Performance Improvement of Multi-Dimensional Parallel Loops in Kokkos

Adrien Taberner*†, Hariprasad Kannan*†, Yuuichi Asahi‡, Dong Hun Lee§, Trévis Morvany‡, Jean-François David¶, Nevin Liber††, Nicolas Morales‖, Christian R Trott§, Julien Bigot‡, Damien Lebrun-Grandie**

* *Université Paris-Saclay, CEA*, *Service de Génie Logiciel pour la Simulation*, Gif-sur-Yvette, France

‡ *Université Paris-Saclay, UVSQ, CNRS, CEA*, *Maison de la Simulation*, Gif-sur-Yvette, France

§ *Sandia National Laboratories*, *Albuquerque, NM, USA*

¶ *Université Paris-Saclay, CEA*, *Institut de recherche sur les lois fondamentales de l'Univers*, Gif-sur-Yvette, France

†† *Argonne National Laboratory*, *Lemont, IL, USA*

‖ *Sandia National Laboratories*, *Livermore, CA, USA*

** *Los Alamos National Laboratory*, *Los Alamos, NM, USA*

† corresponding authors (adrien.taberner@cea.fr and hariprasad.kannan@cea.fr)

***Abstract*—This paper presents performance improvements for tightly nested multi-dimensional parallelism in the Kokkos library. We show how this can be implemented for GPU and CPU backends, which leads to improved performance and code maintainability. A widely used library like Kokkos needs to be updated in a manner that is friendly to the users and the developers. Using various benchmarks, we show how our work enables Kokkos users to write performance-portable code in a productive manner.**

***Index Terms*—Kokkos; parallel loops; Multi-dimensional parallel loops; Tiling; Performance portability and productivity; Performance optimization; GPUs; multi-core CPUs**

I. Introduction

Divergent architectures among exascale supercomputers require a performance-portable programming framework to develop applications. C++ based abstraction frameworks (e.g. Kokkos [1], [2], RAJA [3]) are widely used to achieve performance portability across different architectures including CPUs and GPUs. Kokkos attracts a large user base in the scientific computing community for two reasons. First, it has its own ecosystem [4], offering a range of libraries [5], [6] and compatible tools. Second, as a "core" project of the High Performance Software Foundation (HPSF) [7], [8], it offers open governance and long-term sustainability. Kokkos is widely used in scientific simulation codes including astrophysics [9], climate modeling [10], fluid simulations [11], and plasma turbulence [12].

In this work, we describe our experience improving the performance of a core component of the Kokkos library: the `MDRangePolicy` execution policy for tightly nested loops. Although it is a core component, it has not yet been well optimized for GPUs and CPUs, as Kokkos initially targeted molecular dynamics [13], [14] and unstructured mesh applications, which do not need it. Kokkos is increasingly used for a broad range of simulations, including those based on structured grids. Thanks to user feedback, `MDRangePolicy`'s suboptimal performance is a known issue in the community. For instance, AthenaK [9] deliberately avoids this policy and expresses tightly nested loops through a `RangePolicy`-based workaround. We therefore investigated how to improve the performance of `MDRangePolicy` for both GPU and CPU backends, with respect to Kokkos version 5.0. Our methodology for optimization is quite systematic and scalable from users' applications to the library itself. Our work provides insights into design, implementation and refactoring of performance-portable programming frameworks and will be useful to both users and contributors of Kokkos. The primary contributions of this work are as follows:

- A solution supporting efficient multi-dimensional parallel execution.
- Description of the key updates to Kokkos for `MDRangePolicy` implementation.
- Demonstration of how to optimize the library while making it transparent to the users, with no or minimal changes to the API.
- A set of benchmarks to demonstrate performance improvements.

II. Kokkos and MDRangePolicy

Kokkos offers performance portability through abstractions for parallel operations and data management. We will describe these abstractions, especially in the context of multi-dimensional parallelism.

A. Abstract memory management

In Kokkos, the `View` is the key data structure. It is a multi-dimensional array that can be allocated in different `Memory Spaces`, such as host, device, unified or scratch memory. The `Memory Layout` of the data can be specified, which is important for performance on different architectures. By default, Kokkos chooses `LayoutLeft` for GPUs and `LayoutRight` for CPUs. The former allows contiguous memory accesses of the left-most index on GPUs, whereas the latter makes the right-most index contiguous, improving

SC26 Workshops, November 15-20, 2026, Chicago, Illinois, USA

cache reuse on CPUs. An example of how to define a 2D `View` is shown in Listing 1 (line 1).

Listing 1. STREAM Scale operation using `Kokkos::parallel_for`

```
1 Kokkos::View<double**> a("A", N0, N1), b("B", N0, N1);
2
3 Kokkos::MDRangePolicy<Kokkos::Rank<2>>
4     policy_2d({0, 0}, {N0, N1});
5
6 Kokkos::parallel_for("Scale", policy_2d, KOKKOS_LAMBDA
7     (const int i, const int j) {
8         b(i, j) = scale * a(i, j);
9     });
```

`Kokkos::View` has been instrumental in the standardization of `mdspan` [15] in C++23. In Kokkos 5.0, `Kokkos::View` has been refactored to use `mdspan` as its underlying data structure, which enables better interoperability with other libraries and applications that use `mdspan`.

B. Abstract parallel operations

In scientific computing, parallel independent loops and parallel reductions are the most common operations. Kokkos provides abstractions for these through the functions, `parallel_for` and `parallel_reduce`, respectively. It also offers the `parallel_scan` function. These execution patterns let users express parallelism in a way that is suitable for a given architecture, which is specified via the execution space. All the patterns are executed with semantics comparable to `std::execution::par_unseq` for parallel algorithms in Standard C++. For `parallel_for` (example usage in Listing 1) this means that the computations across the iterations are executed independently and in any order.

An execution pattern is always driven by an execution policy. It defines the iteration space over which the pattern is executed, and how the resulting work is mapped onto the resources of an execution space. `RangePolicy` is an execution policy that targets a one-dimensional range of indices to iterate over. For multi-dimensional parallelism, Kokkos provides `TeamPolicy` and `MDRangePolicy`, which target non-tightly and tightly nested loops, respectively.

`TeamPolicy` is a hierarchical policy that specifies teams of threads executing in parallel, where each team runs a nested policy such as `TeamThreadRange` or `ThreadVectorRange`. It also gives access to scratch memory (shared memory on GPUs) and to team-level collectives such as barriers and reductions, which allow iterations to cooperate with each other. The multi-dimensional nested policies are described in Section II-C2.

`MDRangePolicy` executes computations on tightly nested iteration spaces with ranks (the dimensionality of the space) 1 through 6. The iteration space is partitioned into a tile grid, with each tile processed independently. In a grid of tiles, we have two notions of iteration order: (1) `outer_direction`: the iteration order over tiles. (2) `inner_direction`: the iteration order within a tile. For a given problem and backend, just like layout of the `Views`, iteration orders of the policies have to be properly chosen. It is important for iteration orders to match the layout of the `Views` involved. We can let Kokkos choose the iteration orders for a given backend. For example, the policy in Listing 1 (line 4) has `outer_direction = Iterate::Right` and `inner_direction = Iterate::Right` on CPU backends.

For `MDRangePolicy`, we also need to specify starting and ending indices of the right-open intervals that define the iteration space, as runtime arguments. We can optionally specify the tile sizes and when they are omitted, Kokkos selects default tile sizes for a given backend and rank. In Section III, we will discuss our improvements to the default tile sizes for GPU backends.

C. New features related to multi-dimensional parallelism

We briefly describe updates in Kokkos since version 3.0 [2], that are related to multi-dimensional parallelism.

1) Extended SIMD support: Modern CPUs contain wide vector execution units such as AVX-512 or SVE, which can operate on multiple elements in a single cycle, leading to significant throughput increase. `Kokkos::Experimental::simd` offers an abstraction over platform-specific SIMD intrinsics, allowing users to write portable SIMD code. This subpackage was added in Kokkos 3.7.0 [2]. It has since been largely refactored to align with the C++26 `std::simd` proposal [16].

Listing 2 shows the interface, where the SIMD vectors are defined in two different ways (line 2) and the square root of the sum of squares is computed (line 3). The `simd` class template takes a type and an optional platform-specific vector size as template parameters and provides a set of arithmetic and logical operators, as well as mathematical functions. It also provides reductions, load/store operations, and memory permutations in the same way as in C++26. The default `simd` type on device is a scalar type and thus enables the same code to also run on GPUs. We will further discuss the use of the SIMD API in Section IV-B.

Listing 2. `Kokkos::simd` example

```
1 using simd_type = Kokkos::Experimental::simd<double>;
2 simd_type a([] (int i) { return 0.1 * i; }), b(2.0);
3 simd_type c = Kokkos::sqrt(a * a + b * b);
```

2) Hierarchical MDRangePolicy: Users may parallelize tightly nested loops inside a team of threads via the multi-dimensional variants of the nested policies: `TeamThreadMDRange`, `TeamVectorMDRange` and `ThreadVectorMDRange`, which were introduced in Kokkos 4.0.0. Section III describes them and compares them with `MDRangePolicy` for GPU backends.

3) Support for one-dimensional iteration spaces: As part of this work, support for one-dimensional iteration spaces through `Kokkos::Rank<1>` was introduced in Kokkos 5.2.0. This allows uniform rank-parameterized downstream usage of `MDRangePolicy` from rank 1 to rank 6 while preserving backend-specific work distribution. This is preferable to falling back on `RangePolicy` for rank

1, since the two policies differ in their interfaces, including the `Kokkos::Rank<N>` template argument and constructor arguments. Also, for work-partitioning the tiles based of `MDRangePolicy` behaves the same way as `RangePolicy::chunk_size()`.

D. Other programming models

The treatment of tightly and non-tightly nested loops varies across programming models because they target different abstraction levels. We briefly review different approaches and compare them with Kokkos.

1) OpenMP and OpenACC: OpenMP and OpenACC provide dedicated directives to parallelize loop nests. For tightly nested loops, OpenMP/OpenACC provide the `collapse` clause to collapse the loops into a single loop. Also, they both support the notion of tiles. One key difference with respect to Kokkos is that the tile sizes are known at compile time. In Listing 3, OpenMP (OpenACC code looks very similar) executes in a tiled manner, with parallelism across the tiles. This is quite similar to the Kokkos based approach on multi-core CPUs. For non-tightly nested loops there are the teams and gangs based approaches in OpenMP Offload and OpenACC, respectively.

Listing 3. OpenMP example

```
#pragma omp parallel for collapse(2)
#pragma omp tile sizes(t0, t1)
for (int i = 0; i < N0; ++i)
  for (int j = 0; j < N1; ++j) {
    b[i, j] = scale * a[i, j];
  }
```

2) SYCL: Unlike OpenMP/OpenACC, SYCL does not parallelize source loops [17]. Instead, the programmer specifies an iteration space, as illustrated in Listing 4, and the runtime maps the iteration space to work-items.

Listing 4. SYCL example

```
sycl::nd_range<2>(
  sycl::range<2>(N0, N1), sycl::range<2>(16, 16)
)
```

SYCL distributes work using `nd_range` with two parameters, `global_range` and `local_range`, as shown in Listing 4. `global_range` defines the extent of the entire iteration space and `local_range` defines the extent of the work-group, which corresponds to the tile size in `MDRangePolicy`. Non-tightly nested loops can also be implemented in SYCL. One important difference is that SYCL offers up to 3D iteration spaces, whereas Kokkos supports up to 6D. In Section III-2, we describe how `MDRangePolicy` packs higher-dimensions into the three dimensions exposed by `sycl::nd_range` in the Kokkos SYCL backend [18].

3) C++ parallel algorithm: C++17 parallel algorithms are designed to work with standard containers and iterators, and they provide a simple and efficient way to express parallelism. To implement multi-dimensional parallelism, the standard C++ approach in Listing 5 uses `std::mdspan` [15] and `std::views::cartesian_product` [19] from C++23 and `std::views::indices` [20] from C++26. However, the standard does not support tiling and non-tightly nested loops.

Listing 5. Standard C++ example

```
std::mdspan a(ptr_a, N0, N1);
std::mdspan b(ptr_b, N1, N0);

std::ranges::for_each(
  std::execution::par_unseq,
  std::views::cartesian_product(
    std::views::indices(a.extent(0)),
    std::views::indices(a.extent(1))),
  [=] (auto idx) {
    auto [i, j] = idx;
    b[j, i] = a[i, j];
  });
```

In summary, Kokkos provides a user-friendly approach to obtain the same functionality as other programming models, with support for tightly and non-tightly nested loops. `Kokkos::MDRangePolicy` supports up to 6D iteration spaces and tiling based iteration. Users can specify layout and iteration orders, which are important for performance optimization.

III. MDRangePolicy and GPU backends

This section covers how `MDRangePolicy` is implemented for GPUs and compares it with its hierarchical counterparts, focusing on how these policies map their work-items. We then analyze the origin of the performance issues that motivated this work and present the improvements, which are now a part of Kokkos 5.2.0. Note that this work does not cover the OpenACC backend.

In CUDA, an n-dimensional parallel loop can be implemented by writing a kernel that computes the indices from `blockIdx` and `threadIdx` with bounds checks and invoking the user functor with the resulting indices [21]. When the thread block has more than one dimension, `threadIdx.x` must be mapped to the fastest-varying index, so that consecutive threads access consecutive addresses in memory [22]. Therefore, `threadIdx.x` should follow the iteration order given by `inner_direction` and the remaining indices are mapped to the `y` and `z` hardware dimensions.

In Kokkos there are two possible implementations for a multi-dimensional parallel loop: (1) The `TeamPolicy` approach that leverages the hierarchical nature of the GPU architecture, with its *work-group* and *work-item* levels. (2) The `MDRangePolicy` approach that processes the iteration space with global indexing.

1) TeamPolicy Approach: This approach uses the work-group and local work-item indices to index multiple dimensions. The terms work-group and work-item, in SYCL [23], correspond to a thread block and a thread in CUDA and HIP. The outer dimension is parallelized over the teams: the team index is provided by the `TeamPolicy` and maps to the work-group index. The inner dimensions are parallelized via `TeamThreadMDRange` or `TeamVectorMDRange`, which use up to two work-group dimensions for indexing. These

policies rely on sequential `for` loops for any dimension beyond those exposed by the device backend API.

Listing 6. TeamPolicy approach for rank 3

```
int beg0 = threadIdx.x + lower[0];
int beg1 = threadIdx.y + lower[1];
int beg2 = blockIdx.x  + lower[2];

// Iterate Left
for (int i2 = beg2; i2 < upper[2]; i2 += gridDim.x) {
  for (int i1 = beg1; i1 < upper[1]; i1 += blockDim.y) {
    for (int i0 = beg0; i0 < upper[0]; i0 += blockDim.x)
     {
      functor(i0, i1, i2);
    }
  }
}
```

Listing 6 illustrates the approach for work mapping in TeamPolicy via native CUDA/HIP code. Here `i0` and `i1` are mapped onto thread indices. In CUDA, the maximum block size in `x` and `y` is 1024, so a block-stride loop is needed to extend the iteration space [24]. `i2` corresponds to the team index and is mapped to the block index. For the grid dimension in `x`, the maximum block index is $2^{31} - 1$. Since the index type is `int`, a `for` loop is not strictly necessary here.

The downside of this API is the use of the sequential loops that are needed to cover the higher dimensions and to extend the iteration space due to limited block sizes.

2) MDRangePolicy Approach: This approach uses a *global* index, meaning that there is no distinction between the thread and block levels (no hierarchy). As a consequence, we lose work-group-level capabilities such as scratch memory and synchronization. For ranks 1–3, each logical dimension maps directly to one hardware dimension. To support ranks beyond the (`x`, `y`, `z`) provided by the device, two consecutive logical dimensions are packed into one hardware dimension. This technique, also known as flattening, maps a higher dimensional index into a lower one. It is applied for ranks 4–6, and the order of the packed dimensions is shown in Table I.

Listing 7 illustrates this packing for a rank 4 policy via native CUDA/HIP code, where `tile_size` and `num_tiles` are the tile size and the number of tiles, for each logical dimension. The two logical indices packed into the hardware dimension `x` are recovered with one modulo and one division, while the remaining indices are read directly from `y` and `z`.

Listing 7. `MDRangePolicy` unpacking for rank 4

```
// Global thread index along the packed dimension x
int globalIdx_x = blockIdx.x * blockDim.x + threadIdx.x;

// Extent of the first logical dimension
int extent_0 = tile_size[0] * num_tiles[0];

// Unpack the global index into the two logical indices
int i0 = globalIdx_x % extent_0;
int i1 = globalIdx_x / extent_0;

// Read i2, i3 directly from the hardware dimensions
int i2 = blockIdx.y * blockDim.y + threadIdx.y;
int i3 = blockIdx.z * blockDim.z + threadIdx.z;

// bounds checks omitted for clarity
user_functor(i0, i1, i2, i3);
```

The number of work-groups that can be launched is bounded by the API limits of the device backends. To cover iteration spaces larger than this bound, a grid-stride loop [25] is used. On GPUs, `MDRangePolicy` tiles map directly to the work-group dimensions for policies of rank 1–3. For ranks 4–6, we pack the tile sizes into the three work-group dimensions in the same way as the indices (see Table I). This means that tiles are executed in parallel, and one device thread executes one point of the nd-range. This is a major difference from the CPU backends, where a thread executes an entire tile. Moreover, the work-group size is bounded by the device backend limits, which is not the case on CPU, where tile sizes are not constrained.

In Kokkos, index packing happens on host side when a `parallel_for` is launched, while unpacking happens on device side, in a dedicated structure. Only CUDA, HIP and SYCL backends use this structure and it is not user visible. This is responsible for mapping the user's iteration space onto the hardware resources. We will see that the old implementation of this structure was the main cause of the observed performance issues.

