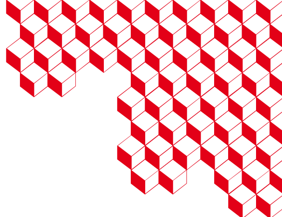


Multi-Dimensional Range Policy Performance Concerns and Improvements

Adrien Taberner, CEXA Team

Kokkos Tea Time June 2026

June 17, 2026

Table of contents

1. **Introduction**
2. **Tiling With MDRangePolicy**
3. **Performance Analysis and Implementation Challenges**
4. **New Implementation Overview**
5. **Finding New Default Tile Sizes**
6. **Benchmark Results and Conclusion**

1. Introduction

Table of contents

1. **Introduction**
2. Tiling With MDRangePolicy
3. Performance Analysis and Implementation Challenges
4. New Implementation Overview
5. Finding New Default Tile Sizes
6. Benchmark Results and Conclusion

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)
- Team-level MDRangePolicy (`TeamThreadMDRange`, `TeamVectorMDRange`) shares no code with MDRangePolicy.

Utilization of MDRangePolicy:

```
// 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});
```

Performance Concerns with MDRangePolicy

Well-known performance concerns among Kokkos users

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

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

```
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   /*...*/
228   const int nkji = nk * nj * ni;
229   const int nji  = nj * ni;
230   Kokkos::parallel_for(name, Kokkos::RangePolicy<>(exec_space, 0, nkji),
231     KOKKOS_LAMBDA(const int &idx) {
232       int k = (idx)/nji;
233       int j = (idx - k*nji)/ni;
234       int i = (idx - k*nji - j*ni) + il;
235       k += kl;  j += jl;
236       function(k, j, i);
237   });
238 }
```

2.
Tiling With MDRangePolicy

Table of contents

1. Introduction

2. **Tiling With MDRangePolicy**
   - CPU Tiling
   - GPU Tiling

3. Performance Analysis and Implementation Challenges

4. New Implementation Overview

5. Finding New Default Tile Sizes

6. Benchmark Results and Conclusion

Table of contents

1. Introduction
2. Tiling With MDRangePolicy
   - CPU Tiling
   - GPU Tiling
3. Performance Analysis and Implementation Challenges
4. New Implementation Overview
5. Finding New Default Tile Sizes
6. Benchmark Results and Conclusion

MDRangePolicy - CPU Tiling

Motivation

Tiling is a technique that divides the iteration space into smaller blocks (tiles) to improve data locality and parallelism. Kokkos `MDRangePolicy` provides built-in support for tiling.

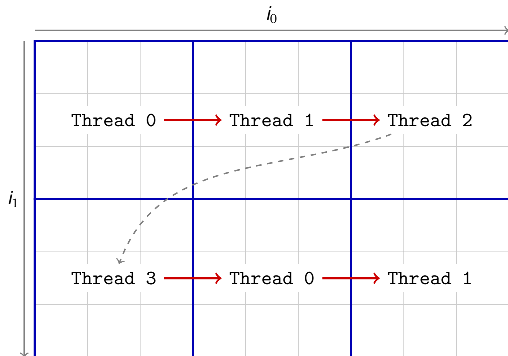
Examples with CPU backends:

- Iteration space: Lower bounds `{0, 0}`, Upper bounds `{9, 6}`
- Tile size: `{3, 3}`
- Number of threads: 4
- Memory layout: `LayoutRight`

```
Kokkos::MDRangePolicy<2>({0,0}, {9,6}, {3,3});
```

MDRangePolicy - CPU Tiling

Outer direction: Right

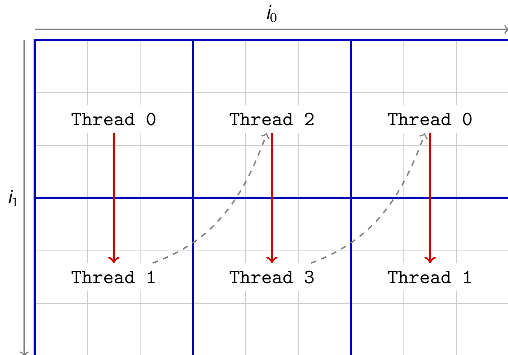


- The outer direction controls the order of tiles assigned to threads.

```
Kokkos::MDRangePolicy<Kokkos::Rank<2,
  Kokkos::Iterate::Right, Kokkos::Iterate::Default>>
  ({0,0}, {9,6}, {3,3});
```

MDRangePolicy – CPU Tiling

Outer direction: Left

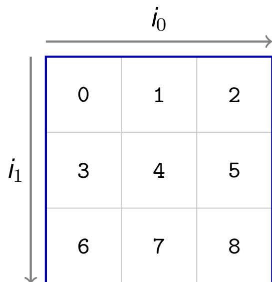


- The outer direction controls the order of tiles assigned to threads.

```
Kokkos::MDRangePolicy<Kokkos::Rank<2,
  Kokkos::Iterate::Left, Kokkos::Iterate::Default>>
  ({0,0}, {9,6}, {3,3});
```

MDRangePolicy - CPU Tiling

Inner direction: Right

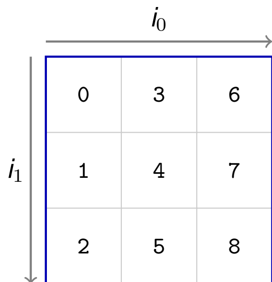



```
// inner direction : right
for (int i = tile_i; i < tile_i + 3; i++) {
  for (int j = tile_j; j < tile_j + 3; j++) {
    view(i,j) = ...;
  }
}
```

- The inner direction controls the order of iterations within each tile.
- For CPU backends, the iteration order can be optimized for cache performance by matching the data layout (row-major or column-major).

MDRangePolicy - CPU Tiling

Inner direction: Left




```
// inner direction : left
for (int j = tile_j; j < tile_j + 3; j++) {
  for (int i = tile_i; i < tile_i + 3; i++) {
    view(i,j) = ...;
  }
}
```

- The inner direction controls the order of iterations within each tile.
- For CPU backends, the iteration order can be optimized for cache performance by matching the data layout (row-major or column-major).

Table of contents

1. Introduction
2. **Tiling With MDRangePolicy**
   CPU Tiling
   GPU Tiling
3. Performance Analysis and Implementation Challenges
4. New Implementation Overview
5. Finding New Default Tile Sizes
6. Benchmark Results and Conclusion

MDRangePolicy - GPU Tiling

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.
- Tile size limited by the maximum number of threads per block.

For policies with rank higher than 3, the indices are packed in pairs into one CUDA/HIP dimension.

Rank	MDRange indices	x dimension	y dimension	z dimension
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

Default Tiling Strategies

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

- It's not the iteration order that should match but the hardware thread id.
- `IterateInner` has no meaning on GPU as the tile is executed in parallel.
- The thread block can be optimized to match the data layout for coalesced access.

3.
Performance Analysis and Im-
plementation Challenges

Table of contents

1. Introduction
2. Tiling With MDRangePolicy
3. **Performance Analysis and Implementation Challenges**
4. New Implementation Overview
5. Finding New Default Tile Sizes
6. Benchmark Results and Conclusion

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 amount of data 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..)
```

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 occurs in the kernel for policies with 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]);

// 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];
```

Some values can be precomputed before the kernel launch

Previous Implementation Overview

Unpacking occurs in the kernel for policies with 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]) {
    // Same thing for dimension 1
    for (index_type i = tile_id1; i < m_policy.m_tile_end[1]; i += numbl1) {
      const index_type offset_1 = i * m_policy.m_tile[1] + thr_id1 + m_policy.m_lower[1];
      if (offset_1 < m_policy.m_upper[1] && thr_id1 < m_policy.m_tile[1]) {
        // ...
        Impl::invoke<Tag>(m_func, offset_0, offset_1, offset_2, offset_3);
      }
    }
  }
}
```

One nested for loop plus one if statement for each dimension

Performance Metrics – Old Implementation

Example: Kokkos 5.0.2, CUDA 12.8, H100 with Stream Add benchmark

Rank	Register Usage	Achieved Occupancy %	Memory Throughput %
2	20	17.0	37.9
3	30	51.5	75.9
4	44	54.8	55.7
5	58	44.5	36.5
6	76	33.9	19.7

- High register usage per thread
- Low measured GPU occupancy (< 50%)
- Performance variability across dimensions

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

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

1. Introduction
2. Tiling With MDRangePolicy
3. Performance Analysis and Implementation Challenges
4. **New Implementation Overview**
5. Finding New Default Tile Sizes
6. Benchmark Results and Conclusion

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 (nested loops, if statements, etc.).
- 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.
- A **recursive template** generates the nested `for` loops and index computations.
- 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` (inclusive), regardless of layout.
- In most cases, the strides (and therefore the `for` loops) are unnecessary since CUDA/HIP APIs already provide large grid sizes.

Unlock GPU Performance: Global Memory Access in CUDA

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 **combined** 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 logic for packed indices

```
const int start  = blockIdx.x * blockDim.x + threadIdx.x;
const int stride = blockDim.x * gridDim.x;
const int end    = m_extent[rankIdx1] * m_extent[rankIdx2];  // extent = tile_size * num_tiles
for (int idx = start; idx < end; idx += stride) {
  if constexpr (is_packed_index<rankIdx>()) {
    // Unpack two consecutive indices
    constexpr unsigned rankIdx1 = (rankIdx % 2 == 0) ? rankIdx : (rankIdx - 1);
    constexpr unsigned rankIdx2 = (rankIdx % 2 == 0) ? (rankIdx + 1) : rankIdx;

    const int id_1 = idx % m_extent[rankIdx1] + m_lower[rankIdx1];
    const int id_2 = idx / m_extent[rankIdx1] + m_lower[rankIdx2];

    //Call the next iteration with the unpacked indices
    if (id_1 < m_upper[rankIdx1] && id_2 < m_upper[rankIdx2]) {
      iterate<R - 2>(id_1, id_2, idxs...);
    }
  }
}
```

Performance Metrics

New DeviceIterate kernel

Example: CUDA 12.8, H100 with Stream Add benchmark

Rank	Register Usage	Achieved Occupancy %	Memory Throughput %
2	20 0	14.5 −2.5%	38.1 1.0x
3	30 0	41.1 −10.4%	75.9 1.0x
4	30 −14	84.8 +30%	89.3 1.6x
5	32 −26	80.8 +36%	84.9 2.3x
6	40 −36	65.6 +31%	42.7 2.1x

- Reduced register usage per thread for higher ranks
- Increased memory throughput for higher ranks

New Implementation Overview

DeviceIterate without grid stride loop

Grid stride loops

- Common pattern for writing flexible kernels.
- Extends CUDA/HIP grid/block size limitations.
- Compile two kernels, choose at runtime which one to use.

```
void saxpy(int n, float a, float *x, float *y) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < n; i += blockDim.x * gridDim.x) {
      y[i] = a * x[i] + y[i];
  }
}
```

Write Flexible Kernels with Grid-Stride Loops

New Implementation Overview

DeviceIterate without grid stride loop

```
const int thIdx = blockIdx.x * blockDim.x + threadIdx.x;
if constexpr (is_packed_index<rankIdx>()) {
  // Unpack two consecutive indices
  constexpr unsigned thIdx1 = (rankIdx % 2 == 0) ? rankIdx : (rankIdx - 1);
  constexpr unsigned thIdx2 = (rankIdx % 2 == 0) ? (rankIdx + 1) : rankIdx;

  const int id_1 = idx % m_extent[thIdx1] + m_lower[thIdx1];
  const int id_2 = idx / m_extent[thIdx1] + m_lower[thIdx2];
  //Call the next iteration with the unpacked indices
  iterate<R - 2>(id_1, id_2, idxs...);
}
/*...*/
// Check before calling the functor, if the index is within the bounds of the iteration space.
template <size_t... R, typename... Idxs>
bool check_bounds(std::index_sequence<R...>, Idxs... idxs) const {
  if constexpr (IterateDir == Iterate::Left) {
    return ((idxs < static_cast<index_type>(m_upper[R])) && ...);
  } else {
    return ((idxs < static_cast<index_type>(m_upper[Rank - 1 - R])) && ...);
  }
}
```

Performance Metrics

DeviceIterate without grid stride loop

Example: CUDA 12.8, H100 with Stream Add benchmark

Rank	Register Usage	Branch Instructions $(\times 10^6)$	Memory Throughput %
2	16 -4	17.9 -50%	92.4 +0.18%
3	16 -14	17.9 -66%	89.7 -0.7%
4	18 -12	23.8 -68%	87.8 +1.7%
5	20 -12	29.8 -61%	89.4 +6.2%
6	26 -14	35.8 -57%	64.8 +14.2%

- Little improvement in memory bandwidth.
- Fewer branch instructions and lower register usage.

5. Finding New Default Tile Sizes

Table of contents

1. Introduction
2. Tiling With MDRangePolicy
3. Performance Analysis and Implementation Challenges
4. New Implementation Overview
5. **Finding New Default Tile Sizes**
6. Benchmark Results and Conclusion

Finding New Default Tile Sizes

Discrepancies between Left and Right Iterate

- Collapse tile sizes the same way as indices.
- Packing and unpacking happen in the same order for both Left and Right Iterate.
- Tile sizes are symmetric between Left and Right Iterate.

Rank	Iterate	Kokkos 5.0.x Tile	Cuda Thread Block
4	Left	(16,2,2,2)	(32,2,2)
	Right	(2,2,2,16)	(4,2,16)
5	Left	(16,2,2,2,2)	(32,4,2)
	Right	(2,2,2,2,16)	(4,4,16)
6	Left	(16,2,2,2,2,1)	(32,4,2)
	Right	(1,2,2,2,2,16)	(2,4,32)

Finding New Default Tile Sizes

CUDA rank 2 problem

Low Occupancy problem: Increase default tile size to allow full utilization of the SM.

CUDA backend Default Tile Sizes for Rank 2 (16, 2)

- 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)

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.
- Tile sizes are packed like indices

```
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 Default Tile Sizes

New default tile dimensions

Backend	Rank	Kokkos 5.0.0	Kokkos 5.1.0
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)

Performance Metrics

New default tile sizes

Example: CUDA 12.8, H100 with Stream and Stencil benchmarks

Rank	Achieved Occupancy %	Memory Throughput %
2	80.0 5.6x	92.1 2.4x
3	79.2 1.9x	90.3 1.2x

BM Name	Iterate	Achieved Occupancy %	Memory Throughput %
Stencil 2D (10240)	Left	80.6 5.7x	79.9 3.0x
	Right	80.7 5.6x	79.6 3.0x
Stencil 3D (512)	Left	78.4 1.3x	65.5 1.08x
	Right	82.2 2.1x	66.9 1.1x

- Better occupancy and memory throughput.

6.
Benchmark Results and Con-
clusion

Table of contents

1. Introduction
2. Tiling With MDRangePolicy
3. Performance Analysis and Implementation Challenges
4. New Implementation Overview
5. Finding New Default Tile Sizes
6. **Benchmark Results and Conclusion**

Overall Impact: Before vs. After

From Kokkos 5.0.2 to Kokkos 5.2.0

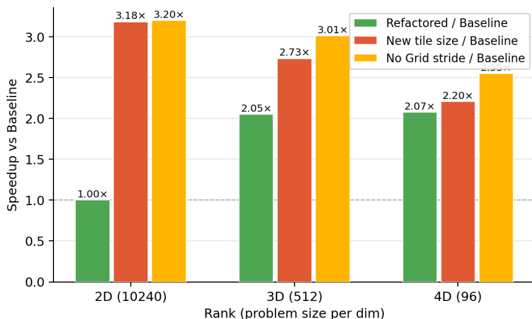
Metric	**Rank 2**	**Rank 3**	**Rank 5**	**Rank 6**
Register Usage	20 → 16	30 → 16	58 → 18	76 → 26
Achieved Occupancy %	17.4 → 79.6	51.5 → 76.4	44.5 → 79.5	33.9 → 87.9
Memory Throughput %	37.9 → 92.4	75.9 → 89.7	36.5 → 87.7	19.7 → 65.8

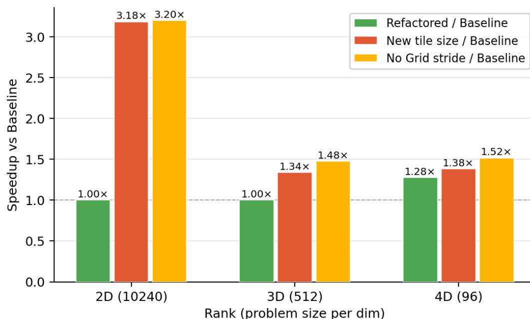
BM Name	Iterate	Achieved Occupancy %	Memory Throughput %
Stencil 2D (10240)	Left	15.8 → 80.1	26.4 → 80.3
	Right	16.6 → 80.2	25.4 → 79.9
Stencil 3D (512)	Left	79.2 → 80.6	60.4 → 74.8
	Right	58.2 → 79.6	42.3 → 74.5

Performance Improvements

Stencil benchmark

NVIDIA H100, CUDA 12.8

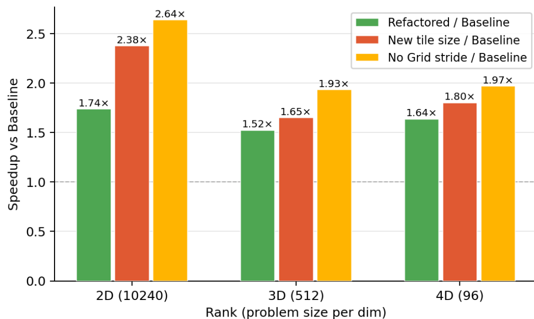


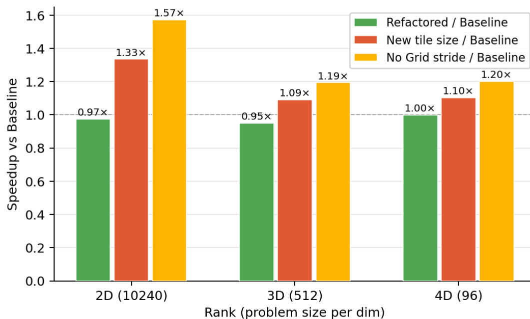


Performance Improvements

Stencil benchmark

AMD MI300A, ROCm 7.2.0

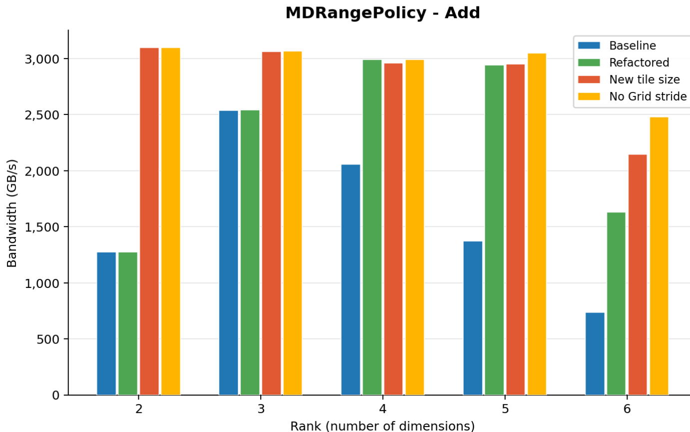




Performance Improvements

Stream benchmark

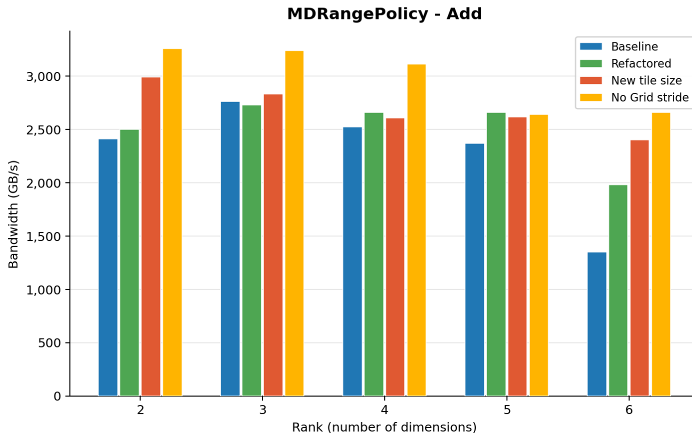
NVIDIA H100, CUDA 12.8



Performance Improvements

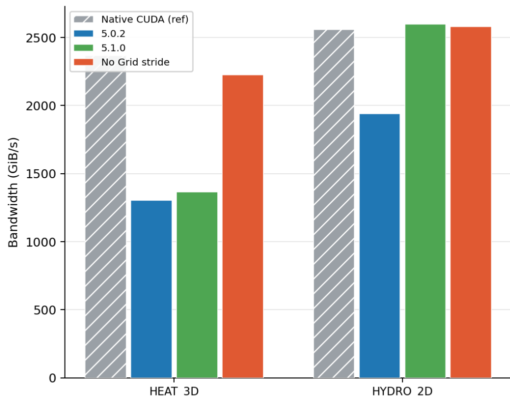
Stream benchmark

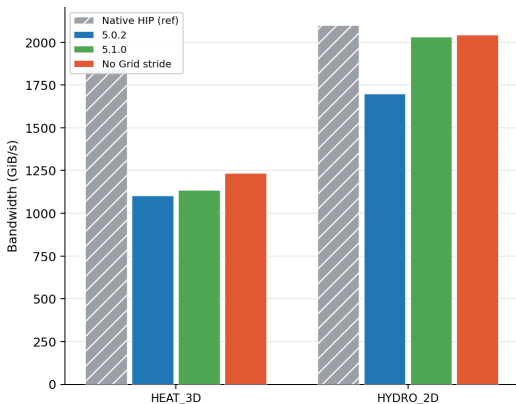
AMD MI300A, ROCm 7.2.0



Performance Improvements

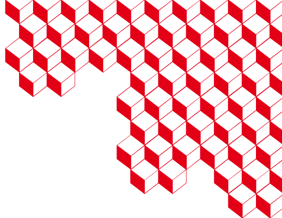
RajaPerf





Summary

- Fixed LayoutRight performance issues, register pressure issues, and low occupancy.
- Measurable performance improvements (1.07x - 3.2x depending on the benchmark)
- No user code change required
- Clearer intent in the implementation.
- Cleaner, more maintainable codebase.



Thank you for your attention

Questions ?

CEA SACLAY
91191 Gif-sur-Yvette Cedex
France
adrien.taberner@cea.fr