::conditional_t<isRecordDim<*T*>, std::conditional_t<*N* == 1, mp_identity<*One*<*T*>>, mp_identity<*One*<*SimdizeN*<*T*, *N*, *MakeSizedSimd*>>>>, std::conditional_t<*N* == 1, mp_identity<*T*>, mp_identity<*SimdizeN*<*T*, *N*, *MakeSizedSimd*>>>>::type

Creates a SIMD version of the given type. Of T is a record dimension, creates a *One* where each field is a SIMD type of the original field type. The SIMD vectors have length N. If N is 1, an ordinary *One* of the record dimension T is created. If T is not a record dimension, a SIMD vector with value T and length N is created. If N is 1 (and T is not a record dimension), then T is produced.

template<typename **T**, template<typename> typename **MakeSimd**>

using `llama::Simd` = typename std::conditional_t<isRecordDim<*T*>, mp_identity<*One*<*Simdize*<*T*, *MakeSimd*>>>, mp_identity<*Simdize*<*T*, *MakeSimd*>>>::type

Creates a SIMD version of the given type. Of T is a record dimension, creates a *One* where each field is a SIMD type of the original field type.

template<typename **T**, typename **Simd**>
inline void `llama::loadSimd`(const *T* &srcRef, *Simd* &dstSimd)

Loads SIMD vectors of data starting from the given record reference to dstSimd. Only field tags occurring in *RecordRef* are loaded. If Simd contains multiple fields of SIMD types, a SIMD vector will be fetched for each of the fields. The number of elements fetched per SIMD vector depends on the SIMD width of the vector. Simd is allowed to have different vector lengths per element.

template<typename **Simd**, typename **TFwd**>

inline void `llama::storeSimd`(const *Simd* &srcSimd, *TFwd* &&dstRef)

Stores SIMD vectors of element data from the given srcSimd into memory starting at the provided record reference. Only field tags occurring in *RecordRef* are stored. If Simd contains multiple fields of SIMD types, a SIMD vector will be stored for each of the fields. The number of elements stored per SIMD vector depends on the SIMD width of the vector. Simd is allowed to have different vector lengths per element.

template<std::size_t **N**, template<typename, auto> typename **MakeSizedSimd**, typename **View**, typename **UnarySimdFunction**>
void `llama::simdForEachN`(*View* &view, *UnarySimdFunction* f)

template<template<typename> typename **MakeSimd**, template<typename, auto> typename **MakeSizedSimd**, typename **View**, typename **UnarySimdFunction**>
void `llama::simdForEach`(*View* &view, *UnarySimdFunction* f)

13.10 Macros

LLAMA_INDEPENDENT_DATA

May be put in front of a loop statement. Indicates that all (!) data access inside the loop is indepent, so the loop can be safely vectorized. Example:

```
LLAMA_INDEPENDENT_DATA
for(int i = 0; i < N; ++i)
    // because of LLAMA_INDEPENDENT_DATA the compiler knows that a and b
    // do not overlap and the operation can safely be vectorized
    a[i] += b[i];
```

LLAMA_FORCE_INLINE

Forces the compiler to inline a function annotated with this macro.

LLAMA_UNROLL(...)

Requests the compiler to unroll the loop following this directive. An optional unrolling count may be provided as argument, which must be a constant expression.

LLAMA_HOST_ACC

Some offloading parallelization language extensions such a CUDA, OpenACC or OpenMP 4.5 need to specify whether a class, struct, function or method "resides" on the host, the accelerator (the offloading device) or both. LLAMA supports this with marking every function needed on an accelerator with `LLAMA_HOST_ACC`.

LLAMA_FN_HOST_ACC_INLINE

LLAMA_LAMBDA_INLINE

Gives strong indication to the compiler to inline the attributed lambda.

LLAMA_COPY(x)

Forces a copy of a value. This is useful to prevent ODR usage of constants when compiling for GPU targets.

CHAPTER

FOURTEEN

LLAMA VS. C++

LLAMA tries hard to provide experience and constructs similar to native C++. The following tables compare how various constructs in C++ translate to LLAMA:

14.1 Containers and views

Construct	Native C++	LLAMA	LLAMA (alternative)

Defining structs/records	`struct VecCpp {`
	`  float x;`
	`  float y;`
	`};`
	`struct ParticleCpp {`
	`  VecCpp pos;`
	`  float mass;`
	`  bool flags[3];`
	`};`
		`struct X{}; struct Y {};`
		`struct Pos{};`
		`struct Mass{};`
		`struct Flags{};`
		`using VecRec = llama::Record<`
		`  llama::Field<X, float>,`
		`  llama::Field<Y, float>`
		`>;`
		`using ParticleRec = llama::Record<`
		`  llama::Field<Pos, VecRec>,`
		`  llama::Field<Mass, float>,`
		`  llama::Field<Flags, bool[3]>`
		`>;`	

Defining array extents	`using size_type = ...;` `size_type n = ...;`	`using ArrayExtents = ...;` `ArrayExtents n = ...;`	
Defining the memory layout	-	`using Mapping = ...;` `Mapping m(n, ...);`	
A collection of n things in memory	`std::vector<ParticleCpp> view(n);`	`auto view = llama::allocView(m);`	`llama::View<ArrayExtents, ParticleRec, ...> view;` Useful for static array dimensions.

14.2 Values and references

Construct	Native C++	LLAMA	LLAMA (alternative)	wrong
Declare single local record	`ParticleCpp p;`	`llama::One<ParticleRec> p`	`ParticleCpp p;` Or any type layout compatible type supporting the tuple interface.	`ParticleRec p;` ParticleRec is an empty struct (a type list)!
Copy memory -> local	`p = view[i];`	`p = view[i];`	`p = view[i];` Assigns field by field using tuple interface.	
Copy local -> memory	`view[i] = p;`	`view[i] = p;`	`view[i] = p;` Assigns field by field using tuple interface.	
Copy a single record from memory to local	`ParticleCpp p = view[i];`	`llama::One<ParticleRec> p = view[i];`	`ParticleCpp p = view[i];` Assigns field by field using tuple interface	`auto p = view[i];` p is a reference, not a copy!
Create a reference to a single record in memory	`ParticleCpp& p = view[i];`	`auto p = view[i];` `// decltype(p) == llama::RecordRe<...>`	`auto&& p = view[i];`	`auto& p = view[i];` Compilation error!
Copy a single sub-record from memory to local	`VecCpp v = view[i].pos;`	`llama::One<VecRec> v = view[i](Pos{});`	`VecRec v = view[i](Pos{});` Assigns field by field using tuple interface.	`auto v = view[i](Pos{});` v is a reference, not a copy!
Create a reference to a single sub-record in memory	`VecCpp& v = view[i].pos;`	`auto v = view[i](Pos{});` `// decltype(v) == llama::RecordRe<...>`	`auto&& v = view[i](Pos{});`	`auto& p = view[i](Pos{});` Compilation error!
Copy a single record leaf field from memory to local	`float y = view[i].pos.y;`	`float y = view[i](Pos{}, Y{});`	`float y = view[i](Pos{})(Y{});`	
Create a reference to a single leaf field in memory	`float& y = `	`float& y = `	`auto&& y = `	`auto y = `

Notice that the use of `auto` to declare a local copy of a value read through a reference, e.g. `auto pos = view[i].pos; // copy`, does not work as expected in LLAMA. LLAMA makes extensive use of proxy reference types (including `llama::RecordRef`), where a reference is sometimes represented as a value and sometimes as a real C++ reference. The only consistent way to deal with this duality in LLAMA is the use a forwarding reference `auto&&` when we want to have a reference (native or proxy) into a LLAMA data structure, and to use a concrete type when we want to make a copy.

INDEX

L