Updates on the Low-Level Abstraction of Memory Access library

Bernhard Manfred Gruber (CERN, CASUS, HZDR, TU Dresden)

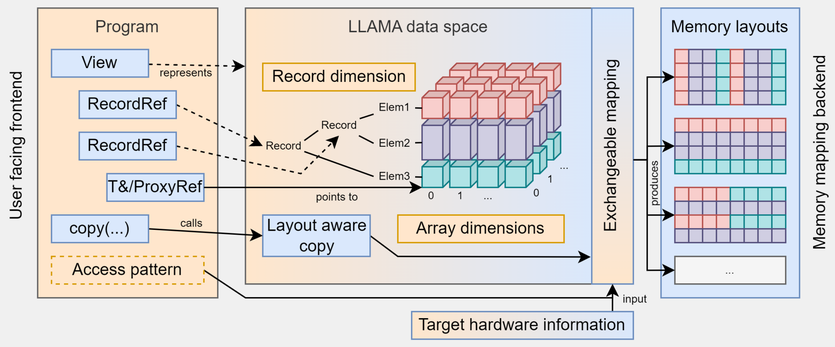
The library

LLAMA is a header-only library for data layout abstraction. It is written in portable, standard C++17. It is designed to integrate well with CUDA, HIP, alpaka, Kokkos, SYCL, SIMD libraries, ..., but stays orthogonal and independent.

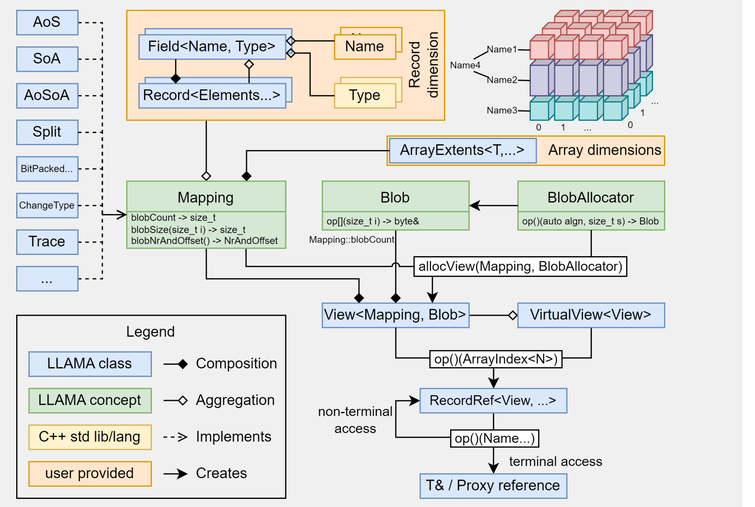
Motivation: The performance gap between CPU and memory widens continuously. Many programs nowadays are memory-bound. Compute and memory hardware is increasingly heterogeneous. Writing portable and performant programs becomes harder. Memory related optimizations typically depend on full control over data layout. There is no fully generic solution to zero-overhead memory layout abstraction (yet). LLAMA tries to fill this gap.

Goals: LLAMA aims to express generic data structures, allow any user-defined mapping of this data structure to memory and augment these mappings with hardware and access pattern information. Memory mappings can be exchanged without touching the algorithm. LLAMA should provide efficient copy routines between different memory layouts and support CPU, GPU and FPGAs.

Find us on GitHub: https://github.com/alpaka-group/llama
Check out our paper: "LLAMA: The low-level abstraction for memory access", by Bernhard Manfred Gruber, Guilherme Amadio, Jakob Blomer, Alexander Matthes, René Widera and Michael Bussmann @ https://doi.org/10.1002/spe.3077



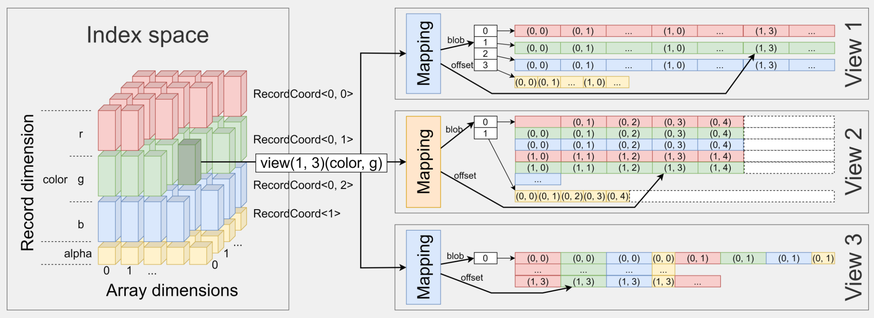
Conceptually, LLAMA uses a record dimension and 0-n array dimensions on span a data space of objects which should be mapped to memory. A user's program interacts with this data space via a View, with individual records via RecordRef and with the final objects via l-value references or proxy references. The data space is mapped using an exchangeable and user-definable mapping into a memory layout. This mapping can be augmented with information on target hardware and access pattern. LLAMA also supports layout aware copy operations.

