Updates on the Low-Level Abstraction of Memory Access

Bernhard Manfred Gruber

EP-SFT, CERN, Geneva, Switzerland
Center for Advanced Systems Understanding (CASUS), Saxony, Germany
Helmholz-Zentrum Dresden-Rossendorf (HZDR), Dresden, Germany
Faculty of Computer Science, Technische Universität Dresden, Dresden, Germany

E-mail: `bernhard.manfred.gruber@cern.ch`

Abstract. Choosing the best memory layout for each hardware architecture is increasingly important as more and more programs become memory bound. For portable codes that run across heterogeneous hardware architectures, the choice of the memory layout for data structures is ideally decoupled from the rest of a program. The low-level abstraction of memory access (LLAMA) is a C++ library that provides a zero-runtime-overhead abstraction layer, underneath which memory mappings can be freely exchanged to customize data layouts, memory access and access instrumentation, focusing on multidimensional arrays of nested, structured data. After its scientific debut, several improvements and extensions have been added to LLAMA. This includes compile-time array extents for zero-memory-overhead views, support for computations during memory access, new mappings for bit-packing, switching types, byte-splitting, memory access instrumentation, and explicit SIMD support. This contribution provides an overview of recent developments in the LLAMA library.

1. Introduction

The performance gap between CPU and memory widens continuously – many programs nowadays are memory-bound. Compute and memory hardware is increasingly heterogeneous and writing portable and performant programs becomes harder. Memory-related optimizations typically depend on full control over data layout and memory access. The Low-Level Abstraction of Memory Access (LLAMA) is being developed as a portable, standard C++17/C++20 library to fill this gap [1]. At its core, LLAMA separates the algorithmic view of data from its mapping to memory, allowing different data layouts to be chosen without touching the algorithm.

Conceptually, LLAMA uses a record dimension and several array dimensions to 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(a user-defined type that acts like a language built-in reference). The data space is mapped to a memory layout using an exchangeable and user-definable mapping. This mapping can be augmented with information on target hardware and access pattern. LLAMA also supports layout-aware copy operations. Despite these many capabilities, further extensions to LLAMA were necessary and have been developed since its first publication [2]. In this article, we would like to present these recently introduced features and discuss their applications and use cases.

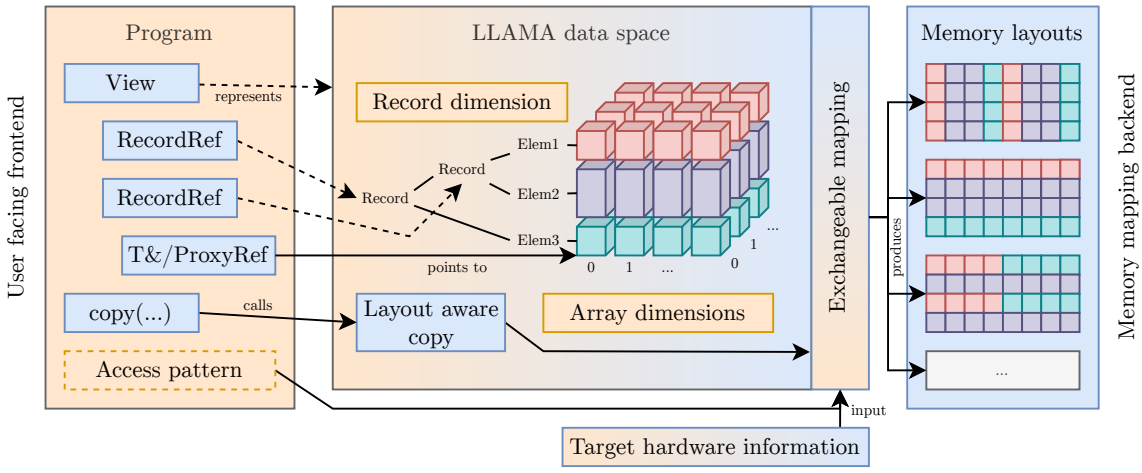

Figure 1. Conceptual overview of LLAMA.

2. Compile-time array extents

Previously, the number of array dimensions in LLAMA were specified at compile-time, but the extents of each dimension were strictly runtime values. Also, all indexing and memory offset calculations used the `std::size_t` data type, which commonly has a 64-bit representation on many systems today. This suffices for common CPUs and big LLAMA views, where the array extents stored inside the view are negligible. However, GPUs can incur higher costs for 64-bit integer arithmetic compared to 32-bit. For example, dedicated 64-bit integer hardware is absent from CUDA architectures like Hopper [3] and the CUDA programming guide [4]. And while the AMD MI200 architecture supports some 64-bit integer instructions, it notable lacks arithmetic ones [5]. Therefore, using smaller integral types is desirable. Furthermore, small LLAMA views, e.g., placed in a GPU's shared memory, cannot afford to additionally store the view's array extents. Additionally, such extents are often derived from hardware quantities, e.g., the shared memory size per streaming multiprocessor, and known at compile time.

To account for these needs, LLAMA now allows to specify the data type which should be used in all indexing computations. Additionally, the array extents can be (partially) specified at compile time and only runtime extents are stored. If all extents are provided at compile time, array extents and mappings become stateless, which is achieved by a careful implementation and use of empty-base-class-optimization [6]. Combined with the right blob allocator, the view becomes a trivial value type and contains only the blobs for the mapped data. The view is thus trivially constructible and storage-wise equivalent to the mapped data. It can thus be memcpy-ed, reinterpret-casted from a buffer, or placed in, e.g., CUDA shared memory. Here are examples of the new array extents API:

```
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>{};
```

The definition `ae1` defines array extents with two dynamic sizes, using `int` as index type. Then, `ae2` defines array extents with one static extent of 3, a dynamic extent, and two more static extents of 4 each, using `std::size_t` as index type. Finally, `ae3` defines fully static/compile-time extents und uses `short` for all index arithmetic. Allowing this mixing of compile and runtime extents was inspired by recent changes to the C++23 proposal `std::mdspan` [7].

3. New memory mappings

Uses of LLAMA in real-world code bases and new environments led to the creation of new memory mappings, lifted into the LLAMA library for general use. In the following we would like to present the newly added mappings and their intended area of application:

BitpackIntSoA and BitpackFloatSoA Experimental data in high-energy physics is often taken using specialized hardware with a precision different than the C++ standard fundamental types. Storing such values in the next bigger fundamental type wastes unnecessary bits of storage space but loads and stores use fast conventional hardware instructions. The BitpackIntSoA mapping allows to specify a desired bit count for integral types. The values will be packed/unpacked when stored to/loaded from memory. The bitpacked values are then further organized as Struct of Arrays (SoA), but we want to generalize this aspect in the future. The BitpackFloatSoA allows the user to individually specify the desired bit count for mantissa and exponent of floating-point values in the record dimension. Floating-point semantics are preserved as best as possible, including handling of NAN and INF values and mapping overflowing values during packing to INF.

Changetype Because of the packing/unpacking overhead required by the bitpack mappings, a mere change of the storage data type is computationally more efficient, because the hardware may have appropriate conversion instructions. Such a conversion could, e.g., map a `double` to a `float`, or even to one of the C++23 extended floating point types [8]: `float16`, `float32`, `float64`, `float128` or `bfloat16`. The adapted record dimension can then be mapped using a further mapping. This mapping was inspired from the accessor of Ginkgo [9].

Bytesplit Many compression algorithms are more efficient when compressing a stream of zeros. If the values in an integer array are small, the higher-order bytes may often be just zero. Splitting the values into their bytes and regrouping those by their order can effectively co-locate many zero-bytes and thus lead to higher compression ratios (cf. the `BYTE_STREAM_SPLIT` encoding in Apache Parquet). The Bytesplit mapping generalized this approach by splitting each type in the record dimension into a byte array of static size, and then forwarding the resulting record dimension to any further mapping.

Null Sometimes we do not need to store all the fields of a LLAMA view. An example is a view acting as a cache to a different view, e.g., in GPU shared memory, for a particular algorithm that only works on a subset of the record dimension. A different use case is to remove the effect of accessing a field when, e.g., profiling. The Null mapping discards any values written to it and returns a default constructed value when reading from it. It is intended to be used together with the Split mapping, to select which part of the record dimension to not map to physical storage.

FieldAccessCount and Heatmap These will be discussed in section 4.

While we have tested these new mappings in simple examples, we would like to properly explore the impact of these new memory mappings in a future publication.

4. Memory access instrumentation

LLAMA provides two instrumentation mappings, FieldAccessCount[1] and Heatmap. The lightweight FieldAccessCount counts the accumulated number of accesses per record field. The heavyweight Heatmap counts accesses to storage bytes at a configurable granularity such as bytes or cache lines. Both mappings forward all mapping logic to, and can thus instrument, an arbitrary inner mapping.

Counting memory accesses is performed as side effect of data access and costs one atomic increment to a dedicated memory location per regular access. For CUDA, we measured, e.g.,

[1] The FieldAccessCount mapping was called Trace in previous versions of LLAMA.

`SimdN<T, N, ...>`	`N > 1`	`N == 1`
Record dim `T`	`One<SimdizeN<T, N, ...>>`	`One<T>`
Scalar `T`	`SimdizeN<T, N, ...>`	`T`

Table 1. LLAMA's `SimdN` creates SIMD versions of scalar types or records, with a desired SIMD width N. For $N = 1$, scalar types and records are created. Otherwise, `SimdizeN` turns a scalar `T` into a SIMD vector of `T`, and a LLAMA record into a record with simdized field types.

a 3x slowdown in a particle transport simulation built with AdePT [10]. While the size of the extra memory to store the counters is negligible for the FieldAccessCount mapping (2 times the number of record fields), the Heatmap at highest granularity requires an extra counter per byte of memory. For a 64-bit (8 bytes) counter this results in an 8x memory overhead.

Software instrumentation, like discussed here, also comes with some limitations: LLAMA cannot observe what the hardware and the compiler/optimizer do. E.g., whether a memory read is served from RAM or cache, or whether a second read to the same address is optimized out and served from a register. Initial refactoring can help to increase the accuracy of instrumentation results, e.g., by replacing repeated access to memory by a local variable.

We extensively demonstrate LLAMA's instrumentation capabilities in our integration into the AdePT project, where we show tracing results and heatmaps of various memory access patterns of a particle transport simulation [11].

5. Explicit SIMD support

Programs using Single Instruction Multiple Data (SIMD) perform the same operation on N operands at the same time, typically supported by dedicated processor instruction sets. Automatic vectorization of scalar code to SIMD instructions by modern compilers is brittle and may fail for advanced codes, requiring the use of explicit SIMD APIs and dedicated libraries [12]. LLAMA has been extended to support such SIMD libraries.

The primary interaction between SIMD types and LLAMA are memory-layout-aware N-element vector load and store operations. Although many SIMD instructions also support memory operands in computational instructions, those can usually be produced by the compiler by fusing a load/store and a compute instruction. For convenience, LLAMA also provides simdized records, a term adopted from the Vc library [12] meaning the creation of a SIMD version of a type. Furthermore, load/store operations to handle scalar and simdized records uniformly have been added to LLAMA. The API is independent of a SIMD library and integration is handled via type traits (i.e., C++ template specialization for user-defined types).

LLAMA can simdize scalar types or record dimensions (structured data) to a specified N using the new `SimdN` API, as described in table 1. Algorithms should be written with a flexible N to be portable. However, N needs to be fixed at compilation by the user and depends on the target hardware, compilation flags and involved data types. While this handles well-established SIMD instruction sets like AVX or NEON, newer ones like ARM's Scalable Vector Extension have a runtime vector length. We have yet to see how to deal with such instruction sets. Beside `SimdN` to declare variables, LLAMA offers the `loadSimd` and `storeSimd` functions to transfer data between a SIMD construct or scalar and a reference to memory. LLAMA will handle records and the underlying memory layout transparently for the user. Figure 2 shows a simdized version of the update routine of the all-pairs n-body simulation from the original LLAMA paper [2]. With $N > 1$ and the right compiler flags, SIMD code is produced. For $N = 1$, a scalar version is generated without any trace of SIMD constructs. The scalar version can run on CUDA.

Figure 3 compares a LLAMA n-body simulation against manually written scalar and SIMD versions on an AMD Ryzen 9 5950X CPU with AVX2. Results are single-threaded to emphasize

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

Figure 2. A SIMD version of the n-body update routine from the original LLAMA paper [2], using `std::fixed_size_simd` as SIMD technology, as proposed for C++26 [13].

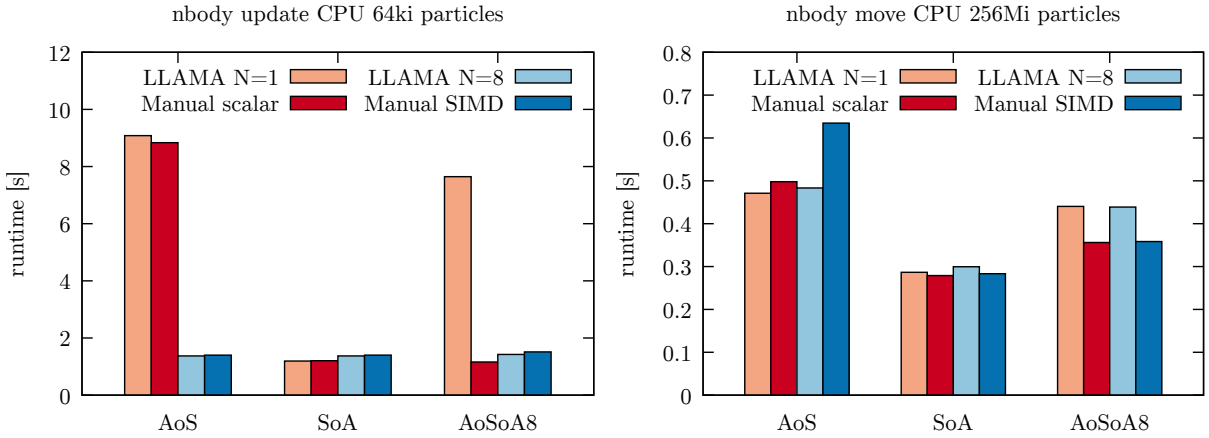

Figure 3. Benchmark of the CPU LLAMA n-body with a selection of popular mappings against various manually written versions.

the efficiency of the generated instructions. The scalar runs with the Arrays of Structs (AoS) layout are not auto-vectorized by the compiler. LLAMA matches the manually written code here. The manual SIMD implementation of the move step for the AoS layout uses gather instructions, whereas LLAMA uses multiple scalar loads, for which the compiler seemed to generate better code for the target CPU. Replacing the gather instructions in the manual SIMD move by multiple scalar loads, gets the runtime on par with the LLAMA SIMD runtime. For SoA, the multi-blob (MB) version is used, which stores each field in a separate allocation, and the results are nearly on-par between LLAMA and manually written versions. Both scalar codes have been auto-vectorized by the compiler. The Arrays of Structs of Arrays (AoSoA) layout in LLAMA has overhead in this example, which is especially visible in the n-body move phase and explained in more detail in the LLAMA paper [2]. It is caused by the LLAMA version using a single for-loop to traverse the array index space once, while the manual AoSoA version can use two nested for-loops (traversing AoSoA blocks and lanes), matching the memory layout structure and allowing easier auto-vectorization. Further investigation is pending to provide such mapping-aware loop structures inside LLAMA as well. Except for the AoSoA, LLAMA generally fulfills the zero-overhead principle in scalar and SIMD code.

6. Summary and outlook

We have presented recent updates and new features of LLAMA since its previous publication. Compile-time specification of array extents support common use cases for shared memory caches in GPGPU programming. New memory mappings open advanced possibilities for separating arithmetic from in-memory precision with various tradeoffs, supporting data arrangement for improved compression, and enabling instrumentation and tracing of memory access patterns. Finally, explicit SIMD support for LLAMA closes the gap between SIMD computing, structured data and arbitrary memory layouts.

In the future, we will focus on testing LLAMA with more real-world applications and on more hardware platforms. This will include the development of a systematic workflow to improve memory-related performance aspects of such applications. Supporting additional access patterns beyond LLAMA's random access could further solve the slow AoSoA and pave the way for LLAMA mappings with block compression algorithms.

Acknowledgments

This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA). The author would like to thank Guilherme Amadio, Verena Gruber and Stephan Hageböck for proof-reading and commentary.

References

[1] Gruber B M 2023 LLAMA – Low-Level Abstraction of Memory Access URL `https://github.com/alpaka-group/llama`

[2] Gruber B M, Amadio G, Blomer J, Matthes A, Widera R and Bussmann M 2023 LLAMA: The low-level abstraction for memory access *Software: Practice and Experience* **53** 115–141

[3] Andersch M, Palmer G, Krashinsky R, Stam N, Mehta V, Brito G and Ramaswamy S 2022 NVIDIA Hopper Architecture In-Depth URL `https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/`

[4] NVIDIA Corporation 2023 CUDA C++ Programming Guide URL `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`

[5] Advanced Micro Devices 2021 "AMD Instinct MI200" Instruction Set Architecture Reference Guide URL `https://developer.amd.com/wp-content/resources/CDNA2_Shader_ISA_18November2021.pdf`

[6] Meyers N 1997 The "Empty Member" C++ Optimization URL `http://www.cantrip.org/emptyopt.html`

[7] Trott C *et al.* 2022 P0009: MDSPAN Tech. rep. ISO JTC1/SC22/WG21 - Papers Mailing List URL `https://wg21.link/p0009r18`

[8] Olsen D, Burylov I and Dominiak M 2022 P1467: Extended floating-point types and standard names Tech. rep. ISO JTC1/SC22/WG21 - Papers Mailing List URL `https://wg21.link/p1467r9`

[9] Grützmacher T, Anzt H and Quintana-Ortí E S 2023 Using ginkgo's memory accessor for improving the accuracy of memory-bound low precision blas *Software: Practice and Experience* **53** 81–98

[10] Amadio G, Apostolakis J, Buncic P, Cosmo G, Dosaru D, Gheata A, Hageboeck S, Hahnfeld J, Hodgkinson M, Morgan B, Novak M, Petre A A, Pokorski W, Ribon A, Stewart G A and Vila P M 2023 Offloading electromagnetic shower transport to gpus *Journal of Physics: Conference Series* **2438** 012055 URL `https://dx.doi.org/10.1088/1742-6596/2438/1/012055`

[11] Gruber B M, Amadio G and Hageböck S 2023 Challenges and opportunities integrating LLAMA into AdePT 21st International Workshop on Advanced Computing and Analysis Techniques in Physics Research

[12] Kretz M and Lindenstruth V 2012 Vc: A c++ library for explicit vectorization *Software: Practice and Experience* **42** 1409–1430

[13] Kretz M 2023 P1928: Merge data -parallel types from the Parallelism TS 2 Tech. rep. ISO JTC1/SC22/WG21 - Papers Mailing List URL `https://wg21.link/p1928r2`