TABLE I
MDRANGEPOLICY INDEX MAPPING TO CUDA AND HIP. SYCL MAPPING IS THE SAME BUT WITH `z` AND `x` SWAPPED.

Rank	MDRangePolicy indices	x	y	z
1	i0	i0	1	1
2	i0, i1	i0	i1	1
3	i0, i1, i2	i0	i1	i2
4	i0, i1, i2, i3	i0, i1	i2	i3
5	i0, i1, i2, i3, i4	i0, i1	i2, i3	i4
6	i0, i1, i2, i3, i4, i5	i0, i1	i2, i3	i4, i5

A. Main Issues

The code in Listing 8 shows how the unpacking logic for rank 4 was earlier implemented within Kokkos. We see that the innermost index for `Iterate::Right`, i_3, never maps to the x dimension, whereas this is always the case for `Iterate::Left` with i_0. There is also an asymmetry in work-group size between the two orders. For example, with HIP and rank 4 we have tile size of (16,4,4,1) for `Iterate::Left` and (1,4,4,16) for `Iterate::Right`. Since the tile sizes are packed in the same fixed order as the indices, this leads to work-group sizes (64,4,1) and (4,4,16), respectively. These two aspects led to performance discrepancies between the two iteration orders, which has been addressed by our work.

Listing 8 also shows that the kernel reasons in terms of two levels of thread hierarchy (work-group, then work-item), even though this distinction is not needed for this policy. This is why it unpacks the index in two steps, one modulo and one division at each level, instead of unpacking it directly from the global thread index. Similarly, the code uses a grid-stride loop for every logical dimension, whereas such loops are only needed when the iteration space exceeds the API limits. Also,

Listing 8. Old unpacking logic

```
// Shape of the packed block grid
int num_blk_0 = /* ... */;
int num_blk_1 = /* ... */;

// Unpack blockIdx.x -> block coordinates
int blk_0 = blockIdx.x % num_blk_0;
int blk_1 = blockIdx.x / num_blk_0;

// Unpack threadIdx.x -> thread coordinates within a tile
int thread_0 = threadIdx.x % tile_size[0];
int thread_1 = threadIdx.x / tile_size[0];

// Grid-stride loop over the tile grid
for (int tile_1 = blk_1; tile_1 < num_tiles[1];
     tile_1 += num_blk_1) {
  int i1 = tile_1 * tile_size[1] + thread_1;
  if (i1 < upper[1]) {
    for (int tile_0 = blk_0; tile_0 < num_tiles[0];
     tile_0 += num_blk_0) {
      int i0 = tile_0 * tile_size[0] + thread_0;
      if (i0 < upper[0]) {
        // i2, i3 read from hardware dimensions y and z
        user_functor(i0, i1, i2, i3);
      }
    }
  }
}
```

the computation of `num_blk_0` and `num_blk_1` happens inside the kernel although it could be performed before the kernel launch.

Thus, the old implementation had, for each rank, one `for` loop (striding over the number of tiles) followed by one `if` statement (checking the upper bound). For rank 6, this leads to six nested `for`/`if` blocks with the user's functor inlined at the innermost level of this structure. With this branching complexity, Kokkos' part in the resulting device kernel leaves little room for the compiler to optimize the user's lambda.

There was one more aspect that impacted execution efficiency: the default tile sizes, which directly control the work-group sizes. The default work-group sizes were inconsistent across ranks. With CUDA, they doubled at each rank, from 32 work-items at rank 2 to 256 from rank 5 onwards. With HIP, rank 2 used 64 work-items and all higher ranks 256. There is no reason for a rank 2 policy to use fewer work-items than a rank 6 one. A poorly chosen tile size leads to poor performance. With CUDA, the default tile size for rank 2 was (16, 2), i.e., 32 threads per block. On NVIDIA H100, at most 32 blocks can be resident per Streaming Multiprocessor (SM), so this yields at most 1024 resident threads per SM, which is half of the 2048 supported by this architecture [24]. A 32-thread block also contains a single warp, with no intra-block parallelism to hide latency. This 50% ceiling on theoretical occupancy penalizes kernels that rely on high occupancy to hide memory latency.

Finally, the old implementation was hard to optimize as-is. The unpacking logic was duplicated for each rank and each layout. It relied on a partial specialization of a struct based on the rank, with the whole structure duplicated for `Iterate::Left` and `Iterate::Right`. This work is therefore also an opportunity for refactoring: we will bring the unpacking logic, the grid-stride loop and the index computation together in one place. Also, the old implementation of `MDRangePolicy`, for both GPUs and CPUs, was based on the pre-C++17 standard. This work is an opportunity to use newer features like `if constexpr`, `consteval` and `concepts`.

B. Implemented Solutions

The issues can be addressed by applying what is generally considered good practice in GPU programming: writing simple kernels with low control flow complexity to avoid branch divergence, keeping index arithmetic cheap and ensuring coalesced memory accesses. We present three changes that have been upstreamed to Kokkos 5.1.0 and 5.2.0. The first change refactors the old index unpacking logic into the new *`DeviceIterate`* structure. The second retunes the default tile sizes. The third adds a variant of `DeviceIterate` that does not use a grid-stride loop.

1) Refactored DeviceIterate: In Listing 9, i_k denotes the user's policy indices, while j_k denotes the internal indices used inside `DeviceIterate`, ordered from the fastest-varying to the slowest-varying. j_0 is always mapped to `threadIdx.x`, i.e. to the hardware dimension x (z for SYCL), which is important for performance [22]. For `Iterate::Left`, $j_k = i_k$; for `Iterate::Right`, $j_k = i_{N-1-k}$ to match the user's indexing order.

Listing 9. Unpacking logic from the new `DeviceIterate` structure

```
template <unsigned R, typename... Idxs>
KOKKOS_INLINE_FUNCTION void iterate(Idxs... idxs) {
  constexpr unsigned rankIdx = R - 1;
  int begin  = my_begin<rankIdx>();
  int end    = my_end<rankIdx>();
  int stride = my_stride<rankIdx>();

  for (int idx = begin; idx < end; idx += stride) {
    if constexpr (is_packed_index<rankIdx>()) {
      // Ranks that are packed together in a single
     linear index
      constexpr unsigned lo = /*...*/, hi = /*...*/;
      // Unpack into two multi-dimensional indices
      int j_lo = idx % extent[lo] + lower[lo];
      int j_hi = idx / extent[lo] + lower[hi];
      if (j_lo < upper[lo] && j_hi < upper[hi]) {
        if constexpr (IterateDir == Iterate::Left)
          iterate<R-2>(j_lo, j_hi, idxs...);
        else
          iterate<R-2>(idxs..., j_hi, j_lo);
      }
    } else {
      // Ranks that are not packed together
      if constexpr (IterateDir == Iterate::Left)
        iterate<R-1>(idx, idxs...);
      else
        iterate<R-1>(idxs..., idx);
    }
  }
}
```

Indices are now unpacked from the global thread index. The grid-stride loop now iterates over the packed global index: we swap the order from unpack → loop to loop → unpack, so that a single `for` loop is needed per hardware dimension of the device backend API. Two-step unpacking is no longer required: we reason in terms of the `extent` (`work_item` × `work_group`), which enables unpacking in a single phase, with one modulo and one division instead of two, and a single `for`+`if` block per packed index instead of two nested ones.

The kernel therefore contains at most three nested `for` loops, corresponding to the (`x`, `y`, `z`) dimensions of the device, and at most three nested `if` statements, since consecutive unpacked indices are checked in the same statement. For a rank 6 policy this halves the nesting depth, from six `for` loops and six `if` statements down to three of each. A simplified version is shown in Listing 9.

The new implementation is based on a recursive template whose recursion depth depends on the rank of the policy, and it gathers all the unpacking logic in a single place: the computation of the `begin`, `end` and `stride` variables depends only on two compile-time constants, the policy rank and the index position. The differences between `Iterate::Left` and `Iterate::Right` are handled by two `if constexpr` statements within the same function. We end up with a more maintainable codebase displaying clearer intent.

Because $j_k = i_{N-1-k}$ for `Iterate::Right`, the bounds must be swapped before calling `DeviceIterate`: the internal index $N-1$ receives the bounds of the user's index 0 and so on. For rank 4 and `Iterate::Right`, the recursion then unfolds as `iterate<4>()` $\rightarrow$ `iterate<3>(`j_3`)` $\rightarrow$ `iterate<2>(`j_3`,`j_2`)` $\rightarrow$ `iterate<0>(`j_3`,`j_2`,`j_1`,`j_0`)`, where the packed pair handles two ranks at once and the $R = 0$ specialization invokes the user functor. The final call, `functor(`j_3`,`j_2`,`j_1`,`j_0`)`, therefore passes the indices in the user's order, since $i_k = j_{3-k}$. The same swapping is applied to the tile sizes to ensure consistency.

2) New default tile sizes: The default tile sizes for the GPU backends have been retuned to improve performance. Choosing a good default that performs well for a wide variety of problems is challenging: one has to take into account the rank of the problem, the target architecture, the problem extents and the register pressure of the user's kernel. The new defaults are based on empirical measurements using the Stream and stencil benchmark suites described in Section V-B, run over multiple architectures and a range of problem sizes. For each rank, we selected the tile size performing best across the test cases. We restricted the search domain to power-of-two extents, with at least 32 work-items in the hardware dimension `x` once the tile sizes are packed, and at most 512 work-items per work-group.

Auto-tuning the tile-size based on the weight of the user kernel (i.e. the number of registers) can also be considered for `MDRangePolicy` as already done in `RangePolicy` and `TeamPolicy` and will be part of future work.

3) Optional grid-stride loop: The grid-stride loop is now used only when necessary. To this end, we compile two kernels: one with the grid-stride loop and one without. The decision of which kernel to launch is made at runtime. The problem size is known at policy construction and the decision is made when the policy is passed to `parallel_for`, based on the launch configuration derived from the backend API. Without grid-stride loops, the code contains no `for` loops. Instead, a single final `if` statement checks all the indices before calling the user kernel. In Listing 9, this only requires removing the `for` loops and computing `idx` directly from the global thread index. This is the simplest possible form of such a kernel.

IV. MDRangePolicy and CPU backends

This section describes `MDRangePolicy` for CPU backends. We note two avenues for improvement and our attempts to address them. Unlike Section III, we focus only on `MDRangePolicy` and not on hierarchical `TeamPolicy`. This work has been tested on Serial, OpenMP and Threads backends with GCC and Clang.

On CPU backends, the grid of tiles, described in Section II-B, is processed in a two-level manner. At the outer level, there is a flat loop over the tiles. At the inner level, for each tile, there is a nested for loop over all the dimensions, with the fastest-varying dimension as the innermost loop. Thus, unlike GPU backends, a given global index within the multi-dimensional iteration space, can be readily obtained from the loops associated with these two levels.

A. Main issues

In multi-core CPUs, parallelism can be achieved in two complementary ways - thread-level parallelism (TLP) and SIMD parallelism. In Kokkos, TLP is achieved by parallelizing the flat outer loop (e.g., via `#pragma omp parallel for`). The work-items (in our case, tiles) are distributed across multiple threads and the computation associated with a tile is executed by a CPU thread. Within a thread, SIMD offers data-level parallelism. There are many algorithms in scientific computing that can benefit from such parallelism.

In Section II, we saw that the computations performed in `Kokkos::parallel_for` are identical and independent across indices and that the iteration order of `MDRangePolicy` leads to unit stride for the innermost loop. Thus, we see that `parallel_for` with `MDRangePolicy` is vectorization friendly and SIMD parallelism could be introduced in the innermost loop of the nest, i.e., inner-loop vectorization. In strip-mining [26] based vectorized code, this loop's iteration space is divided into strips that are executed in a SIMD fashion and any remaining iterations are handled separately (e.g., a scalar loop). This code can be either implemented by the user or generated by the compiler.

However, obtaining such a vectorized form faces important concerns that include data alignment, aliasing, data indexing pattern and conditional flow in loop body. In practice, there are many situations that accommodate these concerns. For example, Kokkos guarantees that when a `View` is allocated, it is aligned by 64 bytes. In many user kernels, `Views` are independently allocated and do not alias.

If a user is ready to trade productivity for performance, while maintaining portability, it is possible to implement vectorized code using the SIMD API described in Section II-C1. The user implements the kernel using `Kokkos::Experimental::simd` types and also, the logic to traverse the innermost loop with step size equal to the strip length and to process the remainder. However, the user could face implementation challenges due to the above

concerns. For example, if the computation involves subviews (NumPy-like slices of `Views`) or `LayoutStride Views` or unmanaged memory allocations, data alignment requires attention.

Many problems that are relatively convenient for manual porting, can, in fact, be auto-vectorized by the compiler. Thus, the user gains in productivity. Much progress has been made in loop nest optimization by compilers [27], [28] and this is especially true for inner-loop vectorization. In order to vectorize a loop, the compiler analyses whether it is possible and profitable [29]–[31]. What we have observed up to Kokkos 5.2.0 is that GCC's analysis decides not to vectorize the inner-loop. This is despite `Kokkos::parallel_for` with `MDRangePolicy` being vectorization friendly. We also describe a vectorization failure case with Clang in Section V-F.

Finally, the CPU backends internally use a structure called `HostIterateTile` and associated code. Here, nested for loops of ranks 1 to 6 are generated via macro substitution. There are repetitive macros (8 sets of macros, with a macro per rank in each set), which handle iteration order being left or right and the functor being tagged or not. It takes much effort to modify this code, just like in the older GPU implementation.

B. Implemented Solutions

Our attempts were along the following lines: First, encouraging auto-vectorization to improve performance. Second, refactoring for better maintainability.

1) Encouraging Auto-vectorization: We investigated three factors that finally enabled inner-loop vectorization in `MDRangePolicy`: (1) The loop being countable, (2) Data indexing in the loop body being simple, (3) No narrowing type conversion of the arguments when calling the user functor.

To discuss these three aspects, we point out that `parallel_for` with `RangePolicy`, shown in Listing 10, readily leads to auto-vectorization with GCC, while the same computation expressed via rank 2 `MDRangePolicy` doesn't.

Listing 10. RangePolicy and MDRangePolicy for STREAM Triad

```
Kokkos::parallel_for("rp", Kokkos::RangePolicy<>(0, M),
  KOKKOS_LAMBDA(const int i) {
    for (int j = 0; j < N; ++j) {
      a(i, j) = b(i, j) + scalar * c(i, j);
    }
  });

Kokkos::parallel_for("mdrp",
  Kokkos::MDRangePolicy<Kokkos::Rank<2>>({0, 0}, {M, N}),
  KOKKOS_LAMBDA(const int i, const int j) {
  a(i, j) = b(i, j) + scalar * c(i, j);
  });
```

We also note that the innermost loop has always been annotated with a pragma that indicates independence across iterations, i.e., `#pragma ivdep`. We also experimented with GCC and Clang specific pragmas. These pragmas were insufficient (especially for GCC) and further compiler-friendly changes were needed.

In Kokkos, the innermost loop is countable for all policies, whether the loop index variable is signed or unsigned. This is because the loop extents are always known when the loop is entered at run time. For signed integers, overflow is undefined behaviour and this aspect of signed integers is leveraged for optimizations [32]. If overflow does happen, then executing vectorized code is as good as any other behavior. For unsigned integers, overflow leads to wrap-around but the loop is still countable. Compilers handle this wrap-around by executing a scalar loop at run time.

The global indices of an element within a tile are computed by adding that tile's global offsets to the loop indices of the nested for loops. This computation occurs at all levels of the nested loops, in a manner similar to what is shown in Listing 11 for the innermost loop. Since, the same global offset is added within a given loop, the data indexing can be assessed to be simple in our case. Thus, the first two aspects may not stand in the way of auto-vectorization in Kokkos.

The type of the loop indices in the nested for loops is determined by the `IndexType` template parameter of `MDRangePolicy`. Since Kokkos 5.2, it defaults to `std::int64_t` type for host backends. Further, the global offsets of a tile are internally represented as `std::int64_t`. Thus, passing the global indices to the user-defined functor can involve a narrowing type conversion, e.g., from `std::int64_t` to `int`. This has been the cause for vectorization analysis to fail in GCC.

We realized that users often define functors taking `int` type arguments, like in Listing 10. For this 2D problem, both `RangePolicy` and `MDRangePolicy` execute a two-level nested loop. The target for vectorization is the computation in the innermost loop. In the case of `RangePolicy`, the inner-loop is inside the user functor and its loop index variable is also of `int` type (chosen by the user). When narrowing type conversion happens when calling the user functor, it concerns the outer-loop, which is internal to Kokkos. We note that type conversions still happen when executing the body of the inner-loop. It is from `int` to `std::size_t`, when indexing into the `Views`. This induction variable widening is supported by compilers [33]. For `MDRangePolicy`, all the loops are internal to Kokkos and there is narrowing conversion from `std::int64_t` to `int` with respect to all the arguments of the user functor.

Thus, we need a way to avoid these narrowing type conversions when calling the user functor in `MDRangePolicy`. Fortunately, the computation of the global indices based on tile indices and fixed tile offsets is simple in our case. The final loop bounds can be computed outside, safely cast to the desired signed integer type and passed to the function executing the nested for loops. That way, there won't be any type conversion when the user functor is called.

Now, the nested for loops need to match the type of their loop indices to this signed integer type. The user could inform Kokkos of the type of their kernel's arguments via the template parameter `IndexType` of `MDRangePolicy`. Internally, we could implement a specialization of `HostIterateTile` when `IndexType` is of signed integer type. In this specialization, the above nested for loops are executed after verifying that the entire iteration space can be represented by the given

signed integer type.

Our implementation based on this approach indeed leads to auto-vectorization with GCC. We have been able to consistently obtain SIMD along with TLP for the STREAM and Stencil benchmarks, as shown in V-F. We point out that so far in our implementation, we specialize `HostIterateTile` for `IndexType` being `int`. We will generalize it to any signed integer type soon. We observe that Clang can vectorize the innermost loop with or without our modification to `MDRangePolicy`, as its vectorization heuristics appear less conservative than GCC's. However, our approach is compiler-friendly in general.

Listing 11. Forms of the innermost loop

```
// Kokkos 5.2.0 implementation
#pragma ivdep
for (index_type i0 = (index_type)0;
     i0 < static_cast<index_type>(tile[rank-1]); ++i0) {
  func(/* variadic arguments */, i0 + offset[rank-1]);
}

// Our vectorization-friendly implementation
#pragma ivdep // or GCC/Clang specific pragma
for (int i0 = extent_st[rank-1]; i0 < extent_en[rank-1];
     ++i0) {
  func(/* variadic arguments */, i0);
}
```

Listing 11 shows, after macro expansion, how the code looks like for the innermost loop of the nested for loops. This case corresponds to `Iterate::Right` and an untagged functor. Here, `index_type` is an alias for `MDRangePolicy::index_type`, `rank` denotes `MDRangePolicy::rank` and after macro expansion, the arguments from the first to the penultimate position appear explicitly in the generated code. The user-defined kernel, `func`, takes index arguments as `int` type. Thus, in our code path, the underlying type of `index_type` is `int` and the loop bounds already account for the tile offsets.

Other ways to ensure no narrowing type conversion when calling the user functor are: (1) Use kernel arguments of type `MDRangePolicy::point_type::value_type`, the internal type used for representing tile offsets. (2) Implement kernels in a generic manner. Refer to the STREAM benchmarks that we have contributed to Kokkos. Both these approaches are not user-friendly, for implementation and for allowing the flexibility of using smaller types.

As future work, it will be nice to develop a compiler pass (say, using LLVM [34]) that generates `Kokkos::Experimental::simd` based code for a given user-defined functor. To start with we could target simpler cases like stencil computations.

2) Refactoring for better maintainability: To aid our experimentation, we implemented the generation of nested for loops using recursive templates, as in the GPU backends III-B1. This implementation was merged into Kokkos but led to performance regression in a downstream application. Thus, `MDRangePolicy` has been reverted to the macros based approach for CPU backends. This experience highlights a big advantage with macros: performance. Macro substitution happens at the preprocessing stage and the nested for loops are always present in the function body. With recursive templates, the inlining budget could run out, leading to function calls. Thus, there are trade-offs involving code maintainability and performance.

V. Benchmarks and Results

In this section, we show how the optimizations described in Sections III and IV lead to performance improvements. Our benchmarks are based on our own contributions to the performance test suite of Kokkos, which were merged at the very beginning of our work.

A. Settings for performance measurements

Table II summarizes the devices and compilers used in the present work. For GPUs, we use NVIDIA H100 [35], AMD MI300A [36] (with `XNACK` enabled), and Intel Data Center GPU Max 1550 [37] (hereinafter PVC). For PVC, we treat a single tile as a device, and the metrics in Table II should be halved. AMD EPYC 9654 is our multi-core CPU.

B. Benchmark codes

Our first benchmark suite consists of stencil kernels for ranks 2, 3 and 4. It is a $(2d+1)$-point stencil, where d is the rank, implemented for `Views` with `LayoutRight` and `LayoutLeft` and a matching iteration order for the policy. Two `Views` of floating-point elements are involved and the dimensions are 10240^2 (2D), 512^3 (3D) and 96^4 (4D).

Our second benchmark suite implements the STREAM benchmark [39] for ranks 2 to 6 with kernels performing the Set, Copy, Scale, Add, and Triad operations on `Views` with `LayoutLeft` (GPUs) or `LayoutRight` (CPUs). The size in each dimension is adjusted so that the total number of elements is the same for all ranks: 24^6 (GPUs) and 32^6 (CPUs). For example, for GPUs, rank 3 problem size is $(24^2 \times 24^2 \times 24^2)$, rank 4 is $(24^2 \times 24 \times 24 \times 24^2)$, and so on. If the element type is double, each view occupies 1,458 MiB of memory. Rank 1 is measured separately against an equivalent `RangePolicy` kernel, over problem sizes from 2^{22} to 2^{26} elements. We report the bandwidth $B = 3 \times N \times \text{sizeof}(T)/(t \times 10^9)$ in GB/s, where N is the number of elements, T is the element type, and t the elapsed time in seconds. The factor 3 accounts for the two loads and one store of the Triad kernel.

These benchmarks were selected because `MDRangePolicy` is the natural choice for parallelizing these kernels. The data elements are of type `double` for the GPU benchmarks and `float` for the CPU ones. Unless stated otherwise, we use the default tile size of the Kokkos version under test. To keep the figures concise, we plot only the Triad kernel of the STREAM suite, as it is representative of the other operations.

1) Kokkos variants for benchmarking:

- **Baseline**: Kokkos 5.0.2.
- **Refactored**: with refactored `DeviceIterate` kernel.
- **New tile sizes**: with new default tile sizes.

TABLE II
HARDWARE DESCRIPTIONS EXTRACTED FROM VENDORS' DATASHEETS [35]–[38].

Processor	**AMD EPYC** 9654	**NVIDIA** H100	**AMD** MI300A	**Intel** PVC
Number of cores	96	132 SMs	228 Compute Units	128 Xe Cores
Shared cache [MB]	384 (L3)	50 (L2)	256 (Infinity Cache)	408 (L2)
Peak compute [TFLOPS]	3.686	33.5	122.6 Matrix, 61.3 Vector	52
Peak B/W [GB/s]	460.8	3,350 (HBM3)	5,300 (HBM3)	3,276.8 (HBM2e)
SIMD/Warp/Wavefront	512 bits	32 threads	64 threads	16 threads
Compilers	GCC 13.2, Clang 18.1.3	CUDA 12.8, GCC 13.3	ROCm 7.2, Clang 22	Intel LLVM 2025.3.2

- **No grid-stride**: with no grid-stride loop inside the kernel.
- **Auto-Vect**: encouraging auto-vectorization in CPUs.

For GPUs, each variant includes the modifications of the previous one, so the reported speedups are cumulative and are always measured against the baseline.

C. Stencil benchmarks on GPUs

Figure 1 shows that refactoring the kernel had no effect on the CUDA backend for rank 2. The problem stems from low occupancy caused by the default tile size, which is why there is a large speedup with the new default tile size for rank 2 (*New tile sizes* bar), regardless of the iteration order.

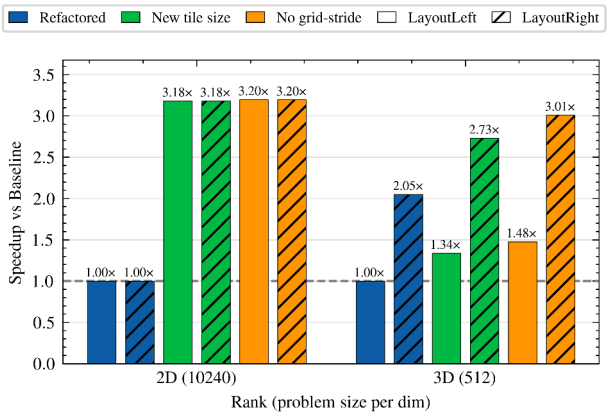


Fig. 1. NVIDIA H100: Stencil results for different ranks and memory layouts. Y-axis shows the speedup.

On HIP and CUDA backends, we can see that the speedups for `LayoutRight` are larger than those for `LayoutLeft` at rank 3 and also at rank 2 with HIP. This is due to the refactoring of `DeviceIterate`, which fixed the mapping of hardware resources to the memory layout.

On the HIP backend, as shown in Figure 2, there is a small regression for ranks 2 and 3 with the refactored `DeviceIterate` kernel functor and `LayoutLeft`. The slowdown is below 5% but we do not have a clear explanation for this. The new default tile size and the removal of the grid-stride loop, show a speedup for all ranks and both iteration orders.

Figure 3 shows how the optimizations benefit the SYCL backend. On this architecture, the dimension mapping is opposite to CUDA and HIP, as the innermost dimension is mapped

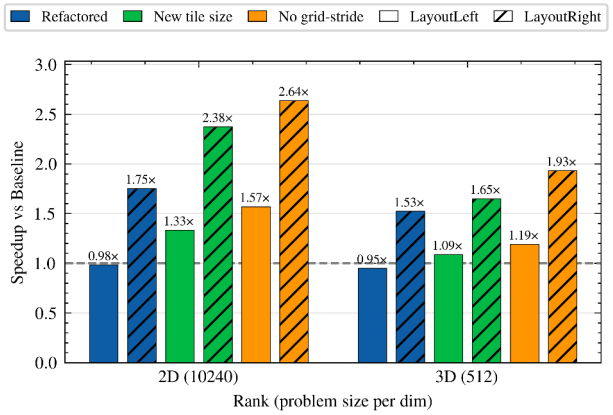


Fig. 2. AMD MI300A: Stencil results for different ranks and memory layouts. Y-axis shows the speedup.

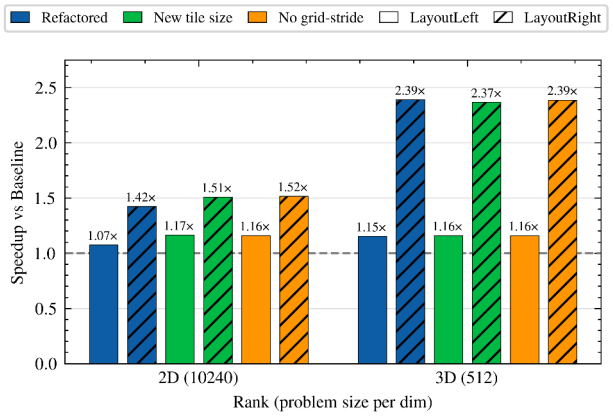


Fig. 3. Intel Ponte Vecchio: Stencil results for different ranks and memory layouts. Y-axis shows the speedup.

to z. The refactoring accounts for this, so the layout-dependent behaviour is the same as other backends. The speedup gap between `LayoutRight` and `LayoutLeft` is pronounced here as well: `LayoutRight` reaches up to $1.52\times$ (2D) and $2.39\times$ (3D), against $1.16\times$ for `LayoutLeft` in both cases.

D. STREAM benchmarks on GPUs

Figure 4 shows the achieved bandwidths for `RangePolicy` and rank-1 `MDRangePolicy` for the STREAM Triad kernel on NVIDIA and AMD GPUs.

Rank-1 `MDRangePolicy` was introduced after the GPU optimizations described in Section III-B, and we use `RangePolicy` as the reference. Both policies achieve comparable bandwidths for this kernel.

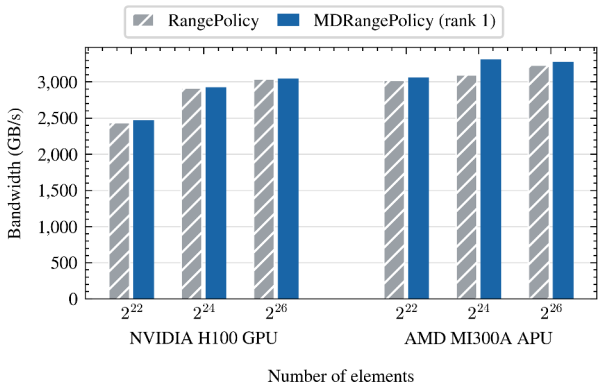


Fig. 4. STREAM Triad bandwidth for `RangePolicy` and rank 1 `MDRangePolicy` on NVIDIA H100 GPU and AMD MI300A APU.

Figures 5 and 6 show results for performance optimizations described in Section III. For the baseline, bandwidth decreases from rank 3 to rank 6 and two distinct causes are at play. At rank 2, occupancy is limited by tiles that are too small, which the new default tile size fixes. At higher ranks, occupancy is limited by register pressure, as reported in Table III. The latter is addressed by the refactoring of `DeviceIterate` (*Refactored* bars), which raises the bandwidth from 1370 to 2940 GB/s at rank 5 on H100. Consistently, the refactoring has the largest impact for ranks 4–6 on CUDA and ranks 3–6 on HIP, whereas the new tile sizes matter most for ranks 2 and 6, and for rank 3 on H100. On NVIDIA hardware, we reach 89.5% of the vendor-announced 3352 GB/s. On MI300A, the achieved bandwidth of about 3000 GB/s falls short of the 5300 GB/s peak, but a more relevant reference is the one-dimensional HIP BabelStream implementation [40], which reaches 3650 GB/s. We would like to explore optimizations like unrolling or using built-in vector types as future work.

The overall speedup is larger on CUDA than on HIP, because the CUDA backend suffered more from low occupancy. After our optimizations, the bandwidth stays approximately steady from rank 2 to rank 5 on H100, whereas it declines gradually beyond rank 4 on MI300A. Rank 6 still remains about 20% below rank 2 on both architectures, which we attribute to index computation overhead.

E. GPU metrics

From table III, we will first discuss metrics for rank 5 STREAM Triad. The low-occupancy issue with high ranks is noticeable. Refactored `DeviceIterate` reduces both register and branch counts, lifting the occupancy limitation. Removing grid-stride loop reduces them further, as expected, but its effect on the achieved bandwidth is marginal. The same table reports metrics for 3D stencil with `Iterate::Right`. The 2D case is omitted because the default tile size limits

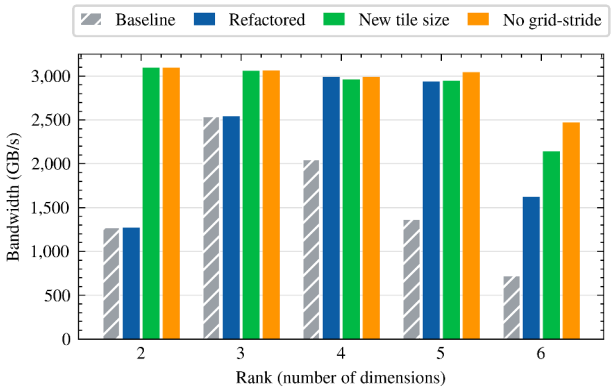


Fig. 5. NVIDIA H100: STREAM Triad results for different ranks. Y-axis shows bandwidth (GB/s).

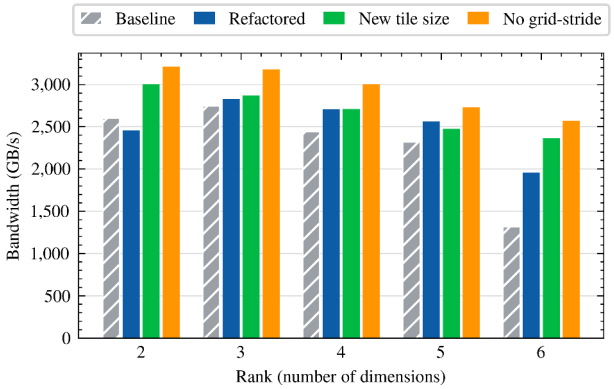


Fig. 6. AMD MI300A: STREAM Triad results for different ranks. Y-axis shows bandwidth (GB/s).

bandwidth considerably. Occupancy decreases with the refactoring while bandwidth increases, indicating that the kernel is no longer latency-bound. The new default tile size then increases occupancy and along with it, bandwidth.

F. Benchmarks on CPUs

To start with, we observed speedup for all benchmarks, across all ranks, on the Serial backend. For multi-threaded execution on CPUs, we tested with both OpenMP and Threads backends. We observed similar trends with both, thus only report OpenMP results on AMD EPYC 9654. With OpenMP, threads are placed on a given Core Complex Die (CCD), before using another CCD. Also, we ensure no thread oversubscription.

Figures 7 and 8 show how auto-vectorization further improves performance on multi-threaded CPUs. When all 8 threads are on the same CCD, one expects the bandwidth to be limited by the Global Memory Interface (GMI) that connects a CCD to the I/O die [38], [41]. Even though there are 12 CCDs in each 9654 processor, the peak bandwidth of one GMI is more than $460.8/12 = 38.4$ GB/s. We note that as the rank increases both stencil and STREAM kernels

TABLE III
NVIDIA H100, CUDA 12.8. METRICS MEASURED USING NSIGHT COMPUTE.

Version	Compute	Bandwidth	Registers	Occupancy	Throughput (TB/s)	Branches (Inst)
Stream Triad (Rank 5)						
Kokkos 5.0.2	71 %	36 %	58	44 %	1.22	137 M
Refactored	68 %	84 %	32	80 %	2.81	78 M
New tile size	68 %	84 %	32	80 %	2.82	78 M
No grid-stride	56 %	89 %	18	79 %	3.00	30 M
Stencil 3D (`Iterate::Right`)						
Kokkos 5.0.2	25 %	42 %	31	89 %	1.37	46.1 M
Refactored	43 %	61 %	32	53 %	1.71	37.7 M
New tile size	55 %	66 %	32	82 %	2.22	37.7 M
No grid-stride	41 %	74 %	30	79 %	2.50	12.5 M

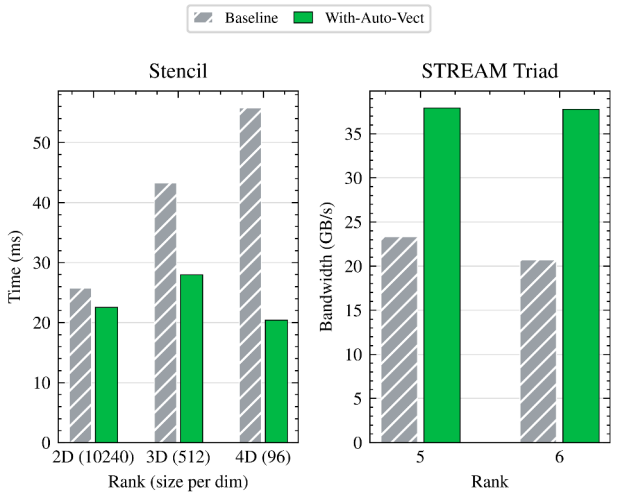


Fig. 7. AMD EPYC 9654 with 8 OpenMP threads: Stencil execution time for ranks 2, 3 and 4 (left) and STREAM Triad bandwidth for ranks 5 and 6 (right).

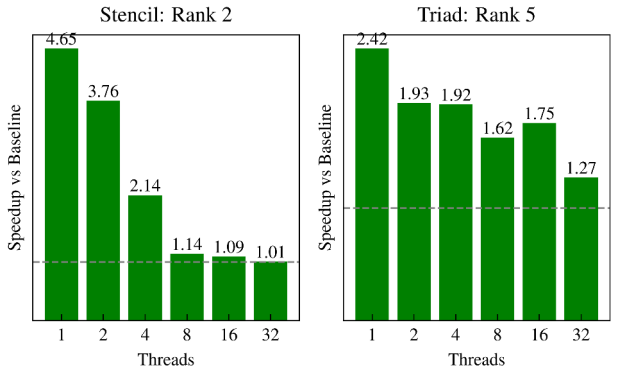


Fig. 8. AMD EPYC 9654: Results for two kernels across increasing number of threads. Y-axis shows the speedup.

perform more computations for traversing a tile. In the case of stencils, there is more computation within the loop body also. In such circumstance, the benefit from SIMD parallelism is more pronounced. Even for highly memory bound kernels like the rank 2 cases, there is a speedup when the number of threads is low.

Clang auto-vectorizes with or without our implementation for most of the benchmark cases. However, Clang always fails to vectorize computations involving rank 2 `Views`. We will investigate this in future work. Also, the investigation of NUMA effects remains future work.

G. Expanding the benchmark suite

As future work, we would like to explore more compute-bound kernels, e.g., stencils with larger radii. In this regard, the RAJA Performance Suite [42], [43] houses an impressive variety of kernels. They facilitate benchmarking by providing native implementations in CUDA/HIP/OpenMP, apart from implementations in RAJA and Kokkos. In recent months, we have already contributed a few Kokkos implementations of kernels. We are planning to pay more attention to the nature of the kernels in that suite and further contribute to it.

VI. CONCLUSION

In the present work, we have described our experiences improving the performance of a core component of the Kokkos library: the execution policy called `MDRangePolicy` for tightly nested loops. We detailed the design considerations and implementations of our work to improve performance on GPUs and CPUs. We evaluated the impact of our optimizations on the performance of `MDRangePolicy` using a set of benchmarks, and we have demonstrated significant performance improvements on both GPUs and CPUs.

ACKNOWLEDGMENT

This project was provided with computing and storage resources by GENCI at CINES and IDRIS thanks to the grant 2026-AD010616293R1 on the supercomputer Adastra's GENOA/MI300 partition and Jean-Zay's H100 partition. This research used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office

of Science User Facility operated under contract DE-AC02-06CH11357. Claude Opus and Codex were used to assist in generating the scripts that are used for plotting the results presented in this paper.

REFERENCES

[1] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[2] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming Model Extensions for the Exascale Era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[3] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable Performance for Large-Scale Scientific Applications," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019.

[4] C. R. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The Kokkos Ecosystem: Comprehensive Performance Portability for High Performance Computing," *Computing in Science Engineering*, vol. 23, no. 5, pp. 10–18, 2021.

[5] S. Rajamanickam, S. Acer, L. Berger-Vergiat, V. Dang, N. Ellingwood, E. Harvey, B. Kelley, C. R. Trott, J. Wilke, and I. Yamazaki, "Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels," 2021. [Online]. Available: https://arxiv.org/abs/2103.11991

[6] Y. Asahi, T. Padioleau, P. Zehner, J. Bigot, and D. Lebrun-Grandie, "kokkos-fft: A shared-memory FFT for the Kokkos ecosystem," *Journal of Open Source Software*, vol. 10, no. 111, p. 8391, 2025. [Online]. Available: https://doi.org/10.21105/joss.08391

[7] "High performance software foundation," https://hpsf.io/projects/kokkos/, last accessed: 2026-06-18.

[8] "HPSF Project Lifecycle Policies," https://github.com/hpsfoundation/tac/blob/main/lifecycle_policy.md, last accessed: 2026-08-11.

[9] J. M. Stone, P. D. Mullen, D. Fielding, P. Grete, M. Guo, P. Kempski, E. R. Most, C. J. White, and G. N. Wong, "Athenak: A performance-portable version of the athena++ AMR framework," *The Astrophysical Journal Supplement Series*, vol. 283, no. 1, 2026. [Online]. Available: https://doi.org/10.3847/1538-4365/ae3717

[10] M. Taylor, P. M. Caldwell, L. Bertagna, C. Clevenger, A. Donahue, J. Foucar, O. Guba, B. Hillman, N. Keen, J. Krishna, M. Norman, S. Sreepathi, C. Terai, J. B. White, A. G. Salinger, R. B. McCoy, L.-y. R. Leung, D. C. Bader, and D. Wu, "The simple cloud-resolving E3SM atmosphere model running on the frontier exascale system," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3627044

[11] E. Saikali, A. Bruneton, and P. Ledac, "Performances of the CFD open-source HPC platform TRUST on GPUs," *EPJ Web Conf.*, vol. 302, p. 03004, 2024. [Online]. Available: https://doi.org/10.1051/epjconf/202430203004

[12] E. Bourne, V. Grandgirard, Y. Asahi, J. Bigot, P. Donnel, A. Hoffmann, A. Kara, P. Krah, B. Legouix, E. Malaboeuf, Y. Munschy, K. Obrejan, T. Padioleau, M. Protais, and P. Vidal, "Gyselalib++: A portable c++ library for semi-lagrangian kinetic and gyrokinetic simulations," *Journal of Open Source Software*, vol. 10, no. 113, p. 8582, 2025. [Online]. Available: https://doi.org/10.21105/joss.08582

[13] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. R. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465521002836

[14] A. Johansson, E. Weinberg, C. R. Trott, M. McCarthy, and S. Moore, "LAMMPS-KOKKOS: Performance Portable Molecular Dynamics Across Exascale Architectures," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2025. [Online]. Available: https://doi.org/10.1145/3731599.3767498

[15] "mdspan: A proposal for a multidimensional array view," https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0009r18.html, last accessed: 2026-07-01.

[16] "SIMD extensions for C++," https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p1928r15.pdf, last accessed: 2026-07-15.

[17] "SYCL 2020 specification," https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, last accessed: 2026-07-17.

[18] D. Arndt, D. Lebrun-Grandie, and C. R. Trott, "Experiences with implementing kokkos' sycl backend," in *Proceedings of the 12th International Workshop on OpenCL and SYCL (IWOCL)*, 2024. [Online]. Available: https://doi.org/10.1145/3648115.3648118

[19] "std::ranges::cartesian_product_view," https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2374r4.html, last accessed: 2026-07-01.

[20] "std::views::indices," https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3060r3.html, last accessed: 2026-07-01.

[21] NVIDIA, *CUDA Programming Guide: Intro to CUDA C++*, https://docs.nvidia.com/cuda/cuda-programming-guide/02-basics/intro-to-cuda-cpp.html, last accessed: 2026-06-25.

[22] NVIDIA, "Unlock GPU performance: Global memory access in CUDA," https://developer.nvidia.com/blog/unlock-gpu-performance-global-memory-access-in-cuda/, last accessed: 2026-06-25.

[23] "Thread mapping and gpu occupancy," https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-2/thread-mapping-and-gpu-occupancy.html, last accessed: 2026-06-18.

[24] NVIDIA, *CUDA Programming Guide: Compute Capabilities*, https://docs.nvidia.com/cuda/cuda-programming-guide/05-appendices/compute-capabilities.html, last accessed: 2026-06-25.

[25] M. Harris, "CUDA pro tip: Write flexible kernels with grid-stride loops," https://developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/, last accessed: 2026-06-25.

[26] M. Weiss, "Strip mining on SIMD architectures," in *Proceedings of the 5th International Conference on Supercomputing*, 1991.

[27] V. Sarkar, "Optimized unrolling of nested loops," in *Proceedings of the 14th international Conference on Supercomputing*, 2000.

[28] M. Kruse and H. Finkel, "Design and use of loop-transformation pragmas," in *International Workshop on OpenMP*. Springer, 2019.

[29] "Auto-vectorization in GCC," https://gcc.gnu.org/projects/tree-ssa/vectorization.html, last accessed: 2026-07-15.

[30] "Vectorization optimization in GCC," https://developers.redhat.com/articles/2023/12/08/vectorization-optimization-gcc, last accessed: 2026-07-15.

[31] "Auto-Vectorization in LLVM," https://llvm.org/docs/Vectorizers.html, last accessed: 2026-07-15.

[32] F. G. Pikus, *The Art of Writing Efficient Programs*. Packt Publishing Ltd, 2021.

[33] J. Lee, Y. Kim, Y. Song, C.-K. Hur, S. Das, D. Majnemer, J. Regehr, and N. P. Lopes, "Taming undefined behavior in LLVM," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017. [Online]. Available: https://doi.org/10.1145/3062341.3062343

[34] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[35] NVIDIA, "NVIDIA H100 Tensor Core GPU Architecture," https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c, 2022.

[36] Advanced Micro Devices, "AMD CDNA™ 3 ARCHITECTURE," https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf, 2023.

[37] Intel, "Intel® Data Center GPU Max 1550," https://www.intel.com/content/www/us/en/products/sku/232873/intel-data-center-gpu-max-1550/specifications.html, 2023.

[38] Advanced Micro Devices, "AMD EPYC™ ARCHITECTURE," https://www.amd.com/content/dam/amd/en/documents/products/epyc/4th-gen-epyc-processor-architecture-white-paper.pdf, 2023.

[39] J. D. McCalpin, "Stream benchmark," https://www.cs.virginia.edu/stream/ref.html, 1995, last accessed: 2026-06-18.
[40] K. Aytekin, "How partition modes shape memory bandwidth on AMD MI300A: A BabelStream study," https://github.com/kkaytekin/mi300a_babelstream_cpx_tpx_spx, last accessed: 2026-09-18.
[41] Advanced Micro Devices, "Tuning Guide AMD EPYC 9004," https://docs.amd.com/v/u/en-US/58002_amd-epyc-9004-tg-hpc, 2024.
[42] O. Pearce, J. Burmark, R. Hornung, B. Bogale, I. Lumsden, M. McKinsey, D. Yokelson, D. Boehme, S. Brink, M. Taufer *et al.*, "RAJA Performance Suite: Performance Portability Analysis with Caliper and Thicket," in *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2024.
[43] Lawrence Livermore National Laboratory, "RAJAPerf: RAJA performance suite," https://github.com/llnl/RAJAPerf, last accessed: 2026-06-25.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: [PERFORMANCE IMPROVEMENT OF MULTI-DIMENSIONAL PARALLEL LOOPS IN KOKKOS]

A. Abstract

This appendix is the artifact description of the paper entitled "Performance Improvement of Multi-Dimensional Parallel Loops in Kokkos". It includes the instructions for compilation and experiment workflows. The expected results and the data evaluation method in the paper are also described.

B. Description

1) Check-list (artifact meta information):

- **Program:**
- **Compilation:** See "Installation" and "Compilation" section.
- **Binary:** `bench_mdrange_stencil`, `bench_mdrange_stream` (both GPUs and CPUs),
- **Run-time environment:** See Table II.
- **Hardware:** AMD EPYC 9654, NVIDIA H100, AMD MI300A, and Intel PVC.
- **Execution:** See "Evaluation and expected result" section.
- **Metrics:** Achieved bandwidth (GB/s) and kernel run time, as reported by Google Benchmark.
- **Output:** JSON performance profiles and PNG plots.
- **Experiment workflow:** See "Evaluation and expected result" section.
- **Publicly available?:** Yes

2) How software can be obtained (if available): The source code is publicly available on the Github repository: https://github.com/kokkos/code-examples.

3) Hardware dependencies: We have tested on the following hardware platforms summarized in table II. A single node with one GPU or one CPU is sufficient to run the benchmark codes.

4) Software dependencies: Our artifact relies on the following software dependencies:

- Kokkos [https://github.com/kokkos/kokkos] (fetched automatically)
- Google Benchmark [https://github.com/google/benchmark]
- CMake 3.28.2

C. Experimental workflow for "Benchmark" (section V)

1) Compilation: We provide a script that configures the benchmark codes. It creates a build directory, invokes CMake to fetch Google Benchmark and Kokkos, and checks out the relevant Kokkos versions evaluated in this paper. One sub-build directory is created per version, so that the same benchmarks are compiled against every version presented in this paper, see Table IV. The Kokkos versions for GPU backends are the first four items listed and the versions for CPU backends are the last two. GPU or CPU should be chosen by passing `--gpu` or `--cpu` as the first option to `configure_all.sh` and `run_platform.sh`. Kokkos and compiler options must be passed after the `--` separator on the command line. One should replace the compiler and Kokkos options according to the target platform.

The following commands clone the repository and configure the benchmarks for the Intel PVC platform.

```
git clone \
https://github.com/kokkos/code-examples.git
cd code-examples/papers/kokkos-mdrange-2026
./configure_all.sh --gpu --build-dir build \
  -- -DCMAKE_CXX_COMPILER=icpx \
     -DKokkos_ENABLE_SYCL=ON \
     -DKokkos_ARCH_INTEL_PVC=ON
```

Each sub-build directory is then compiled independently, for example, the following command compiles the baseline version for GPU backends.

```
cmake --build build/build_502
```

TABLE IV
MAPPING BETWEEN THE BUILD SUB-DIRECTORIES CREATED BY `CONFIGURE_ALL.SH` THE KOKKOS COMMIT EACH ONE IS BUILT AGAINST AND THE LABEL USED IN THE FIGURES. THE FULL COMMIT SHAS ARE PINNED IN THE TOP-LEVEL `CMAKELISTS.TXT` OF THE ARTIFACT.

Sub-directory	Commit	Label
`build_502`	`dfceb772`	Baseline
`build_device_iterate`	`7a78e92a`	Refactored
`build_new_tile`	`c6ab03f3`	New tile size
`build_no_stride`	`b77b924f`	No grid-stride
`build_520` (CPU)	`e484fccb`	Baseline
`build_auto_vect` (CPU)	`f2cac810`	With-Auto-Vect

2) Evaluation and expected results: Running a benchmark produces a performance profile in JSON format. The `run_platform.sh` script runs every benchmark against every variant for a given backend (GPU or CPU) from Table IV. This script takes three arguments: `--gpu`/`--cpu`, the build directory created by `configure_all.sh` and the target platform tag. The following command writes the profiles to the folders `out_sycl_pvc/stencil_bench` and `out_sycl_pvc/stream_bench`:

```
export ZE_FLAT_DEVICE_HIERARCHY=FLAT

./run_platform.sh --gpu build sycl_pvc
```

Once all the profiles have been collected, the plotting scripts in the `scripts` directory read the JSON files and produce the figures in PNG format. They require the Python packages listed in `requirements.txt`. Python 3.12 is required for installing a few of the packages with the listed versions.

For GPU backends, the command below generates Figure 3:

```
cd out_sycl_pvc/stencil_bench
python3 ../../scripts/plot_gpu_stencil.py \
  sycl_pvc_stencil_502.json \
  sycl_pvc_stencil_device_iterate.json \
  sycl_pvc_stencil_new_tile.json \
  sycl_pvc_stencil_no_stride.json
```

Stream benchmark `Triad` plots are generated with the following command:

```
cd out_sycl_pvc/stream_bench
python3 ../../scripts/plot_gpu_stream.py Triad \
  sycl_pvc_stream_502.json \
  sycl_pvc_stream_device_iterate.json \
  sycl_pvc_stream_new_tile.json \
  sycl_pvc_stream_no_stride.json
```

In the case of CPU backends, the following command configures the benchmarks for a CPU like AMD EPYC 9654 with OpenMP backend.

```
./configure_all.sh --cpu --build-dir build \
  -- -DKokkos_ENABLE_OPENMP=ON -DKokkos_ARCH_NATIVE=ON \
```

```
    -DKokkos_ENABLE_AGGRESSIVE_VECTORIZATION=ON
```

Note that enabling aggressive vectorization is not critical. The number of threads should be passed as the last option of `run_platform.sh`. The figure in Figure 7 was generated by running on 8 threads, with a command as follows:

```
./run_platform.sh --cpu build omp_t8 8
```

If number of threads is not passed, a default value of 1 is used. For the CPU OpenMP backend, the following settings are recommended: `OMP_PROC_BIND=close`, `OMP_PLACES=cores`. It is also recommended to set `OMP_NUM_THREADS` to a value that is less than or equal to what was passed to `run_platform.sh`.

For the above `run_platform.sh` command, the profiles will be written to the folders `out_omp_t8/stencil_bench` and `out_omp_t8/stream_bench`. For CPU benchmarks, `plot_cpu_stencil.py` and `plot_cpu_stream.py` are run in the same manner as it is done for GPU backends, with the appropriate JSON files. The comparisons are between the files ending as `_520.json` and `_auto_vect.json`.

For the above `run_platform.sh` command, the following command generates Figure 7:

```
cd out_omp_t8
python3 ../scripts/plot_cpu_stencil_stream.py \
  stencil_bench/omp_t8_stencil_520.json \
  stencil_bench/omp_t8_stencil_auto_vect.json \
  stream_bench/omp_t8_stream_520.json \
  stream_bench/omp_t8_stream_auto_vect.json \
  Baseline With-Auto-Vect Triad
```

To generate Figure 8, one needs to run the benchmarks for different numbers of threads (1, 2, 4, 8, 16, 32). To start with, the following commands are run: `./run_platform.sh --cpu build omp_t1`, `./run_platform.sh --cpu build omp_t2 2`, ..., `./run_platform.sh --cpu build omp_t32 32`. A file (like `info_file.txt` shown in `plot_cpu_scale.py`) needs to be prepared with the input information. Then, the following command generates the required figure:

```
cd scripts
python3 plot_cpu_scale.py info_file.txt
```