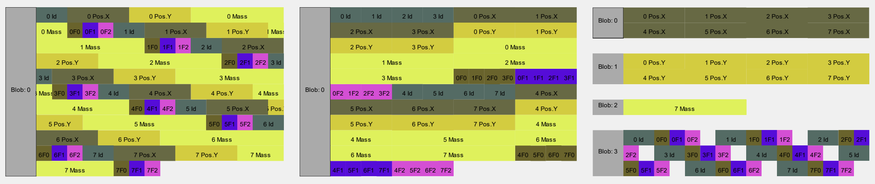


The above figure gives an overview of the main parts of the C++ library.

Mappings

A LLAMA mapping determines the number of blobs (flat byte arrays) and their sizes to provide storage for a data space described via record and array dimensions. (Physical) mappings transform an index tuple of access information into a memory location. The index tuple is composed of a runtime array index and a compile-time tags from the record dimension. The mapping resolves this into a blob index and offset. The view uses these two indices to retrieve the final memory location from its array of byte blobs.





Above are three examples of different mappings of the same data structure. Left: A packed AoS mapping. Middle: An AoSoA4 mapping. Right: Split mapping, where the field at record coordinate 1 (Pos) is split off into a multiblob SoA, followed by splitting the new record coordinate 1 (Mass) into the mapping One and layouting the remaining record (Id and Flags) as aligned AoS. All figures were dumped by LLAMA. Below: the corresponding LLAMA mapping definitions.

```
using Vec2 = llama::Record<
  llama::Field<X, float>,
  llama::Field<Y, float>
>;

using Particle2 = llama::Record<
  llama::Field<Id, uint16_t>,
  llama::Field<Pos, Vec2>,
  llama::Field<Mass, double>,
  llama::Field<Flags, bool[3]>
>;
```

```
using MapLeft =
  llama::mapping::AoS<ArrayExtents, Particle2>;
using MapMid =
  llama::mapping::AoSoA<ArrayExtents, Particle2, 4>;
using MapRight = llama::mapping::Split<
  ArrayExtents, Particle2, llama::RecordCoord<1>,
  llama::mapping::BindSoA<true>::fn,
  llama::mapping::BindSplit<llama::RecordCoord<1>,
    llama::mapping::PackedOne,
    llama::mapping::AlignedAoS, true>::fn,
  true>
```

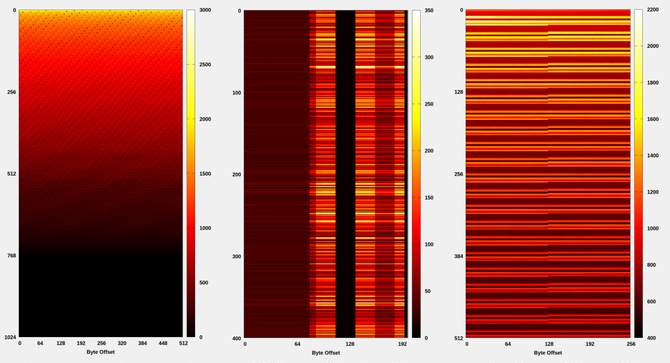
New: Array dimensions can specify the index type to use in all LLAMA computations (e.g., int or size_t). This is relevant for some architectures, like CUDA or FPGAs. And array dimensions can be (partially) specified at compile time now; only runtime extents are stored. If all extents are provided at compile time, the array extents and mappings become stateless. Combined with the Stack blob allocator the view becomes a trivial value type and contains only the mapped data. It now can be e.g. memcpy-ed or placed in CUDA __shared__ memory.

```
auto ae1 = llama::ArrayExtentsDynamic<int, 2>{size1, size2};
auto ae2 = llama::ArrayExtents<std::size_t, 3, llama::dyn, 4, 4>{size};
auto ae3 = llama::ArrayExtents<short, 32, 4, 4>{};
```

LLAMA provides meta mappings for software instrumentation. It can either count the total number of reads/writes per record field (lightweight) or provide a memory heatmap (heavyweight).

