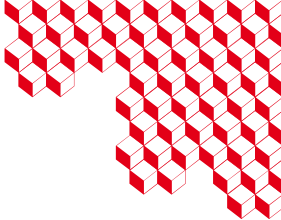


Improving the Efficiency of Kokkos Multi-Dimensional Range Policy for GPUs

Adrien Taberner, CEXA Team

HPSF - Community Summit 2026

February 25, 2026

Table of contents

1. Introduction

Table of contents

What is MDRangePolicy?

- High-level abstraction for iterating over multi-dimensional index spaces
- Works with `parallel_for` and `parallel_reduce` constructs
- Supports N-dimensional spaces (up to 6 dimensions)
- Most intuitive approach for:
  - Multi-dimensional array iterations
  - Stencil computations in scientific applications
  - Every workload with multi-dimensional iteration space

MDRangePolicy Usage Example

You define a policy by specifying the **rank** (number of dimensions) and the **bounds** of the iteration space:

```
Kokkos::MDRangePolicy<Kokkos::Rank<2>> policy_2d({0, 0}, {N, M});
```

You can also control the **iteration order** to match your data layout (row-major or column-major):

```
using Policy_2d = Kokkos::MDRangePolicy<Kokkos::Rank<2, Kokkos::Iterate::Right, Kokkos::Iterate::Right>>;
Policy_2d policy_2d({0, 0}, {N, M});
```

The number of indices of the functor or lambda you pass to `parallel_for` or `parallel_reduce` should match the rank specified in the policy.

MDRangePolicy parallel_for Example

A typical use case:
Applying a stencil computation over a 3D grid. Each thread receives its own `(i, j, k)` indices.

```
using policy_3d = Kokkos::MDRangePolicy<Kokkos::Rank<3>>;

Kokkos::parallel_for("Stencil 3D", policy_3d({1, 1, 1}, {N - 1, M - 1, L - 1}),
  KOKKOS_LAMBDA(const int i, const int j, const int k) {
    // Access multi-dimensional array
    A(i, j, k) = (1.0/7.0) * (B(i + 1, j, k) + B(i - 1, j, k) +
                              B(i, j + 1, k) + B(i, j - 1, k) +
                              B(i, j, k + 1) + B(i, j, k - 1) + B(i, j, k));
  });
```

Figure: 7-point stencil computation in 3D

Performance Concerns with MDRangePolicy

Well-known performance concerns among Kokkos users

AthenaK: an AMR framework with fluid solvers rewritten in Kokkos. Users avoid MDRangePolicy and manually flatten indices for performance.

```
221 // Experiments in K-Athena and Parthenon indicate that 1D-range policy is
222 // generally faster than multidimensional MD-range policy, so the latter is not used.
223 //--------------------------------
224 // 3D loop using Kokkos 1D Range
225 template <typename Function>
226 inline void par_for(/*...*/) {
227   const int nkji = nk * nj * ni;
228   const int nji  = nj * ni;
229   Kokkos::parallel_for(name, Kokkos::RangePolicy<>(exec_space, 0, nkji),
230     KOKKOS_LAMBDA(const int &idx) {
231       int k = (idx)/nji;
232       int j = (idx - k*nji)/ni;
233       int i = (idx - k*nji - j*ni) + il;
234       k += kl;  j += jl;
235       function(k, j, i);
236   });
237 }
```

Figure: Code snippet from AthenaK `src/athena.hpp` (IAS - Astrophysics)

Performance Concerns with MDRangePolicy

Example with simple kernel: $A = A + 2 \times B$

Users' solution: use a flattened index and compute multi-dimensional indices in the kernel. Fine on GPU, where there is plenty of computational power.
Requires writing two different code paths for `LayoutLeft` and `LayoutRight`.

```
KOKKOS_INLINE_FUNCTION
void operator()(const int r) const {
  if constexpr (Layout == Kokkos::LayoutLeft) {
    int i0 = r % N;
    int i1 = r / N;
    A(i0, i1) += 2.0 * B(i0, i1);
  } else {
    int i1 = r % M;
    int i0 = r / M;
    A(i0, i1) += 2.0 * B(i0, i1);
  }
}
```

Performance Concerns with MDRangePolicy

Benchmarks with $M = N = 16384$ on Nvidia A100

Policy	Layout	Time (ms)	Speedup
Flattened index RangePolicy (1D)	Left	4.74	1.0x
	Right	4.74	1.0x
MDRangePolicy (2D)	Left	6.25	0.76x
	Right	6.25	0.76x

Notes

- The flattened RangePolicy is faster than the MDRangePolicy in this example, which is a common observation among users.
- The new MDRangePolicy has the same performance as the flattened RangePolicy for this simple kernel.

Why Optimize MDRangePolicy?

- Growing Kokkos adoption in CEA applications for porting code to GPU (e.g., Gysela, TRUST (CFD))
- Many users report performance issues with the current implementation, not only on GPU backends
- Some fall back to manual flattening as a workaround for better performance
- MDRangePolicy is widely used across applications
- Optimizing this core functionality benefits a large user base

2. Tiling Strategies

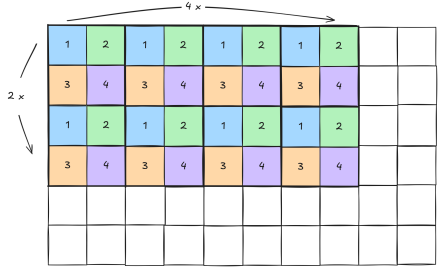
Table of contents

What is Tiling?

What does that mean for devices?

Tiling is a technique that divides the iteration space into smaller blocks (tiles) to improve data locality and parallelism.

- On GPU, you can reuse a thread or a thread block to process multiple elements in the iteration space.
- The work between threads should be strided by the CUDA/HIP block size.
- Requires inner loops. Not all problems benefit from tiling.



Tiling with Kokkos MDRangePolicy

Users can specify custom tile sizes as a third argument, and control the iteration order via template parameters:

```
// Custom tile sizes (third argument)
using Policy_2d = Kokkos::MDRangePolicy<Kokkos::Rank<2>, IterateOuter, IterateInner>;
Policy_2d policy_2d({0, 0}, {N, M}, {TileSizeX, TileSizeY});
```

Kokkos MDRangePolicy

- One WorkItem (thread) is mapped to one element in the multi-dimensional range.
- The tile size parameter sets the thread block dimensions of CUDA/HIP thread blocks.

Important Distinction

Kokkos MDRangePolicy's "tile size" parameter sets the **thread block dimensions**, not algorithmic tiling.

Example: What "Tiling" Looks Like in Kokkos

What People Might Expect
(True algorithmic tiling)

```
// Each thread processes
// a 4 x 2 tile
for (int ty = 0; ty < 4; ty++) {
  for (int tx = 0; tx < 2; tx++) {
    int i = tile_i + ty;
    int j = tile_j + tx;
    result[i][j] = ...;
  }
}
```

What Kokkos Actually Does
(Block size configuration)

```
// Each thread processes
// exactly one element
int i = blockIdx.y * 4
      + threadIdx.y;
int j = blockIdx.x * 2
      + threadIdx.x;
result[i][j] = ...;
```

Summary

The third argument `{4, 2}` sets `blockDim = {4, 2}`, determining how many threads run in parallel per block, **not** how many elements each thread processes.

Tiling with Kokkos MDRangePolicy

For GPU backends, tiling maps directly to CUDA/HIP thread blocks: each block processes one tile of the iteration space. Users can specify custom tile sizes as a third argument.

For policies with rank higher than 3, the dimensions of tiles are packed by pairs into one CUDA/HIP dimension.

Rank	MDRange indices	x dimensions	y dimensions	z dimensions
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

Figure: Mapping of MDRange indices to CUDA/HIP dimensions for different ranks

Default Tiling Strategies

Users can also control the iteration order via template parameters:

```
// Iteration order: IterateOuter (between tiles), IterateInner (within tiles)
Kokkos::Rank<N, IterateOuter, IterateInner>;
// Custom tile sizes (third argument)
using Policy_2d = Kokkos::MDRangePolicy<Kokkos::Rank<2>>;
Policy_2d policy_2d({0, 0}, {N, M}, {TileSizeX, TileSizeY});
```

- IterateOuter: iteration pattern between tiles.
- IterateInner: iteration pattern within tiles.

Important

- `IterateOuter` has no effect on GPU as we cannot control the order of block execution.
- The iteration order should match the data layout for performance.
- For device backends, one WorkItem (thread) is mapped to one element in the multi-dimensional range.

Kokkos MDRangePolicy Tiling

Default Tiling Strategies

Current default tile dimensions in CUDA

Iterate	Rank	Kokkos default tile	Block dimension	Total block size
Left	2	(16, 2)	(16, 2)	32
	3	(16, 2, 2)	(16, 2, 2)	64
	4	(16, 2, 2, 2)	(32, 2, 2)	128
	5	(16, 2, 2, 2, 2)	(32, 4, 2)	256
	6	(16, 2, 2, 2, 2, 1)	(32, 4, 2)	256
Right	2	(2, 16)	(2, 16)	32
	3	(2, 2, 16)	(2, 2, 16)	64
	4	(2, 2, 2, 16)	(4, 2, 16)	128
	5	(2, 2, 2, 2, 16)	(4, 4, 16)	256
	6	(1, 2, 2, 2, 2, 16)	(2, 4, 32)	256

Kokkos MDRangePolicy Tiling

Default Tiling Strategies

Current default tile dimensions in HIP

Iterate	Rank	Kokkos default tile	Block dimension	Total block size
Left	2	(16, 4)	(16, 4)	64
	3	(16, 4, 4)	(16, 4, 4)	256
	4	(16, 4, 4, 1)	(64, 4, 1)	256
	5	(16, 4, 4, 1, 1)	(64, 4, 1)	256
	6	(16, 4, 4, 1, 1, 1)	(64, 4, 1)	256
Right	2	(4, 16)	(4, 16)	64
	3	(4, 4, 16)	(4, 4, 16)	256
	4	(1, 4, 4, 16)	(4, 4, 16)	256
	5	(1, 1, 4, 4, 16)	(1, 16, 16)	256
	6	(1, 1, 1, 4, 4, 16)	(1, 4, 64)	256

3. Performance Analysis and Implementation Challenges

Table of contents

Benchmarks Overview

Stencil Benchmarks

- 2D, 3D, and 4D stencils
- Implemented with different iteration orders
- Evaluate performance across dimensions and configurations

```
// 2D 5-point stencil
A(i, j)  = 0.2 * (B(i, j)
                + B(i + 1, j) + B(i, j + 1)
                + B(i - 1, j) + B(i, j - 1));
```

Stream-like Benchmarks

- Set, copy, scale, add, and triad
- Implemented from 2D to 6D, same data amount across all ranks: 22^6
- Memory-bound kernels to evaluate memory access patterns and bandwidth utilization

```
// Set:    A(i..)  =  scalar
// Copy:   A(i..)  =  B(i..)
// Scale:  A(i..)  =  scalar * B(i..)
// Add:    A(i..)  =  B(i..) + C(i..)
// Triad:  A(i..)  =  B(i..) + scalar * C(i..)
```

Performance Metrics - Old Implementation

Example: Kokkos 5.0.1, CUDA 12.8, A100 with Stream Add benchmark

Rank	Register Usage	Achieved Occupancy %	Memory Throughput %
2	20	15.8	13.5
3	30	28.3	26.9
5	56	46.4	13.3
6	70	34.5	7.3

- High register usage per thread
- Low GPU theoretical occupancy (< 50% on CUDA 2D)
- Memory access patterns not fully coalesced
- Performance variability across dimensions
- Significant overhead compared to hand-written kernels

Previous Implementation Overview

Overview of the old `DeviceIterate` implementation

- Object responsible for computing the multi-dimensional indices in the kernel.
- For each tile, computes the multi-dimensional indices based on the block and thread indices.
- Calls the user functor with the computed indices.

```
// iterate over x blocks
for (int tile_id0 = blockIdx.x; tile_id0 < m_policy.m_tile_end[0]; tile_id0 += gridDim.x) {
  // compute index for dimension 0
  const int offset_0 = tile_id0 * m_policy.m_tile[0] + threadIdx.x + m_policy.m_lower[0];
  // check index for dimension 0 is within range
  if (offset_0 < m_policy.m_upper[0] && threadIdx.x < m_policy.m_tile[0]) {
    // call kernel with computed indices
    Impl::invoke<Tag>(m_func, offset_0, offset_1);
  }
}
```

Previous Implementation Overview

Unpacking of tiles for rank 4 and above.

```
// number of tiles for dimension 0, 1
const int temp0 = m_policy.m_tile_end[0];
const int temp1 = m_policy.m_tile_end[1];

// number of virtual blocks for dimension 0,1
const int numbl0 = Kokkos::min(temp0, m_max_grid_size[0]);
const int numbl1 = Kokkos::min(temp1, m_max_grid_size[0]);

// first virtual block index for dimension 0,1
const int tile_id0 = blockIdx.x % numbl0;
const int tile_id1 = blockIdx.x / numbl0;

// virtual thread index for dimension 0,1
const int thr_id0 = threadIdx.x % m_policy.m_tile[0];
const int thr_id1 = threadIdx.x / m_policy.m_tile[0];
```

Previous Implementation Overview

Unpacking of tiles for rank 4 and above.

```
// iterate over virtual blocks for dimension 0
for (index_type i = tile_id0; i < m_policy.m_tile_end[0]; i += numbl0) {
  // compute index for dimension 0
  const index_type offset_0 = i * m_policy.m_tile[0] + thr_id0 + m_policy.m_lower[0];
  // check index for dimension 0 is within range
  if (offset_0 < m_policy.m_upper[0] && thr_id0 < m_policy.m_tile[0]) {
    // ...
    Impl::invoke<Tag>(m_func, offset_0, offset_1, offset_2, offset_3);
  }
}
```

- One for loop and one check for each dimension
- Some values can be precomputed before the kernel launch

Code Complexity and Profiling Insights

- Tiling is performed by thread blocks: one thread computes one element of the multi-dimensional range.
- A loop is only needed to work around the CUDA/HIP grid size limit.
- Packing and unpacking already exist for MDRangePolicy but could be simplified.
- Performance differences between IterateLeft and IterateRight.

Code Complexity Issues

- High register usage from rank 4 and above
- Unnecessary checks for tile boundaries
- Unnecessary for loops for extending virtual dimensions
- Mathematical functions for index calculations

Profiling Insights

- Register pressure limiting occupancy
- Suboptimal default block sizes for GPU backends
- Inefficient memory access patterns
- Excessive branches in kernel code

4. New Implementation Overview

Table of contents

Optimization Goals

New version of **DeviceIterate**, the object responsible for computing the multi-dimensional indices in the kernel. Minimize the cost and complexity of index computation to leave more resources for the user functor.

Objectives

- No performance difference between LayoutLeft and LayoutRight
- Reduce register pressure $\Leftrightarrow$ Improve GPU occupancy
- Simplify code paths
- Reduce code size and complexity

New Implementation Overview

- The innermost loop always maps to the `x` dimension, which is the most efficient for memory access.
  - The iteration order of the CUDA/HIP grids is fixed, so we need to adapt the mapping of indices to dimensions accordingly.
- The innermost loop is always mapped to the fastest-varying index.
- A **recursive template** generates the nested `for` loops.
- At most **3 nested loops** are used, regardless of the policy rank.
  - These loops help bypass kernel limitations on maximum grid size.
  - Device parallelism is applied to these 3 loops at most.
- Index construction always iterates from `Rank-1` down to `0`, regardless of layout.
- In most cases, the strides (and therefore the `for` loops) are unnecessary since CUDA/HIP APIs already provide large grid sizes.

Dimension Mapping - Example with Rank = 4

Loop order: `RankIdx = 3, 2, 1, 0`

Index	CUDA/HIP Dimension
0, 1	`x` (innermost)
2	`y`
3	`z` (outermost)

- Indices are always constructed from highest rank to lowest
- Multiple indices can be **collapsed** onto a single CUDA/HIP dimension
- The `x` dimension handles the innermost (fastest-varying) indices

Layout Handling: LayoutLeft vs. LayoutRight

The layout determines the **order in which indices are passed** to the functor.
LayoutLeft: leftmost index is fastest-varying:

```
functor(i0, i1, i2, i3)  // i0 maps to x (innermost)
```

LayoutRight: rightmost index is fastest-varying:

```
functor(i3, i2, i1, i0)  // i0 maps to x (innermost)
```

How it works:

- The iteration order over dimensions stays the same internally
- For `LayoutRight`, the **bounds are swapped** before calling `deviceIterate`
- This ensures the functor always receives indices in natural order `(i0, i1, i2, i3)`, while the **innermost loop** corresponds to the correct index for each layout

New Implementation Overview

The recursive template generates the nested loops for all ranks (2 to 6) in a single function.

```
template <unsigned R, typename... Idxs>
void iterate(Idxs... idxs) {
  // Start with the highest rank index (R-1) and compute its index
  if constexpr (Layout == Iterate::Left) {
    iterate<R - 1>(idx, idxs...);  // Pass current index first for LayoutLeft
  } else {
    iterate<R - 1>(idxs..., idx);  // Pass current index last for LayoutRight
  }
}
```

Example of generated loops for Rank 4 with LayoutLeft

```
// Generated loops for Rank = 4 with LayoutLeft
iterate<4>();
iterate<3>(idx_3);
iterate<2>(idx_2, idx_3);
// Unpack 2 indices
iterate<0>(idx_0, idx_1, idx_2, idx_3); // last rank calls the functor
```

New Implementation Overview

Unpacking of tiles for rank 4 and above

```
const int start  = blockIdx.x * blockDim.x + threadIdx.x;
const int stride = blockDim.x * gridDim.x;
for (int idx = start; idx < end; idx += stride) {
    if constexpr (is_packed_index<rankIdx>()) {
      // Unpack two consecutive indices
      constexpr unsigned idx1 = (rankIdx % 2 == 0) ? rankIdx : (rankIdx - 1);
      constexpr unsigned idx2 = (rankIdx % 2 == 0) ? (rankIdx + 1) : rankIdx;

      const int id_1 = idx % m_extent[idx1] + m_lower[idx1];
      const int id_2 = idx / m_extent[idx1] + m_lower[idx2];

      //Call the next iteration with the unpacked indices
      iterate<R - 2>(id_1, id_2, idxs...);
    }
}
```

Performance Metrics - New Implementation

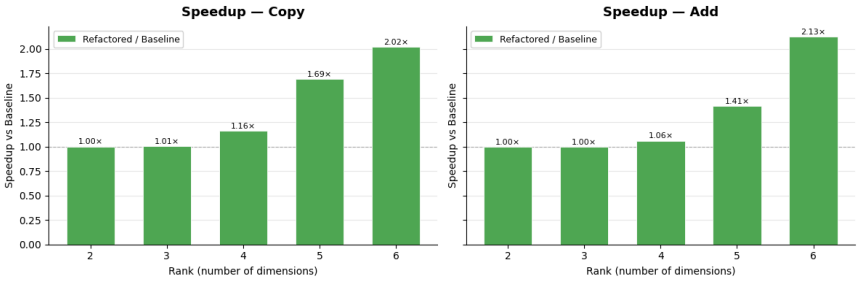
Significant improvements in register usage, occupancy, and memory throughput compared to the old implementation (Kokkos 5.0.1).

Example: New DeviceIterate, CUDA 12.8, A100 with Stream Add benchmark

Rank	Register Usage	Achieved Occupancy %	Memory Throughput %
2	16 - 4	18.8 1.2x	51.7 3.9x
3	21 - 9	87.4 3.7x	88.5 3.3x
5	30 - 26	90.9 1.9x	88.7 6.6x
6	40 - 30	67.1 1.9x	55.8 7.6x

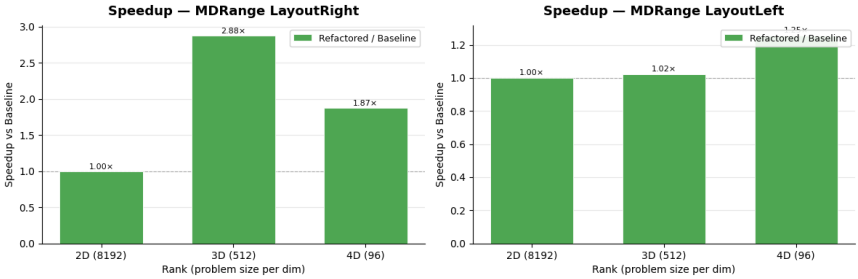
Performance Improvements

NVIDIA A100: CUDA 12.8, tested against Kokkos 5.0.1.



Performance Improvements

NVIDIA A100: CUDA 12.8, tested against Kokkos 5.0.1.



Key Improvements

- Refactored template specialization by ranks
  - One recursive template function for ranks 1 to 6
  - `constexpr` handling of layout (Left / Right)
  - Explicit unpacking of indices
- Reduced register usage through code simplification
- Better memory access patterns
- Highly reduced code length: +293 -1039
- No changes required to user code
- Modernized codebase with cleaner abstractions

5. Finding New Good Default Tile Sizes

Table of contents

Finding New Good Default Tile Sizes

Motivation

Good default block sizes vary between problem types and problem sizes. We need to find a good default with the benchmarks we have.

Strategies

- Evaluated performance across a range of tile sizes, thousands of configurations tested.
- Tested only tiles that range from 32 to 256.
- Selected tile sizes that provided the best overall performance across benchmarks.

Swapping Tile Sizes for LayoutRight

- The tile sizes for LayoutRight are swapped to match the iteration order of the indices.
- For LayoutRight, the rightmost index is the fastest-varying, so it should map to the x dimension.

```
using Policy_2d = Kokkos::MDRangePolicy<Kokkos::Rank<2, Iterate::Right, Iterate::Right>>;
// When user writes this for LayoutRight
Policy_2d policy({0, 0}, {N, M}, {8, 32});
// The block size will be (32, 8) for LayoutRight
```

Users don't need to change anything: the tile sizes are swapped internally to match the iteration order of the indices.

Finding New Good Default Tile Sizes

New default tile dimensions in CUDA

Proposed default tile dimensions

Backend	Rank	Kokkos default tile	New default tile
CUDA	2	(16, 2)	(64, 4)
	3	(16, 2, 2)	(32, 2, 4)
	4	(16, 2, 2, 2)	(16, 4, 1, 4)
	5	(16, 2, 2, 2, 2)	(16, 2, 4, 2, 1)
	6	(16, 2, 2, 2, 2, 1)	(8, 4, 2, 2, 2, 1)
HIP	2	(16, 4)	(64, 4)
	3	(16, 4, 4)	(32, 2, 4)
	4	(16, 4, 4, 1)	(16, 4, 1, 4)
	5	(16, 4, 4, 1, 1)	(16, 4, 2, 2, 1)
	6	(16, 4, 4, 1, 1, 1)	(8, 4, 2, 2, 2, 1)

Figure: New default tile dimensions in CUDA and HIP for LayoutLeft

Finding New Good Default Tile Sizes

Optimizing for GPU Occupancy

Goal: Choose tile sizes that maximize SM occupancy.

Example: NVIDIA A100

- Hardware limits:
  - Max thread blocks per SM: 32
  - Max active threads per SM: 2048
- Block size = 32 threads:
  - Active blocks per SM: 32
  - Active threads per SM: $32 \times 32 = 1024$
  - Theoretical maximum occupancy: 50% (1024 / 2048)

Note

If your kernel uses many registers, smaller block sizes may be necessary to maintain occupancy.

Performance Metrics - New Default Tile Sizes

Better occupancy and memory throughput across benchmarks.
Example: DeviceIterate, CUDA 12.8, A100 with Stream and Stencil benchmarks

Rank	Achieved Occupancy %	Memory Throughput %
2	91.5 4.8x	90.1 1.74x
3	88.1 1.0x	91.2 1.03x
5	90.9 1.0x	88.9 1.01x
6	68.1 1.02x	74.1 1.32x

BM Name	Iterate	Achieved Occupancy %	Memory Throughput %
Stencil 2D (8192)	Left	90.2 5.5x	85.9 2.4x
	Right	90.1 5.6x	86.3 2.5x
Stencil 3D (512)	Left	81.6 0.92x	70.3 1.06x
	Right	83.4 1.14x	70.9 1.02x

Figure: Summary of performance metrics for the new default tile sizes

Performance Improvements - New Default Tile Sizes

NVIDIA A100: CUDA 12.8, speedup with new default tile sizes.

BM name	Rank	Improvement	Time Old (ms)	Time New (ms)
Stream Add	2	-24.1%	4.42	3.36
	3	-0.3%	3.41	3.4
	4	-1.4%	3.4	3.35
	5	-0.3%	3.35	3.34
	6	-15.8%	4.17	3.51

BM name	Iterate	Improvement	Time Old (ms)	Time New (ms)
Stencil 2D (8192)	Left	-48.7%	1.56	0.8
	Right	-48.7%	1.56	0.8
Stencil 3D (512)	Left	0.0%	1.93	1.93
	Right	-0.5%	1.86	1.85
Stencil 4D (96)	Left	-10.1%	1.69	1.52
	Right	-9.6%	1.67	1.51

Overall Impact: Before vs. After

Combined effect of refactored DeviceIterate + new default tile sizes

Metric	Rank 2	Rank 3	Rank 5	Rank 6
Register Usage	20 → 16	30 → 21	56 → 30	70 → 40
Achieved Occupancy %	15.8 → 91.5	28.3 → 88.1	46.4 → 90.9	34.5 → 68.1
Memory Throughput %	13.5 → 90.1	26.9 → 91.2	13.3 → 88.9	7.3 → 74.1

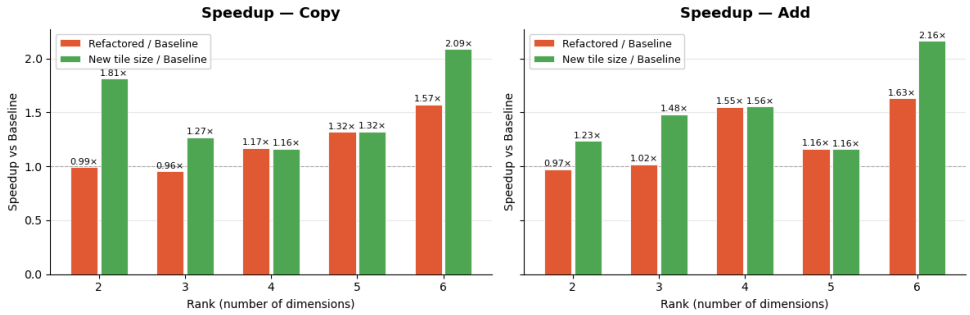
Benchmark	Time Old (ms)	Time New (ms)
Stream Add Rank 6	8.87	3.51 (2.5x)
Stencil 3D Right (512)	5.05	1.85 (2.7x)
Stencil 2D Left (8192)	1.56	0.80 (1.95x)
Stencil 4D Right (96)	3.14	1.51 (2.1x)

Figure: Nvidia A100, CUDA 12.8, old = Kokkos 5.0.1, new = refactored DeviceIterate + new tiles

Overall Impact: Before vs. After

Combined effect of refactored DeviceIterate + new default tile sizes

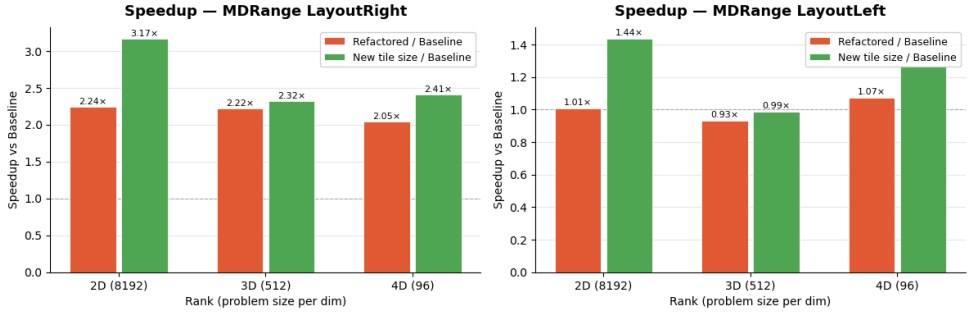
AMD MI250X, ROCm 6.3.3



Overall Impact: Before vs. After

Combined effect of refactored DeviceIterate + new default tile sizes

AMD MI250X, ROCm 6.3.3



6. Conclusion

Table of contents

Summary

- Successfully optimized core Kokkos functionality
- Fixed LayoutRight performance issues
- Measurable performance improvements (1.0x - 2.7x depending on rank and benchmark)
- Full backward compatibility maintained
- Cleaner, more maintainable codebase: +293 -1039 lines
- Benefits the entire Kokkos user community

Lessons Learned

- **Register pressure was the main bottleneck:** the old implementation used up to 70 registers per thread at Rank 6, severely limiting occupancy. Simplifying the index computation logic had a large impact, as expected.
- **Theoretical occupancy matters just as much:** for Rank 2, the refactored code alone showed no clear improvement. The gains came almost entirely from better default tile sizes (block size 32 $\rightarrow$ 256).
- **Abstractions and performance are not mutually exclusive:** the new implementation is both simpler (one recursive template vs. rank-specialized code) and faster.
- **Exhaustive search risks over-tuning to specific benchmarks:** testing thousands of tile configurations was necessary, but the results are tuned to our benchmark suite. We still lack a wider variety of real-world workloads to validate generality.

Status and How to Try It

Current Status

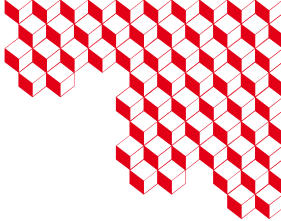
- Refactored `DeviceIterate`: merged into Kokkos
- New default tile sizes: under review / in progress

How to Follow or Contribute

- Kokkos GitHub: https://github.com/kokkos/kokkos
- DeviceIterate refactor: PR #8638
- New default tile sizes: PR #8731
- Feedback and benchmarks on your own applications are welcome

Future Work

1. No grid stride: pure CUDA/HIP indexing through the iteration space
   - This would further reduce register usage and simplify code paths
   - Draft mdrange-no-stride
2. Static unrolling of loops for higher instruction count
3. Dynamic tile size selection for small problem sizes
4. User-guided performance hints
5. `parallel_reduce` benchmarks and possible optimizations



Thank you for your attention

Questions ?

CEA SACLAY
91 191 Gif-sur-Yvette Cedex
France
adrien.taberner@cea.fr