LLAMA Documentation

Release 0.4

Alexander Matthes, Bernhard Manfred Gruber

Jan 25, 2022

USER DOCUMENTATION

LLAMA is a cross-platform C++17 template header-only library for the abstraction of memory access patterns. It distinguishes between the view of the algorithm on the memory and the real layout in the background. This enables performance portability for multicore, manycore and gpu applications with the very same code.

In contrast to many other solutions LLAMA can define nested data structures of arbitrary depths. It is not limited to struct of array and array of struct data layouts but also capable to explicitly define memory layouts with padding, blocking, striding or any other run time or compile time access pattern.

To achieve this goal LLAMA is split into mostly independent, orthogonal parts completely written in modern C++17 to run on as many architectures and with as many compilers as possible while still supporting extensions needed e.g. to run on GPU or other many core hardware.

LLAMA is licensed under the LGPL3+.

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

- Boost 1.70.0 or higher
- libfmt 6.2.1 or higher (optional) for building the tests and examples and LLAMA supporting to dump mappings as SVG/HTML
- Alpaka (optional) for building some examples
- Vc (optional) for building some examples

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 ..
ccmake .. // optionally change configuration after first run of cmake
cmake --build .
```

This will search for all depenencies and create a build system for your platform. If Alpaka or Vc is 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. The tests can be disabled by setting *BUILD_TESTING* to *OFF* (default: *ON*). The examples can be disabled by setting *LLAMA_BUILD_EXAMPLES* to *OFF* (default: *ON*).

1.4 Install LLAMA

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

```
sudo 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 the proposed 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 heterogenous 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.

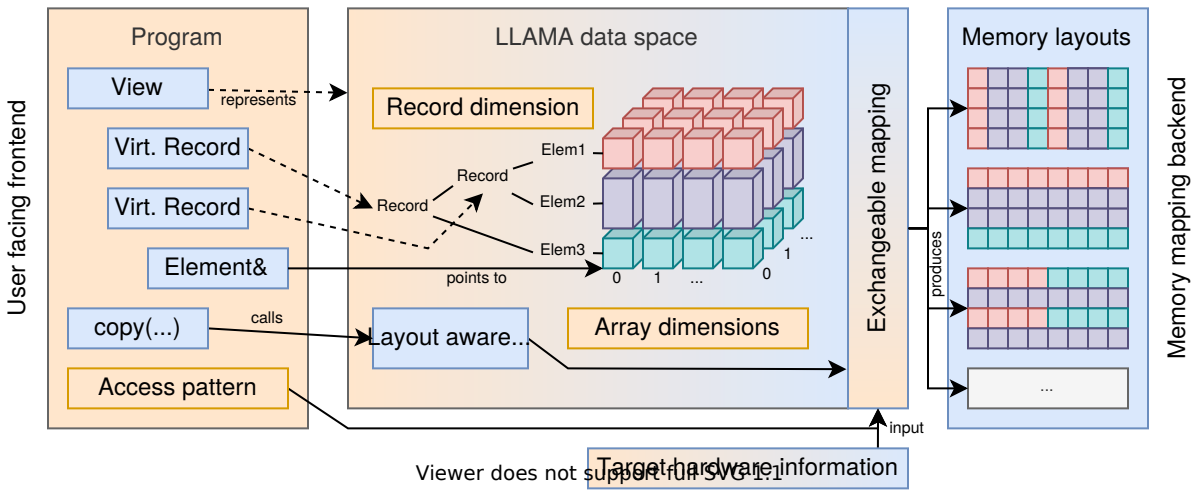
Furthermore, third party libraries may expect specific memory layouts at their interface, into which custom data structures need to be converted.

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, softwares, compilers and third party libraries, LLAMA tries to stay within C++17. 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

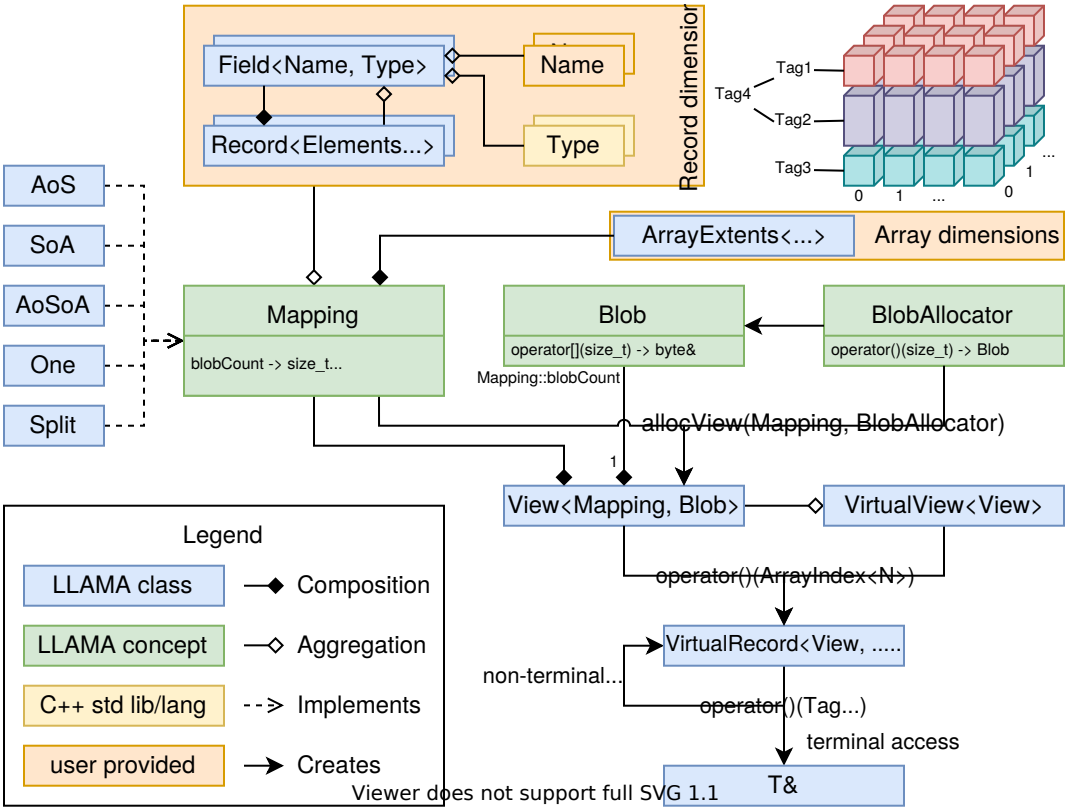


LLAMA separates the data structure access and physical memory layout by an opaque abstract data type called data space. The data space is an hypercubic index set described by the record dimension and one or more array dimensions. The record dimension consistes of a hierarchy of names and describes nested, 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 virtual records 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 *VirtualView* 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 VirtualView, by calling `operator()` with an array index as instance of `ArrayIndex`. This access returns a *VirtualRecord*, 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 *run time* whereas the record dimension is defined 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 CTAD. The full type of `extents` is `llama::ArrayExtents<llama::dyn, llama::dyn, llama::dyn>`.

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

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

The template argument list specifies the order and nature (compile vs. runtime) of the extents. 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<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<128, 256, 32> extents{};
static_assert(sizeof(extents) == 1); // no object can have size 0
struct S : llama::ArrayExtents<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<3>
```

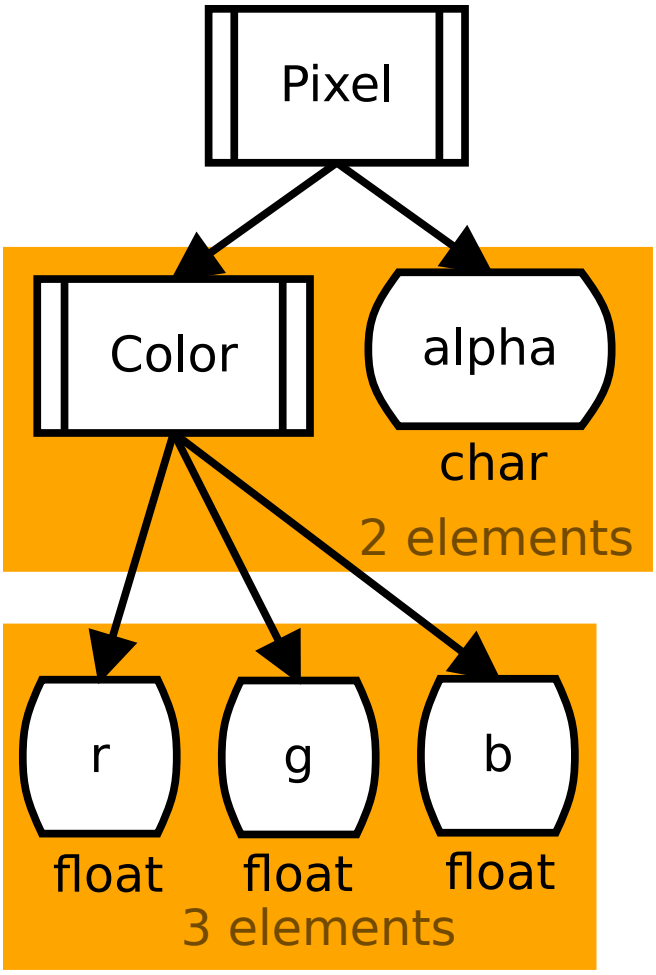
Contrary to `llama::ArrayExtents` which can store a mix of compile and runtime values, `llama::ArrayIndex` only stores runtime indices, so it is templated on the number of dimensions. This might change at some point in the future, if we find sufficient evidence that a design similar to `llama::ArrayExtents`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

4.3 VirtualView

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

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

CHAPTER

FIVE

VIRTUALRECORD

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::VirtualRecord`.

```
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::VirtualRecord`, storing this array dimensions coordiante. This object can be thought of like 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 real object in the C++ sense and thus called virtual.

Accessing subparts of a `llama::VirtualRecord` 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::VirtualRecord` is returned, binding the tags already used for navigating down the record dimension.

A `llama::VirtualRecord` 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::VirtualRecord` 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::VirtualRecord`. 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::VirtualRecord` 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 operatores

`llama::VirtualRecord` 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 VirtualRecord>
void foobar(VirtualRecord 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 virtual records, 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::VirtualRecord` 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 virtual record like `record1(color{}) *= 7.0` is also possible. `record1(color{})` itself returns a new virtual record with the first record dimension coordiante (`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 virtual record with the record dimension "Particle".

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

5.3 Tuple interface

WARNING: This is an experimental feature and might completely change in the future.

A struct in C++ can be modelled by a `std::tuple` with the same types as the struct's members. A `llama::VirtualRecord` 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::VirtualRecord`. 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::VirtualRecord` can integreate with these using the `load()`, `loadAs<T>()` and `store(T)` functions.

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

(continues on next page)

(continued from previous page)

```
// implement std::tuple_size<MyPixel>, std::tuple_element<MyPixel> and get(MyPixel)

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::VirtualRecord`.

5.4 Structured bindings

WARNING: This is an experimental feature and might completely change in the future.

A `llama::VirtualRecord` implementes the C++ tuple interface itself to allow destructuring:

```
auto record = view(1, 2, 3);
auto [color, a] = record; // color is another VirtualRecord, 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::VirtualRecord` 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 recommented 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{});
```

A more detailed example can be found in the simpletest example.

6.3 View iterators

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

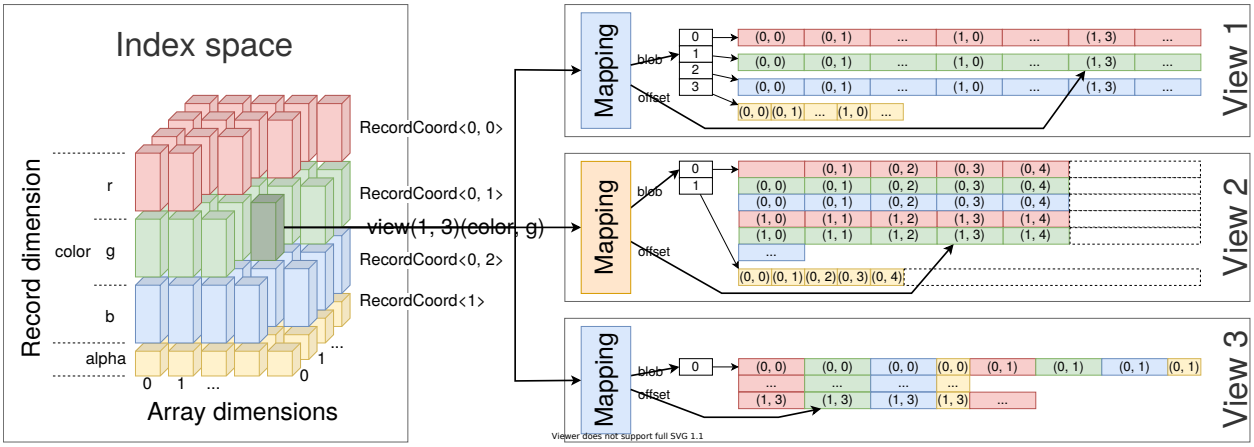
```
using Pixel = ...;
using ArrayExtents = llama::ArrayExtents<llama::dyn>;
// ...
auto view = llama::allocView(mapping);
// ...

// 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 fileds 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::ArrayIndex;
    typename M::RecordDim;
    { M::blobCount } -> std::convertible_to<std::size_t>;
    { m.blobSize(std::size_t{}) } -> std::same_as<std::size_t>;
};
```

That is, each mapping type needs to expose the types `M::ArrayExtents`, `M::ArrayIndex` and `M::RecordDim`. Furthermore, each mapping needs to 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. `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>;
};
```

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.

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::LinearizeArrayDimsFortran> mapping{extents}; // pack fields, column␣
↪major
```

By default, the array dimensions spanned by `ArrayExtents` are linearized using `llama::mapping::LinearizeArrayDimsCpp`. 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 contigiously 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::LinearizeArrayDimsFortran> mapping{extents}; // separate blob for␣
↪each attribute, column major
```

By default, the array dimensions spanned by `ArrayExtents` are linearized using `llama::mapping::LinearizeArrayDimsCpp` 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::LinearizeArrayDimsFortran> mapping{extents}; // inner array has 8␣
↪values, column major
```

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

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 the `llama::One` virtual record, but also offers interesting applications in conjunction with the `llama::mapping::Split` mapping.

7.3.6 Split

WARNING: This is an experimental feature and might completely change in the future.

The Split mapping is a meta mapping in the sense, that it transforms the record dimension and delegates mapping to other mappings. Using a record coordinate, 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};
...
std::ofstream("plot.sh") << mapping.toGnuplotScript();
```

7.3.8 Trace

The Trace 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 the stdout when requested or the mapping instance is destroyed.

```
{
    auto anyMapping = ...;
    llama::mapping::Trace mapping{anyMapping};
    ...
    mapping.print(); // print report explicitely
} // report is printed implicitely
```

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.

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

7.3.11 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.

```
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.

7.3.12 BitPackedIntSoA

The BitPackedIntSoA mapping is a fully computed mapping that bitpacks integral values to reduce size and precision. The bits are stored as struct of arrays. 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
```

7.3.13 BitPackedFloatSoA

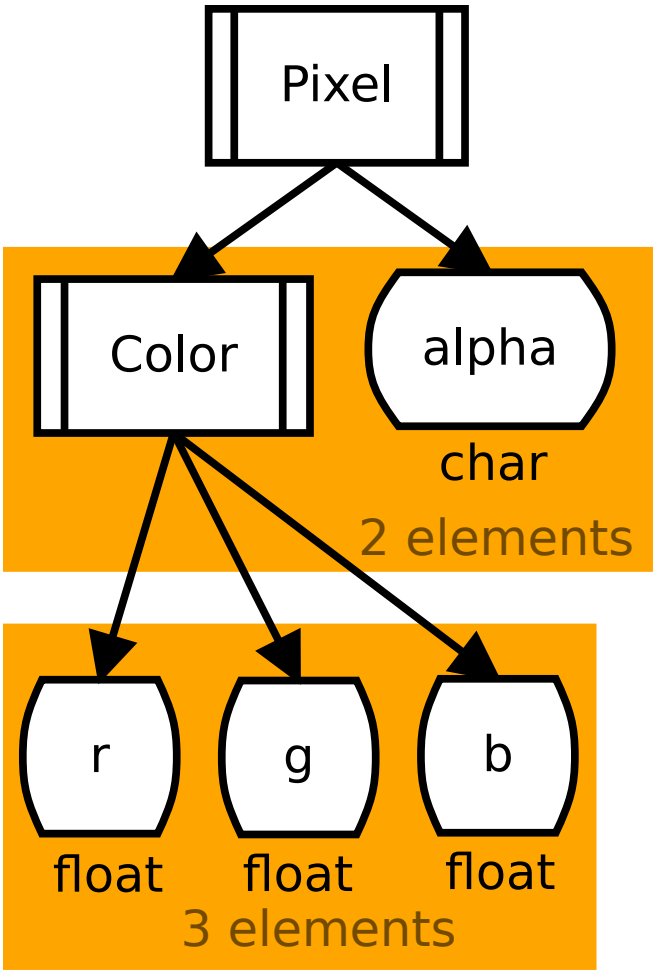
The BitPackedFloatSoA mapping is a fully computed mapping that bitpacks floating-point values to reduce size and precision. The bits are stored as struct of arrays. The number of bits used to store the exponent and mantissa is configurable. All field types in the record dimension must be floating-point. This 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
```

7.3.14 Tree (deprecated)

WARNING: The tree mapping is currently not maintained and we consider deprecation.

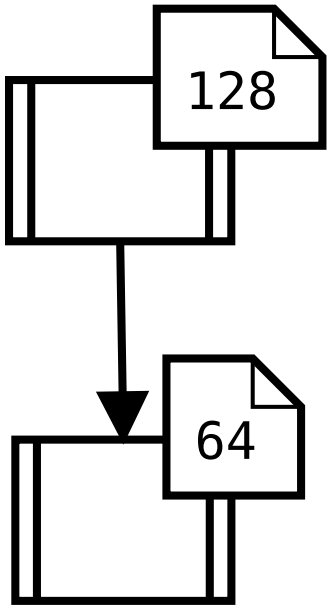
The LLAMA tree mapping is one approach to archieve the goal of mixing different mapping approaches. Furthermore, it tries to establish a general mapping description language and mapping definition framework. Let's take the example record dimension from the *dimensions section*:



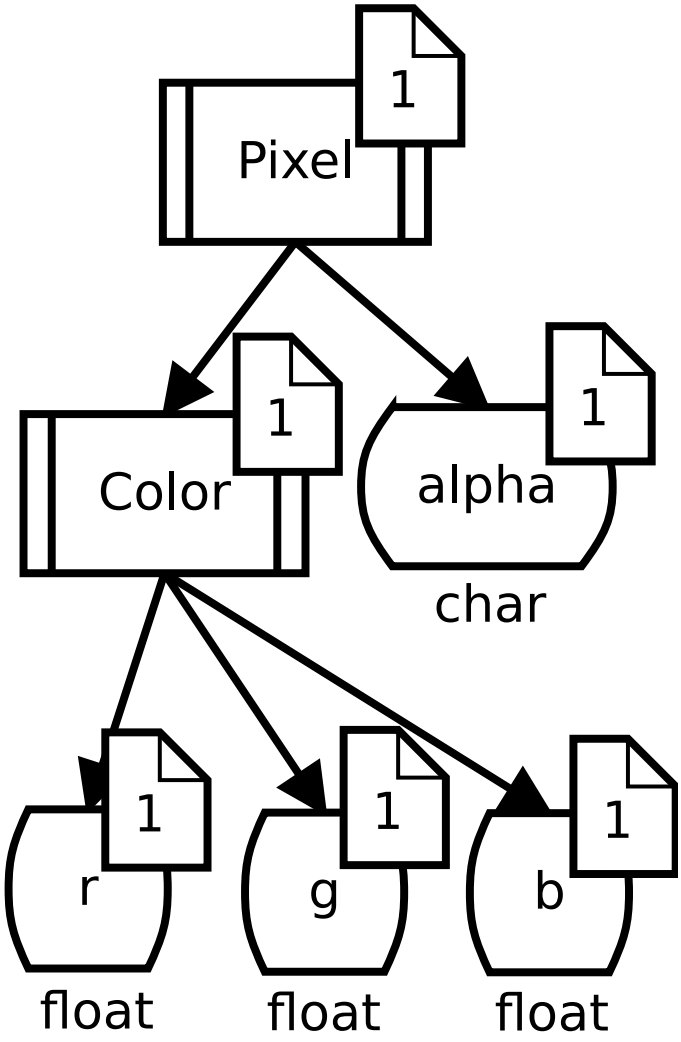
As already mentioned this is a compile time tree. The idea of the tree mapping is now to extend this model to a compile time tree with run time annotations representing the repetition of branches and to define tree operations which create

new trees out of the old ones while providing methods to translate tree coordinates from one tree to another.

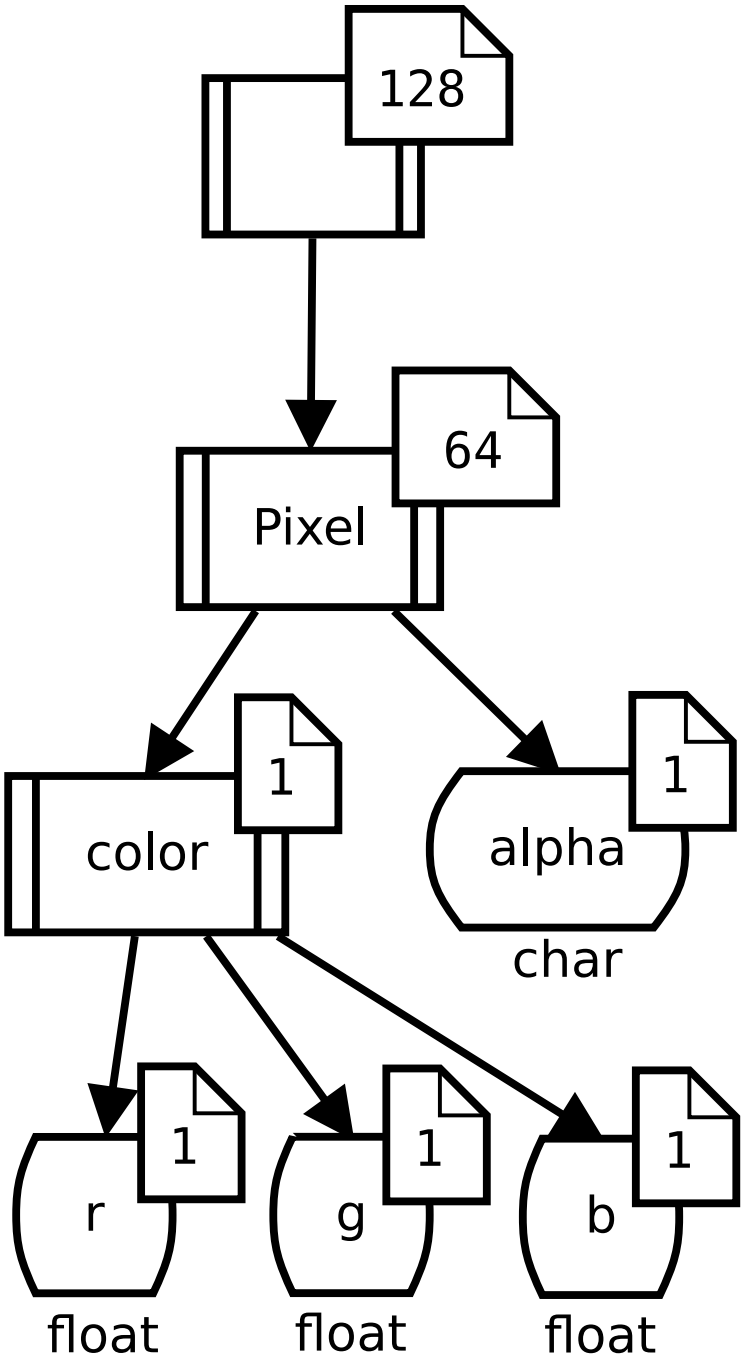
This is best demonstrated by an example. First of all the array dimensions needs to be represented as such an tree too. Let's assume array dimensions of 128×64:



The record dimension is already a tree, but as it has no run time influence, only 1 is annotated for these tree nodes:

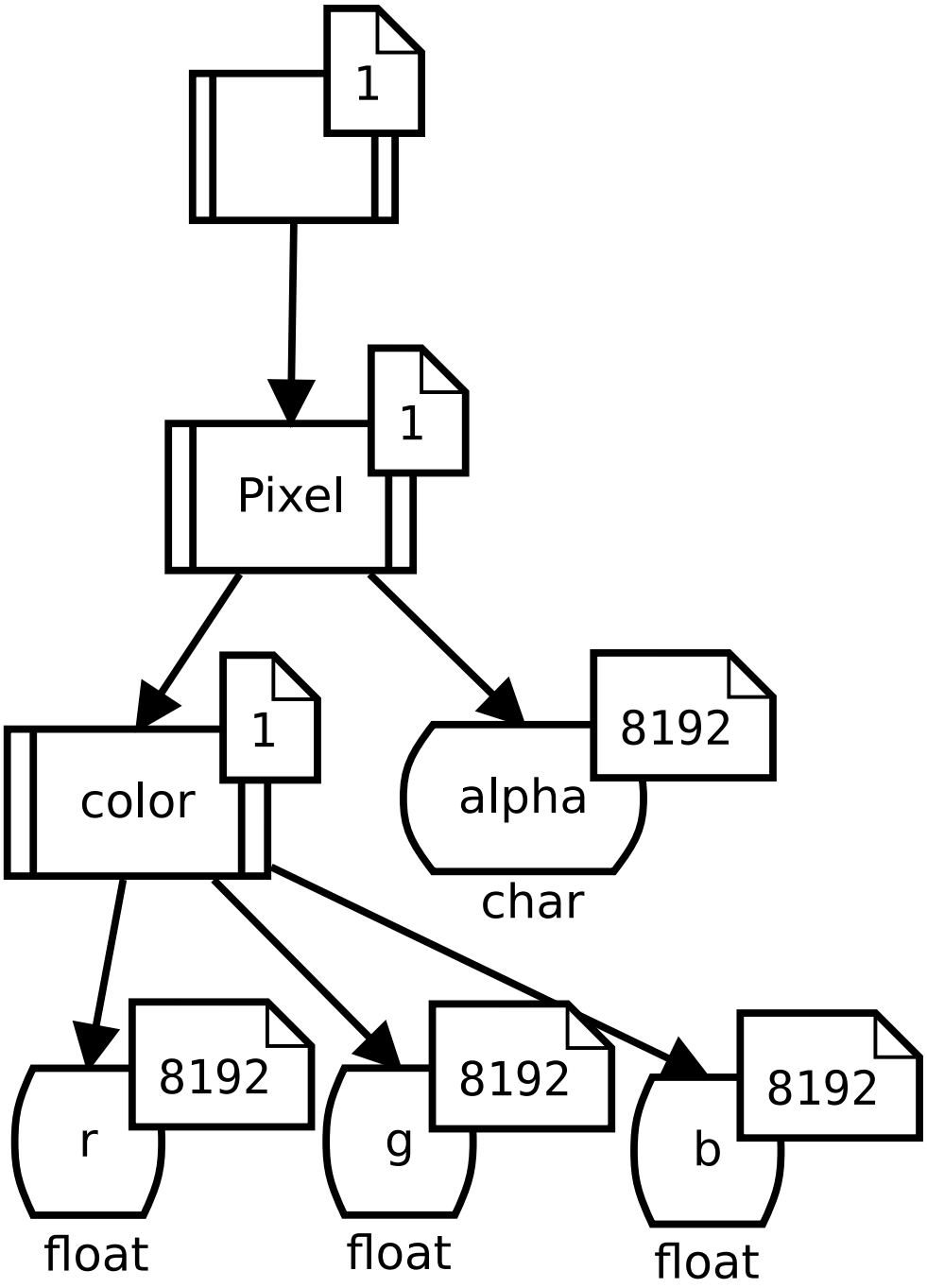


Now the two trees are connected so that we can represent array and record dimensions with one tree:

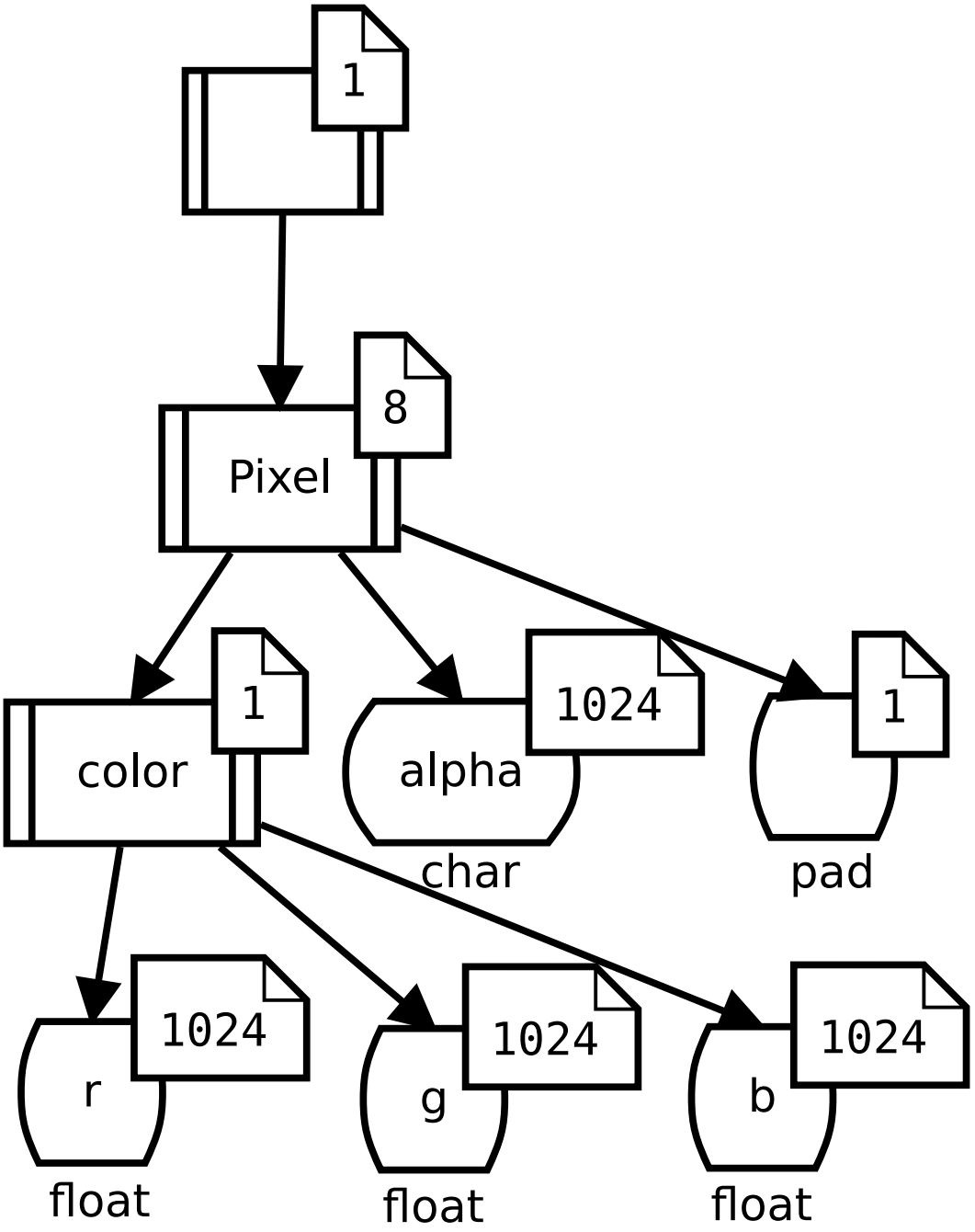


The mapping works now in this way that the tree is "flattened" from left to right using a breadth first traversal. Annotations represent repetitions of the node branches. So for this tree we would copy the record dimension 64 times and 128 times again – basically this results in an array of struct approach, which is most probably not desired.

So we want to transform the tree before flattening it. A struct of array approach may look like this:

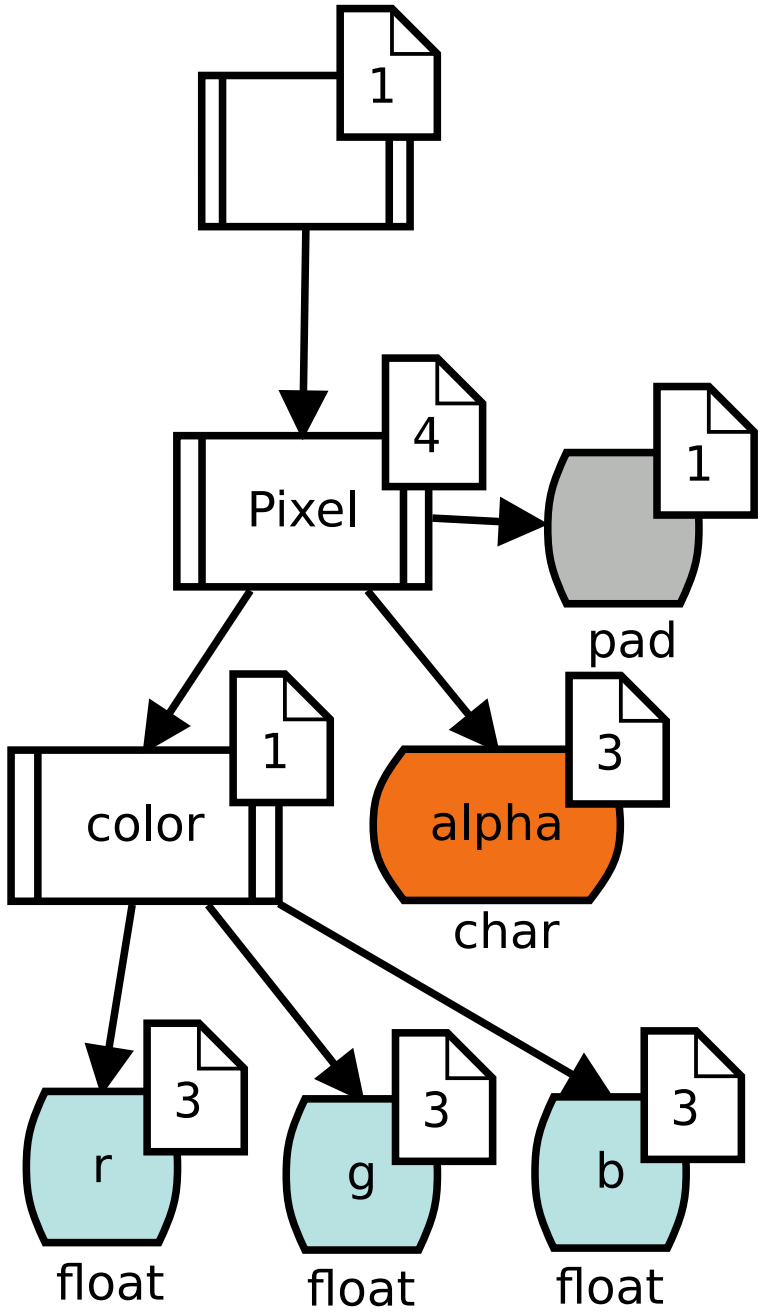


Struct of array but with a padding after each 1024 elements may look like this:



The size of the leaf type in “pad” of course needs to be determined based on the desired aligment and sub tree sizes.

Such a tree (with smaller array dimensions for easier drawing) …



… may look like this mapped to memory:

r	r	r	g	g	g	b	b	b	aaa	pad
r	r	r	g	g	g	b	b	b	aaa	pad
r	r	r	g	g	g	b	b	b	aaa	pad
r	r	r	g	g	g	b	b	b	aaa	pad

In code a tree mapping is defined as `llama::mapping::tree::Mapping`, but takes one more template parameter for the type of a tuple of tree operations and a further constructor parameter for the instantiation of this tuple.

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

using Mapping = llama::mapping::tree::Mapping<
    ArrayExtents,
    RecordDim,
    decltype(treeOperationList)
>;

Mapping mapping(
    extents,
    treeOperationList
);

// or using CTAD and an unused argument for the record dimension:
llama::mapping::tree::Mapping mapping(
    extents,
    llama::Tuple{
        llama::mapping::tree::functor::LeafOnlyRT()
    },
    RecordDim{}
);
```

The following tree operations are defined:

7.4 Idem

`llama::mapping::tree::functor::Idem` does not change the tree at all. Basically a test functor for testing, how much the number of tree operations has an influence on the run time.

7.5 LeafOnlyRT

`llama::mapping::tree::functor::LeafOnlyRT` moves all run time parts of the tree to the leaves, basically creates a struct of array as seen above. However unlike `llama::mapping::SoA` a combination with other mapping would be possible.

7.6 MoveRTDown

`llama::mapping::tree::functor::MoveRTDown` moves a runtime multiplier from a node identified by a tree coordinate one level downward. This effectively divides the annotation at the node by a given factor and multiplies the direct child nodes by this factor.

7.7 MoveRTDownFixed

Same as MoveRTDown but with a compile time factor.

7.7.1 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);
```

This feature requires to have libfmt installed.

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 notrious 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 refered 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&& element : v)
   bool b = element;
```

Mind, that we explicitely state `bool` as the type of the resulting value. 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 access contained objects, LLAMA views are similar to standard C++ containers. For references to whole records, LLAMA views hand out virtual records. Although a virtual record 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 virtual record in LLAMA is thus conceptually a proxy reference. An exception is however made for read/write access in the current API, which is governed by the `load()` and `store()` member functions. We might change this in the future.

```
auto view = llama::allocView(...);
auto vr = view(1, 2, 3); // vr is a VirtualRecord, a proxy reference
Pixel p = vr.load(); // read access
vr.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 = vr(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 virtual record, 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.
- Explicitely specify the type of copies of individual fields you want to make from references obtains from a LLAMA view. This avoids accidentially coping a proxy reference.

8.4 Concept

Proxy references in LLAMA fullfill the following concept:

```
template <typename R>
concept ProxyReference = 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&>;
};
```

That is, the provide a member type `value_type`, which indicates the type of the values which can be loaded and stored through the proxy reference. Furthmore, a proxy reference can be converted to its value type (thus calling `operator value_type ()`) or assigned an instance of its value type.

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`.

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, Alignment>{}, size);
```

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

9.1.1 Shared memory

`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.

9.1.2 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.

9.1.3 Stack

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::Stack` 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<4, 4>;
constexpr ArrayExtents extents{};

using Mapping = /* a simple mapping */;
auto blobAllocator = llama::bloballoc::Stack<
    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::Stack<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::allocViewStack<N, RecordDim>();
```

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

The following descriptions are for alpaka users. Without an understanding of alpaka they may hard to understand.

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. However, a pointer to the underlying storage can be obtained, which may be used for a view:

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

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]}};
```

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 lstinline{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 bluring 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␣
↪LinearizeArrayDims>
struct Copy<
    llama::mapping::AoS<ArrayExtents, RecordDim, Aligned, LinearizeArrayDims>,
    UserDefinedMapping<ArrayExtents, RecordDim>>
{
    template <typename SrcBlob, typename DstBlob>
    void operator()(
        const View<mapping::AoS<ArrayExtents, RecordDim, Aligned, LinearizeArrayDims>,␣
↪SrcBlob>& srcView,
        View<mapping::SoA<ArrayExtents, RecordDim, DstSeparateBuffers,␣
↪LinearizeArrayDims>, 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

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.

11.1 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`.

11.2 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

TWELVE

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.

12.1 Useful helpers

struct **NrAndOffset**

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

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.

12.1.1 Array

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

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

tparam T type if array elements.

tparam N rank of the array.

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

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

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

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

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

12.1.2 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::`**pop_front**(const *Tuple*<*Elements*...> &tuple)
Returns a copy of the tuple without the first element.

12.2 Array dimensions

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

struct **ArrayExtents** : public llama::*Array*<*ArrayIndex*<sizeof...(*Sizes*)>::value_type, ((*Sizes* == dyn) + ... + 0)>
ArrayExtents holding compile and runtime indices. This is conceptually equivalent to the std::extent of std::mdspan. See: https://wg21.link/P0009

Subclassed by *llama::ArrayIndexRange< ArrayExtents >*

using `llama::`**ArrayExtentsDynamic** =
internal::mp_unwrap_values_into<boost::mp11::mp_repeat_c<boost::mp11::mp_list_c<std::size_t, dyn>, N>, *ArrayExtents*>
N-dimensional *ArrayExtents* where all values are dynamic.

using `llama::`**ArrayExtentsStatic** =
internal::mp_unwrap_values_into<boost::mp11::mp_repeat_c<boost::mp11::mp_list_c<std::size_t, Extent>, N>, *ArrayExtents*>
N-dimensional *ArrayExtents* where all values are Extent.

template<std::size_t **Dim**>

struct **ArrayIndex** : public llama::*Array*<std::size_t, *Dim*>
Represents a run-time index into the array dimensions.

tparam Dim Compile-time number of dimensions.

template<typename **ArrayExtents**>

struct **ArrayIndexIterator**
Iterator supporting *ArrayIndexRange*.

template<typename **ArrayExtents**>

struct **ArrayIndexRange** : private llama::*ArrayExtents*<Sizes>
Range allowing to iterate over all indices in an *ArrayExtents*.

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

12.3 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*.

tparam Tag Name of the node. May be any type (struct, class).

tparam 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*.

using `llama::`**GetFieldTag** = boost::mp11::mp_first<*Field*>
Get the tag from a *Field*.

using `llama::`**GetFieldType** = boost::mp11::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.

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*.

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 **LocalA**, typename **RecordDimB**, typename **LocalB**>

constexpr auto `llama::`**hasSameTags** = `[]() constexpr{if constexpr(LocalA::size != LocalB::size)return false;else if constexpr(LocalA::size == 0 && LocalB::size == 0)return true;elsereturn std::is_same_v<GetTags<RecordDimA, LocalA>, GetTags<RecordDimB, LocalB>>;}()`
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.
- **LocalA** – *RecordCoord* based on StartA along which the tags are compared.
- **RecordDimB** – second record dimension.
- **LocalB** – *RecordCoord* based on StartB along which the tags are compared.

using `llama::`**GetCoordFromTags** = typename internal::GetCoordFromTagsImpl<RecordDim, *RecordCoord*<>, Tags...>::type
Converts a series of tags, or a list of tags, navigating down a record dimension into a *RecordCoord*.

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.

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.

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.

using `llama::TransformLeaves` = typename internal::TransformLeavesImpl<RecordDim, FieldTypeFunctor>::type

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.

llama::MergedRecordDims = typename decltype(internal::mergeRecordDimsImpl(boost::mp11::mp_identity< Rec
boost::mp11::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**>
auto `llama::recordCoordTags`(*RecordCoord*<*Coords*...>) -> std::string

Returns the tags interspersed by '.' represented by the given record coord in the given record dimension.

12.4 Record coordinates

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

struct `llama::RecordCoord`

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

tparam Coords... the compile time coordinate.

Public Types

using **List** = boost::mp11::mp_list_c<std::size_t, *Coords*...>

The list of integral coordinates as `boost::mp11::mp_list`.

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

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

using `llama::Cat` = *RecordCoordFromList*<boost::mp11::mp_append<typename RecordCoords::List...>>
Concatenate a set of *RecordCoord*s.

using `llama::PopFront` = *RecordCoordFromList*<boost::mp11::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.

12.5 View creation

template<typename **Mapping**, typename **Allocator** = bloballoc::*Vector*>
inline auto `llama::allocView`(*Mapping* mapping = {}, const *Allocator* &alloc = {}) -> *View*<*Mapping*, internal::AllocatorBlobType<*Allocator*, typename *Mapping*::RecordDim>>
Creates a view based on the given mapping, e.g. AoS or :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. The constructors are run for all fields by calling *constructFields*. This function is the preferred way to create a *View*. See also *allocViewUninitialized*.

template<typename **Mapping**, typename **BlobType**>
inline void `llama::constructFields`(*View*<*Mapping*, *BlobType*> &view)
Runs the constructor of all fields reachable through the given view. Computed fields are not constructed.

template<typename **Mapping**, typename **Allocator** = bloballoc::*Vector*>
inline auto `llama::allocViewUninitialized`(*Mapping* mapping = {}, const *Allocator* &alloc = {}) -> *View*<*Mapping*, internal::AllocatorBlobType<*Allocator*, typename *Mapping*::RecordDim>>
Same as *allocView* but does not run field constructors.

template<std::size_t **Dim**, typename **RecordDim**>
inline auto `llama::allocViewStack`() -> decltype(auto)
Allocates a *View* holding a single record backed by stack memory (*bloballoc::Stack*).

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

using `llama::One` = *VirtualRecord*<decltype(*allocViewStack*<0, RecordDim>()), *RecordCoord*<>, true>
A *VirtualRecord* that owns and holds a single value.

template<typename **VirtualRecord**>
inline auto `llama::copyVirtualRecordStack`(const *VirtualRecord* &vd) -> decltype(auto)
Creates a single *VirtualRecord* owning a view with stack memory and copies all values from an existing *VirtualRecord*.

12.5.1 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*.

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

struct **Stack**
Allocates stack memory for a *View*, which is copied each time a *View* is copied.

tparam BytesToReserve the amount of memory to reserve.

12.6 Mappings

using `llama::mapping::`**AlignedAoS** = *AoS*<*ArrayExtents*, RecordDim, true, LinearizeArrayDimsFunctor>
Array of struct mapping preserving the alignment of the field types by inserting padding.

See *AoS*

using `llama::mapping::`**PackedAoS** = *AoS*<*ArrayExtents*, RecordDim, false, LinearizeArrayDimsFunctor>
Array of struct mapping packing the field types tightly, violating the types alignment requirements.

See *AoS*

template<typename **TArrayExtents**, typename **TRecordDim**, bool **AlignAndPad** = true, typename **TLinearizeArrayDimsFunctor** = *LinearizeArrayDimsCpp*, template<typename> typename **FlattenRecordDim** = *FlattenRecordDimInOrder*>

struct **AoS** : private *TArrayExtents*
Array of struct mapping. Used to create a *View* via *allocView*.

tparam AlignAndPad If true, padding bytes are inserted to guarantee that struct members are properly aligned. If false, struct members are tightly packed.

tparam T_LinearizeArrayDimsFunctor Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.

tparam FlattenRecordDim Defines how the record dimension's fields should be flattened. See *FlattenRecordDimInOrder*, *FlattenRecordDimIncreasingAlignment*, *FlattenRecordDimDecreasingAlignment* and *FlattenRecordDimMinimizePadding*.

using `llama::mapping::`**SingleBlobSoA** = *SoA*<*ArrayExtents*, RecordDim, false, LinearizeArrayDimsFunctor>
Struct of array mapping storing the entire layout in a single blob.

See *SoA*

using `llama::mapping::`**MultiBlobSoA** = *SoA*<*ArrayExtents*, RecordDim, true, LinearizeArrayDimsFunctor>
Struct of array mapping storing each attribute of the record dimension in a separate blob.

See *SoA*

template<typename **TArrayExtents**, typename **TRecordDim**, bool **SeparateBuffers** = true, typename **TLinearizeArrayDimsFunctor** = *LinearizeArrayDimsCpp*, template<typename> typename **FlattenRecordDimSingleBlob** = *FlattenRecordDimInOrder*>

struct **SoA** : private *TArrayExtents*

Struct of array mapping. Used to create a *View* via *allocView*.

tparam SeparateBuffers If true, every element of the record dimension is mapped to its own buffer.

tparam LinearizeArrayDimsFunctor Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.

tparam FlattenRecordDim Defines how the record dimension's fields should be flattened if SeparateBuffers is false. See *FlattenRecordDimInOrder*, *FlattenRecordDimIncreasingAlignment*, *FlattenRecordDimDecreasingAlignment* and *FlattenRecordDimMinimizePadding*.

template<typename **TArrayExtents**, typename **TRecordDim**, bool **AlignAndPad** = true, template<typename> typename **FlattenRecordDim** = *FlattenRecordDimMinimizePadding*>

struct **One** : public *TArrayExtents*

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

tparam AlignAndPad If true, padding bytes are inserted to guarantee that struct members are properly aligned. If false, struct members are tightly packed.

tparam FlattenRecordDim Defines how the record dimension's fields should be flattened. See *FlattenRecordDimInOrder*, *FlattenRecordDimIncreasingAlignment*, *FlattenRecordDimDecreasingAlignment* and *FlattenRecordDimMinimizePadding*.

template<typename **TArrayExtents**, typename **TRecordDim**, std::size_t **Lanes**, typename **TLinearizeArrayDimsFunctor** = *LinearizeArrayDimsCpp*, template<typename> typename **FlattenRecordDim** = *FlattenRecordDimInOrder*>

struct **AoSoA** : private *TArrayExtents*

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

tparam Lanes The size of the inner arrays of this array of struct of arrays.

tparam FlattenRecordDim Defines how the record dimension's fields should be flattened. See *FlattenRecordDimInOrder*, *FlattenRecordDimIncreasingAlignment*, *FlattenRecordDimDecreasingAlignment* and *FlattenRecordDimMinimizePadding*.

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

constexpr std::size_t `llama::mapping::`**maxLanes** = `[]() constexpr{auto max = std::numeric_limits<std::size_t>::max();forEachLeafCoord<RecordDim>([&](auto rc){using AttributeType =GetType<RecordDim, decltype(rc)>;max = std::min(max, VectorRegisterBits / (sizeof(AttributeType) * CHAR_BIT));});return max;}()`

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 **RecordCoordForMapping1**, 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.

tparam RecordCoordForMapping1 A *RecordCoord* or a list of RecordCoords selecting the part of the record dimension to be mapped differently.

tparam MappingTemplate1 The mapping used for the selected part of the record dimension.

tparam MappingTemplate2 The mapping used for the not selected part of the record dimension.

tparam SeparateBlobs If true, both pieces of the record dimension are mapped to separate blobs.

template<typename **Mapping**>

struct **Trace**

Forwards all calls to the inner mapping. Traces all accesses made through this mapping and prints a summary on destruction.

tparam Mapping The type of the inner mapping.

template<typename **Mapping**, typename **CountType** = std::size_t>

struct **Heatmap**

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

tparam Mapping The type of the inner mapping.

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

struct `llama::mapping::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 **TArrayExtents**, typename **TRecordDim**, template<typename, typename> typename **InnerMapping**, typename **ReplacementMap**>

struct **ChangeType** : private *InnerMapping*<*TArrayExtents*, internal::ReplaceType<*TRecordDim*, *ReplacementMap*>>

Mapping that changes the type in the record domain for a different one in storage. Conversions happen during load and store. /tparam ReplacementMap A type list of binary type lists (a map) specifiying which type to replace by which other type.

template<typename **TArrayExtents**, typename **TRecordDim**, typename **Bits** = unsigned, typename **LinearizeArrayDimsFunctor** = *LinearizeArrayDimsCpp*, typename **StoredIntegral** = internal::StoredUnsignedFor<*TRecordDim*>>

struct **BitPackedIntSoA** : public *TArrayExtents*, private llama::internal::BoxedValue<*Bits*>

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.

tparam 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.

tparam LinearizeArrayDimsFunctor Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.

tparam StoredIntegral Integral type used as storage of reduced precision integers.

template<typename **TArrayExtents**, typename **TRecordDim**, typename **ExponentBits** = unsigned, typename **MantissaBits** = unsigned, typename **LinearizeArrayDimsFunctor** = *LinearizeArrayDimsCpp*, typename **StoredIntegral** = internal::StoredIntegralFor<*TRecordDim*>>

struct **BitPackedFloatSoA** : public *TArrayExtents*, public llama::internal::BoxedValue<*ExponentBits*, 0>, public llama::internal::BoxedValue<*MantissaBits*, 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.

tparam 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.

tparam MantissaBits Like ExponentBits but for the mantissa bits.

tparam LinearizeArrayDimsFunctor Defines how the array dimensions should be mapped into linear numbers and how big the linear domain gets.

tparam StoredIntegral Integral type used as storage of reduced precision floating-point values.

12.6.1 RecordDim flattener

template<typename **RecordDim**>

struct **FlattenRecordDimInOrder**

Flattens the record dimension in the order fields are written.

template<typename **RecordDim**, template<typename, typename> typename **Less**>

struct **FlattenRecordDimSorted**

Flattens the record dimension by sorting the fields according to a given predicate on the field types.

tparam 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.

using `llama::mapping::`**FlattenRecordDimIncreasingAlignment** = *FlattenRecordDimSorted*<RecordDim, internal::LessAlignment>

Flattens and sorts the record dimension by increasing alignment of its fields.

using `llama::mapping::`**FlattenRecordDimDecreasingAlignment** = *FlattenRecordDimSorted*<RecordDim, internal::MoreAlignment>

Flattens and sorts the record dimension by decreasing alignment of its fields.

using `llama::mapping::`**FlattenRecordDimMinimizePadding** = *FlattenRecordDimIncreasingAlignment*<RecordDim>

Flattens and sorts the record dimension by the alignment of its fields to minimize padding.

12.6.2 Common utilities

struct `llama::mapping::LinearizeArrayDimsCpp`
Functor that maps an *ArrayIndex* into linear numbers the way C++ 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[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 -> std::size_t

Parameters

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

Returns Linearized index.

struct `llama::mapping::LinearizeArrayDimsFortran`
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[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 -> std::size_t

Parameters

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

Returns Linearized index.

struct `llama::mapping::LinearizeArrayDimsMorton`
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 -> std::size_t

Parameters

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

Returns Linearized index.

12.6.3 Tree mapping (deprecated)

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

struct **Mapping** : private *TArrayExtents*

An experimental attempt to provide a general purpose description of a mapping. *Array* and record dimensions are represented by a compile time tree data structure. This tree is mapped into memory by means of a breadth-first tree traversal. By specifying additional tree operations, the tree can be modified at compile time before being mapped to memory.

For a detailed description of the tree mapping concept have a look at LLAMA tree mapping

Tree mapping functors

struct **Idem**

Functor for *tree::Mapping*. Does nothing with the mapping tree. Is used for testing.

struct **LeafOnlyRT**

Functor for *tree::Mapping*. Moves all run time parts to the leaves, creating a *SoA* layout.

template<typename **TreeCoord**, typename **Amount** = std::size_t>

struct **MoveRTDown**

Functor for *tree::Mapping*. Move the run time part of a node one level down in direction of the leaves by the given amount (runtime or compile time value).

See *tree::Mapping*

tparam TreeCoord tree coordinate in the mapping tree which's run time part shall be moved down one level

12.6.4 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

12.7 Data access

template<typename **TMapping**, typename **BlobType**>

struct `llama::View` : private *TMapping*

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*.

tparam TMapping The mapping used by the view to map accesses into memory.

tparam BlobType The storage type used by the view holding memory.

Public Functions

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

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

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

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

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

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

inline auto **operator[]**(std::size_t index) const -> decltype(auto)

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

template<typename **TStoredParentView**>

struct `llama::VirtualView`

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

tparam 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

using **Mapping** = typename *ParentView*::Mapping

mapping of the parent view

using **ArrayExtents** = typename *Mapping*::ArrayExtents

array extents of the parent view

using **ArrayIndex** = typename *Mapping*::ArrayIndex

array index of the parent view

Public Functions

template<typename **StoredParentViewFwd**>
inline **VirtualView**(*StoredParentViewFwd* &&parentView, *ArrayIndex* offset)
Creates a *VirtualView* 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 *VirtualView*.

template<typename ...**Indices**>
inline auto **operator()**(*Indices*... indices) const -> decltype(auto)
Same as corresponding operator in *View*, but shifted by the offset of this *VirtualView*.

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 `llama::VirtualRecord` : private *ArrayIndex*
Virtual record type returned by *View* after resolving an array dimensions coordinate or partially resolving a *RecordCoord*. A virtual record does not hold data itself (thus named "virtual"), it just binds enough information (array dimensions coord and partial record coord) to retrieve it from a *View* later. Virtual records should not be created by the user. They are returned from various access functions in *View* and *VirtualRecord* itself.

Public Types

using **View** = *TView*
View this virtual record points into.

using **BoundRecordCoord** = *TBoundRecordCoord*
Record coords into View::RecordDim which are already bound by this *VirtualRecord*.

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

template<typename **OtherView**, typename **OtherBoundRecordCoord**, bool **OtherOwnView**>
inline **VirtualRecord**(const *VirtualRecord*<*OtherView*, *OtherBoundRecordCoord*, *OtherOwnView*>
&virtualRecord)
Create a VirtuaRecord from a different *VirtualRecord*. Only available for if the view is owned. Used by *llama::One*.

template<typename **T**, typename = std::enable_if_t<!is_VirtualRecord<*T*>>>
inline explicit **VirtualRecord**(const *T* &scalar)
Create a VirtuaRecord 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 virtual record using a *RecordCoord*. If the access resolves to a leaf, a reference to a variable inside the *View* storage is returned, otherwise another virtual record.

template<typename ...**Tags**>
inline auto **operator()**(*Tags*...) const -> decltype(auto)
Access a record in the record dimension underneath the current virtual record using a series of tags. If the access resolves to a leaf, a reference to a variable inside the *View* storage is returned, otherwise another virtual record.

struct **Loader**

struct **LoaderConst**

12.8 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 view to destination view. Both views need to have the same array and record dimensions. 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, bool readOpt, 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.

12.9 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_FORCE_INLINE_RECURSIVE

Forces the compiler to recursively inline the call hiearchy started by the subsequent function call.

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.

INDEX

L