Counting is performed as side effect of data access. It costs one atomic inc. per access. We measured, e.g., a ~3x slowdown in a CUDA simulation.

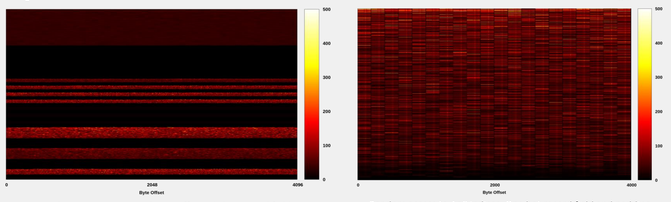
Limitations: We cannot observe what the hardware does. E.g., whether a memory read is served from VRAM or cache. We cannot observe what the compiler/optimizer does. E.g., whether a second memory read to the same memory location is optimized away. Preliminary refactoring can help. E.g., replacing repeated access to memory by a local variable.



AoS with one hot record field and padding. Higher access counts towards buffer front.

AoS with uneven field and array index accesses.

AoSoA with uneven field access. GPU cachelines are still hit coalescingly.



Random access to SoA indices. Differently hot record fields.

Random access to AoS indices. One hot record field and padding.

All LLAMA built-in mappings and their customizations:

- **AoS**: Aligned/Packed, ND-array linearizers, struct member reordering
- **SoA**: Single/Multi blob, Aligned/Packed sub arrays, ND-array linearizers, struct member reordering
- **AoSoA**: Inner array size, ND-array linearizers, struct member reordering
- **BitPackFloatSoA**, **BitPackIntSoA**: Bit count for value/mantissa/exponent
- **ChangeType**: Replace record dim types for storage, forward to inner mapping
- **Bytesplit**: Split all types in byte arrays, then forward to inner mapping
- **Trace**: Trace access/read/write counts, then forward to inner mapping
- **Heatmap**: Count accesses per byte (or coarser), then forward to inner mapping
- **One**: Aligned/Packed, struct member reordering, Map all array indices to the same record instance
- **Null**: Read returns default constructed value, writes are discarded
- **Split**: Split record dimension in two, forward each part to inner mappings, leave or merge blobs of inner mappings

Example n-body simulation

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

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

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

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

void move(auto& particles) {
  LLAMA_INDEPENDENT_DATA
  for(std::size_t i = 0; i < problemSize; i++)
    particles(i)(tag::Pos{}) += particles(i)(tag::Vel{}) * timestep;
}

int main() {
  using ArrayExtents = llama::ArrayExtentsDynamic<std::size_t, 1>;
  using Mapping = llama::mapping::AoS<ArrayExtents, Particle>; // !!!

  auto mapping = Mapping{ArrayExtents{problemSize}};
  auto view = llama::allocViewUninitialized(mapping);

  for(auto&& p : view) {
    p(tag::Pos{}, tag::X{}) = random();
    // ...
    p(tag::Mass{}) = random();
  }

  for(std::size_t s = 0; s < steps; ++s) {
    update(view);
    move(view);
  }
}
```

Try LLAMA n-body yourself: https://godbolt.org/z/4fTjhWq6d

Benchmark in left bottom corner, SIMD version below.

SIMD

SIMD primarily concerns computation. The only interaction point with LLAMA are memory layout aware N-element vector load/store operations. For convenience, LLAMA also provides SIMD records. The API is independent of a SIMD library and integration is handled via traits.

A SIMD enabled program with LLAMA involves these steps:

1. Pick a SIMD library (e.g. xsimd, std::simd)
2. Specialize llama::SimdTraits
3. Choose a SIMD vector width N
4. Create local variables using SIMD-ized types
5. Load/store between LLAMA views and SIMD-zied variables
6. Navigate LLAMA-created SIMD types like llama::RecordRef
7. Express computations in accordance with your SIMD library

SimdN is LLAMA's construct to create SIMD-ized constructs for holding values. LLAMA can simdize scalar types or record dimensions to a specified N. MakeSizedSimd is a user provided type function which LLAMA uses to create SIMD types.

```
template<typename T, std::size_t N,
  template<typename, std::integral auto>
    typename MakeSizedSimd>
using SimdN = …;
```

SimdN<T, N, …>	N > 1	N == 1
Record dim T	One<SimdizeN<T, N, …>>	One<T>
scalar T	SimdizeN<T, N, …>	T

Example: SimdizeN creates SIMD-ized record dimension:

```
llama::SimdizeN<Vec2, 8, std::fixed_size_simd>
// aliases to:
llama::Record<
   llama::Field<X, std::fixed_size_simd<float, 8>>,
   llama::Field<Y, std::fixed_size_simd<float, 8>>>
```

N: SIMD code deals in vectors of N elements (aka. lane count). Loops stride with N, as each iteration processes N elements. N depends on data types and compilation flags/target hardware. E.g., int vs. double, AVX2 vs. NEON. Code needs to be written with a flexible N in mind. Within one compilation, only data types are relevant. Either, all involved data types use same N. E.g., int/float on AVX2 will use N = 8. Or, they require different N. E.g., int/double on AVX2 need N = 8 or 4. The latter case requires extra. LLAMA offers some constructs to help choosing N, but N must ultimately be supplied by the user.

LLAMA offers functions (below) to transfer data between a SIMD construct or scalar and a reference to memory. This reference may be an l-value or a RecordRef. In the latter case, LLAMA will handle the underlying memory layout transparently for the user.

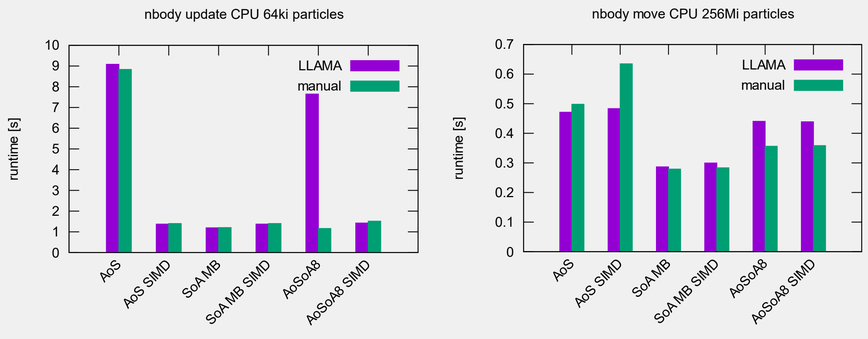
```
template<typename T, typename Simd>
void loadSimd(const T& srcRef, Simd& dstSimd);
template<typename Simd, typename T>
void storeSimd(const Simd& srcSimd, T&& dstRef);
```

Below is SIMD-ized version of the update routine of the n-body simulation above. For N > 1 and the right compiler flags, vectorized code is produced. For N = 1, a scalar version is generated without any trace of SIMD constructors. The scalar version runs successfully on CUDA.

```
template <int N>
void updateSimd(auto& particles) {
  using RecordDim =
    typename decltype(particles)::RecordDim;
  for(std::size_t i = 0; i < problemSize; i += N) {
    llama::SimdN<RecordDim, N, std::fixed_size_simd> pis;
    llama::loadSimd(particles(i), pis);
    for(std::size_t j = 0; j < problemSize; ++j)
      pPInteraction(pis, particles(j));
    llama::storeSimd(pis(tag::Vel{})
                     particles(i)(tag::Vel{}));
  }
}
```

Benchmarks

To the right we show a benchmark of the LLAMA n-body simulation for a CPU. The update step of n-body is usually compute-bound whereas the move step is memory-bound. We compare the LLAMA version against manually implemented codes.
The main take-away is that LLAMA delivers almost the same performance characteristics as the respective hand-written memory layout (except for the AoSoA). The abstraction layer thus generally fulfills the zero-overhead principle.
For SoA, we use the MB (multi-blob) version, which stores each field in a separate allocation.
The benchmark CPU was done on an AMD Ryzen 9 5950X with AVX2. Only single threaded results are shown to emphasize the efficiency of the generated instructions.



The AoSoA performs worse with LLAMA, because the client code uses a single for loop to traverse the array index space (see n-body example). The manual AoSoA version can use two nested for loops, which the compiler can unroll and vectorize. To allow for the same optimization in LLAMA, we would need to provide a foreach construct that uses the right, mapping aware loop structure. Or compilers need to get smarter. We are working on it.

Future work

- More elaborate tracing of memory access pattern.
  E.g., which part of the data structure is hot at which time/stage of the kernel.
- Better visualization of large memory traces.
  E.g., how to show a byte-wise trace on a 10GiB buffer on 1 screen?
- Support self-referential data structures (pointers)
- Formal approach to mappings
- Support more access patterns. LLAMA views are random access. We could do better if we knew that e.g., sequential iteration is intended. This could also solve the slow non-SIMD AoSoA.
- GPU/CUDA: Explore mixing global/shared memory behind a single view
- Layout aware copying between host and device (involving driver calls)
- Dynamic subarrays aka. jagged/awkward arrays
- Compressed blobs

This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA)