SAND2019-14348PE

Sandia National Laboratories

mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards

D. S. Hollman, Bryce Adelstein Lelbach, H. Carter Edwards, Mark Hoemmen, Daniel Sunderland, and Christian R. Trott

NNSA U.S. DEPARTMENT OF ENERGY

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidary of Honeywell International Inc., for the U. S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This document is SAND #2019-XXXX C.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Follow along!

dsh.fyi/TODO

Disclaimer

Disclaimer

This presentation contains some C++ source code.

Outline

Outline

- Introduction

Outline

- Introduction
- Design Overview

Outline

- Introduction
- Design Overview
  - Shape and Indexing

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`
  - Layout Customization

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`
  - Layout Customization
  - Accessor Customization

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`
  - Layout Customization
  - Accessor Customization
- Benchmarks

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`
  - Layout Customization
  - Accessor Customization
- Benchmarks
  - Overview

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`
  - Layout Customization
  - Accessor Customization
- Benchmarks
  - Overview
  - Matrix Vector Multiply

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`
  - Layout Customization
  - Accessor Customization
- Benchmarks
  - Overview
  - Matrix Vector Multiply
  - Subspan Abstraction Overhead

Outline

- Introduction
- Design Overview
  - Shape and Indexing
  - `subspan`
  - Layout Customization
  - Accessor Customization
- Benchmarks
  - Overview
  - Matrix Vector Multiply
  - Subspan Abstraction Overhead
- Concluding Remarks

Introduction

Introduction

- `mdspan` is a non-owning multidimentional array view for C++23.

Introduction

- `mdspan` is a non-owning multidimentional array view for C++23.
- Proposal: P0009R9, http://wg21.link/P0009R9

Introduction

- `mdspan` is a non-owning multidimentional array view for C++23.
- Proposal: P0009R9, http://wg21.link/P0009R9
  - First submitted to Autumn 2015 meeting

Introduction

- `mdspan` is a non-owning multidimentional array view for C++23.
- Proposal: P0009R9, http://wg21.link/P0009R9
  - First submitted to Autumn 2015 meeting
  - Design approved for C++23

Introduction

- `mdspan` is a non-owning multidimentional array view for C++23.
- Proposal: P0009R9, http://wg21.link/P0009R9
  - First submitted to Autumn 2015 meeting
  - Design approved for C++23
  - Currently under wording review

Introduction

- `mdspan` is a non-owning multidimentional array view for C++23.
- Proposal: P0009R9, http://wg21.link/P0009R9
  - First submitted to Autumn 2015 meeting
  - Design approved for C++23
  - Currently under wording review
- Support for arbitrary (rectangular) shapes, mixing of static and runtime extents, layout customization, and memory access customization

Design Overview

Basic Usage

```
void some_function(double* data, float* data2) {

  // std::dynamic_extent means it's a dimension given at runtime
  auto my_matrix = mdspan<double, dynamic_extent, dynamic_extent>(data, 20, 40);

  // runtime and compile-time dimensions can be mixed
  auto other_matrix = mdspan<float, 20, dynamic_extent>(data2, 40);

  /* ... */

  my_matrix(17, 34) = 3.14;
  other_matrix(0, 12) = my_matrix(17, 34);
}
```

Basic Usage

```cpp
void some_function(double* data, float* data2) {

  // std::dynamic_extent means it's a dimension given at runtime
  auto my_matrix = mdspan<double, dynamic_extent, dynamic_extent>(data, 20, 40);

  // runtime and compile-time dimensions can be mixed
  auto other_matrix = mdspan<float, 20, dynamic_extent>(data2, 40);

  /* ... */

  my_matrix(17, 34) = 3.14;
  other_matrix(0, 12) = my_matrix(17, 34);
}
```

View **`data`** as a 20x40 matrix of doubles (runtime extents)

Basic Usage

```
void some_function(double* data, float* data2) {

  // std::dynamic_extent means it's a dimension given at runtime
  auto my_matrix = mdspan<double, dynamic_extent, dynamic_extent>(data, 20, 40);

  // runtime and compile-time dimensions can be mixed
  auto other_matrix = mdspan<float, 20, dynamic_extent>(data2, 40);

  /* ... */

  my_matrix(17, 34) = 3.14;
  other_matrix(0, 12) = my_matrix(17, 34);
}
```

Mixed compile-time and runtime dimensions

Basic Usage

```
void some_function(double* data, float* data2) {

  // std::dynamic_extent means it's a dimension given at runtime
  auto my_matrix = mdspan<double, dynamic_extent, dynamic_extent>(data, 20, 40);

  // runtime and compile-time dimensions can be mixed
  auto other_matrix = mdspan<float, 20, dynamic_extent>(data2, 40);

  /* ... */

  my_matrix(17, 34) = 3.14;
  other_matrix(0, 12) = my_matrix(17, 34);
}
```

Access to values uses the parenthesis operator

Basic Usage

```
void some_function(double* data, float* data2) {

  // std::dynamic_extent means it's a dimension given at runtime
  auto my_matrix = mdspan<double, dynamic_extent, dynamic_extent>(data, 20, 40);

  // runtime and compile-time dimensions can be mixed
  auto other_matrix = mdspan<float, 20, dynamic_extent>(data2, 40);

  /* ... */

  my_matrix(17, 34) = 3.14;
  other_matrix(0, 12) = my_matrix(17, 34);
}
```

mdspan is short for basic_mdspan

```
template <typename T, ptrdiff_t... Exts>
using mdspan = basic_mdspan<T, extents<Exts...>>;
```

`mdspan` is short for `basic_mdspan`

```
template <typename T, ptrdiff_t... Exts>
using mdspan = basic_mdspan<T, extents<Exts...>>;
```

- `mdspan` is just an alias for `basic_mdspan` (just like `string` is an alias for `basic_string`)

`mdspan` is short for `basic_mdspan`

```
template <typename T, ptrdiff_t... Exts>
using mdspan = basic_mdspan<T, extents<Exts...>>;
```

- `mdspan` is just an alias for `basic_mdspan` (just like `string` is an alias for `basic_string`)
- `std::extents` is a class template that expresses the shape of the `mdspan`.

subspan

```
auto s = mdspan<double, 3, 4, 5>(data);
s(1, 2, 3) = 2.78;
auto s2 = subspan(s, 1, pair{1, 3}, all);
assert(s2(1, 3) == s(1, 2, 3));
```

subspan

```
auto s = mdspan<double, 3, 4, 5>(data);
s(1, 2, 3) = 2.78;
auto s2 = subspan(s, 1, pair{1, 3}, all);
assert(s2(1, 3) == s(1, 2, 3));
```

- Supports three different types of slices:

subspan

```
auto s = mdspan<double, 3, 4, 5>(data);
s(1, 2, 3) = 2.78;
auto s2 = subspan(s, 1, pair{1, 3}, all);
assert(s2(1, 3) == s(1, 2, 3));
```

- Supports three different types of slices:
  - A single index

subspan

```
auto s = mdspan<double, 3, 4, 5>(data);
s(1, 2, 3) = 2.78;
auto s2 = subspan(s, 1, pair{1, 3}, all);
assert(s2(1, 3) == s(1, 2, 3));
```

- Supports three different types of slices:
  - A single index
  - A pair of begin and end

subspan

```
auto s = mdspan<double, 3, 4, 5>(data);
s(1, 2, 3) = 2.78;
auto s2 = subspan(s, 1, pair{1, 3}, all);
assert(s2(1, 3) == s(1, 2, 3));
```

- Supports three different types of slices:
  - A single index
  - A pair of begin and end
  - The `all` slice

Layout Customization

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)
  - `layout_right` (C ordering)

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)
  - `layout_right` (C ordering)
  - `layout_stride` (regularly strided dimensions)

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)
  - `layout_right` (C ordering)
  - `layout_stride` (regularly strided dimensions)
- The customization point is flexible enough to support things like

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)
  - `layout_right` (C ordering)
  - `layout_stride` (regularly strided dimensions)
- The customization point is flexible enough to support things like
  - tiled layouts

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)
  - `layout_right` (C ordering)
  - `layout_stride` (regularly strided dimensions)
- The customization point is flexible enough to support things like
  - tiled layouts
  - various forms of symmetric layouts

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)
  - `layout_right` (C ordering)
  - `layout_stride` (regularly strided dimensions)
- The customization point is flexible enough to support things like
  - tiled layouts
  - various forms of symmetric layouts
  - sparse layouts

Layout Customization

- The optional third template parameter to `basic_mdspan` is a `LayoutPolicy`
- `LayoutPolicy` is a customization point that lets the user control how multi-indices are translated into memory offsets.
- The proposal provides three layout policies:
  - `layout_left` (FORTRAN ordering)
  - `layout_right` (C ordering)
  - `layout_stride` (regularly strided dimensions)
- The customization point is flexible enough to support things like
  - tiled layouts
  - various forms of symmetric layouts
  - sparse layouts
  - compressed layouts (with the help of an `AccessorPolicy`)

Accessor Customization

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`
  - The pointer type through which access occurs

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`
  - The pointer type through which access occurs
  - A function for converting a pointer and an offset into a reference

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`
  - The pointer type through which access occurs
  - A function for converting a pointer and an offset into a reference
- With these tools, you can write accessors that do things like:

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`
  - The pointer type through which access occurs
  - A function for converting a pointer and an offset into a reference
- With these tools, you can write accessors that do things like:
  - Expose non-aliasing semantics (i.e., like `restrict` in C)

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`
  - The pointer type through which access occurs
  - A function for converting a pointer and an offset into a reference
- With these tools, you can write accessors that do things like:
  - Expose non-aliasing semantics (i.e., like `restrict` in C)
  - Access remote memory

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`
  - The pointer type through which access occurs
  - A function for converting a pointer and an offset into a reference
- With these tools, you can write accessors that do things like:
  - Expose non-aliasing semantics (i.e., like `restrict` in C)
  - Access remote memory
  - Access data stored in a compressed format of some sort

Accessor Customization

- The optional fourth template parameter to `basic_mdspan` is an `Accessor`
- The `Accessor` customization point provides:
  - The reference type to be returned by `basic_mdspan::operator()`
  - The pointer type through which access occurs
  - A function for converting a pointer and an offset into a reference
- With these tools, you can write accessors that do things like:
  - Expose non-aliasing semantics (i.e., like `restrict` in C)
  - Access remote memory
  - Access data stored in a compressed format of some sort
  - Access data atomically (using P0019, `atomic_ref`)

Benchmarks

Sum3D

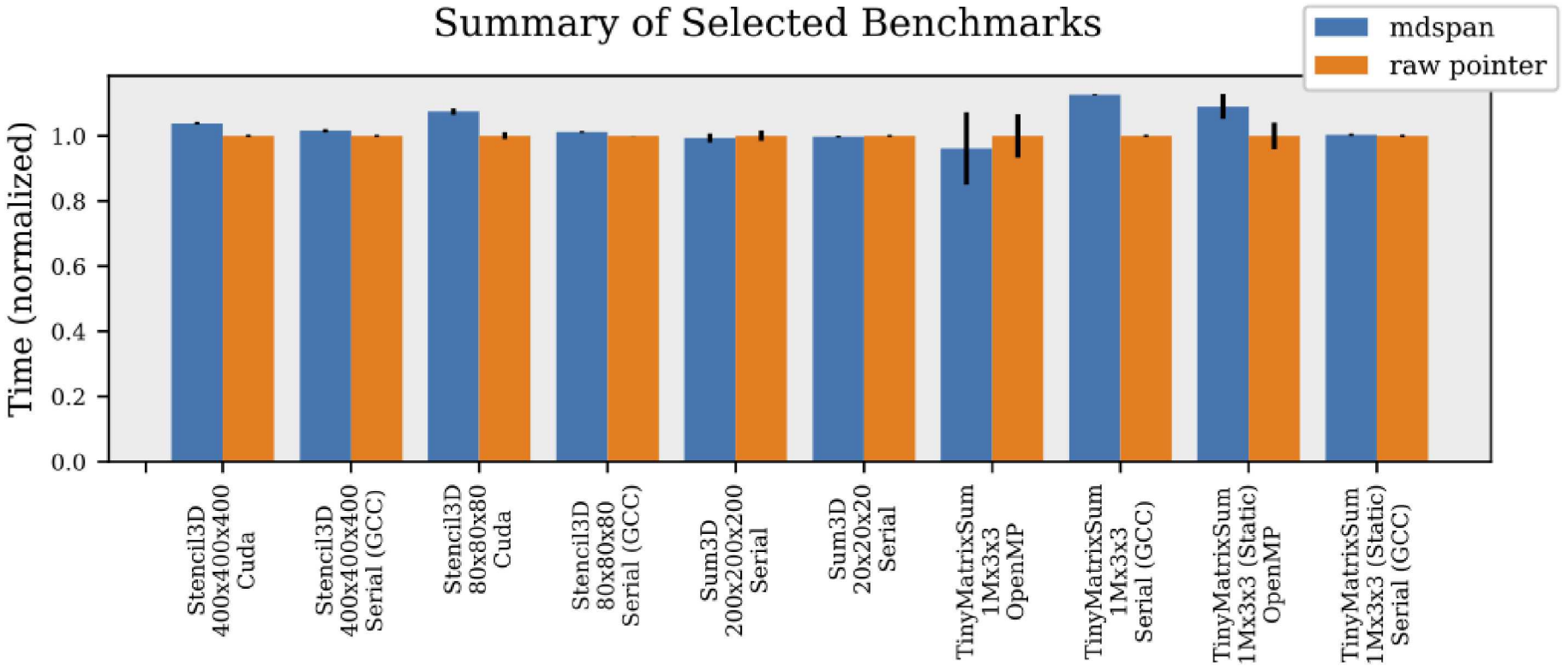
```
for(ptrdiff_t i = 0; i < s.extent(0); ++i) {
  for (ptrdiff_t j = 0; j < s.extent(1); ++j) {
    for (ptrdiff_t k = 0; k < s.extent(2); ++k) {
      sum += s(i, j, k);
    }
  }
}
```

Sum3D

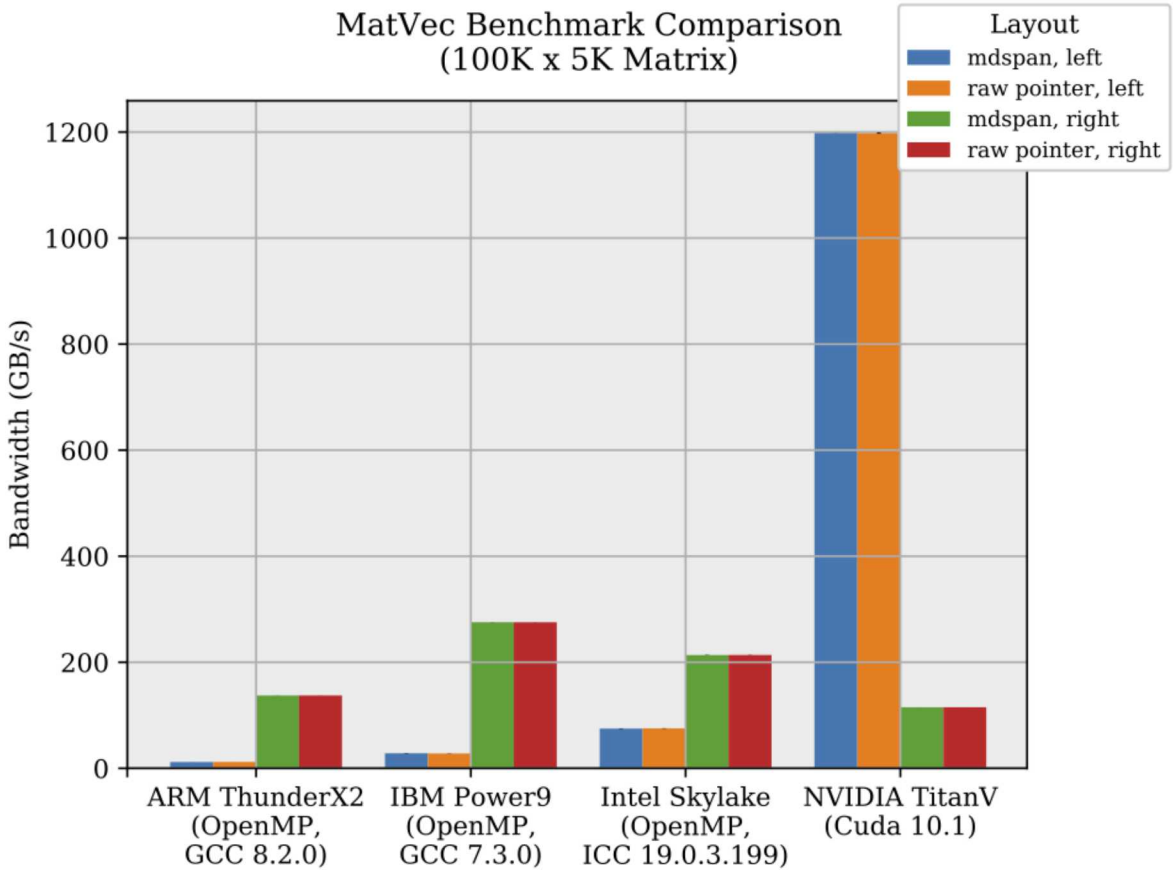
```
for(ptrdiff_t i = 0; i < s.extent(0); ++i) {
  for (ptrdiff_t j = 0; j < s.extent(1); ++j) {
    for (ptrdiff_t k = 0; k < s.extent(2); ++k) {
      sum += s(i, j, k);
    }
  }
}
```

```
for(ptrdiff_t i = 0; i < x; ++i) {
  for (ptrdiff_t j = 0; j < y; ++j) {
    for (ptrdiff_t k = 0; k < z; ++k) {
      sum += s_ptr[k + j*z + i*y*z];
    }
  }
}
```

Comparison to Raw Pointer



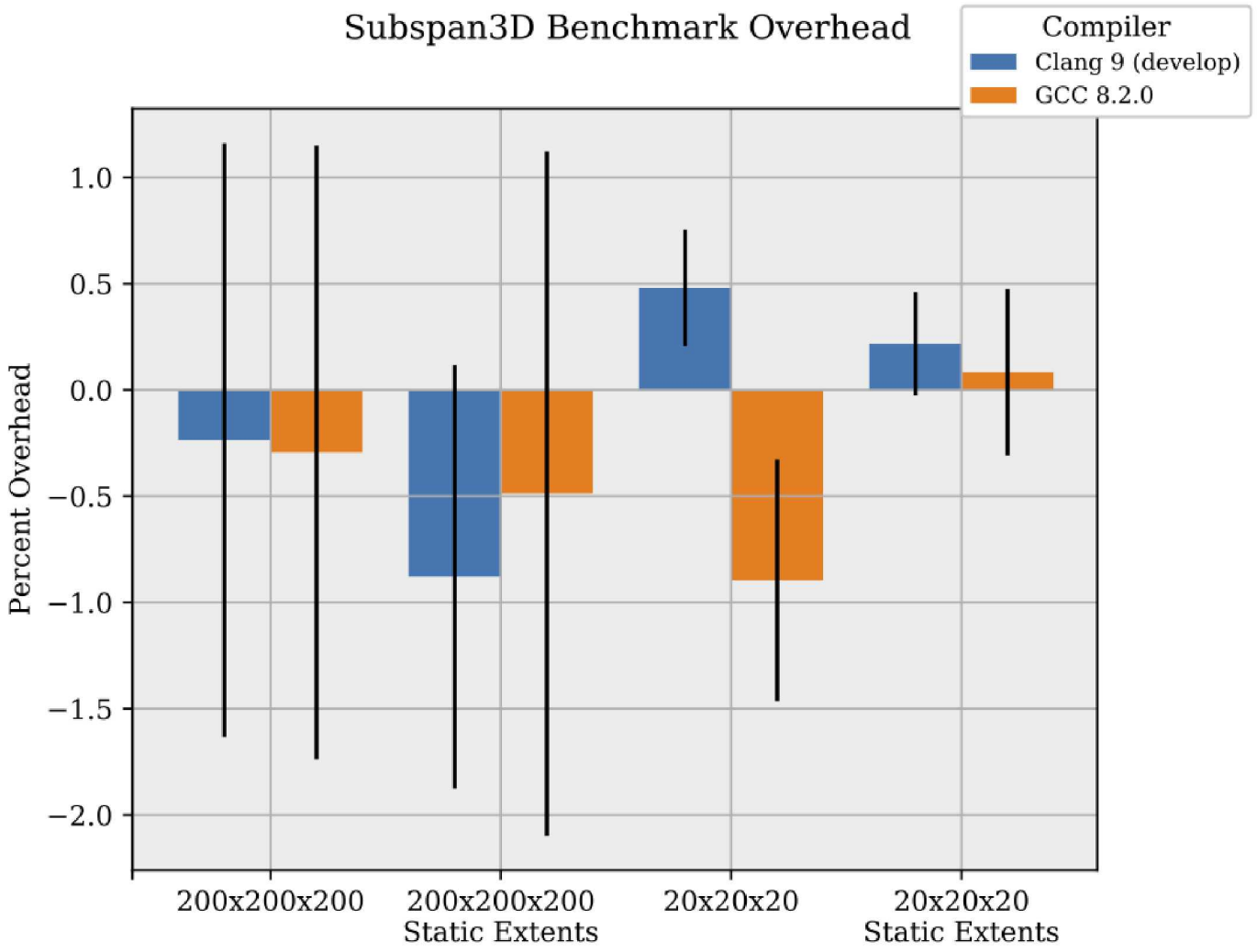
Effect of Layout on `MatVec`



Subspan3D: Abstraction Overhead Stress Test

```
for(ptrdiff_t i = 0; i < s.extent(0); ++i) {
  auto sub_i = subspan(s, i, all, all);
  for (ptrdiff_t j = 0; j < s.extent(1); ++j) {
    auto sub_i_j = subspan(sub_i, j, all);
    for (ptrdiff_t k = 0; k < s.extent(2); ++k) {
      sum += sub_i_j(k);
    }
  }
}
```

Subspan3D Overhead



Concluding Remarks

Concluding Remarks

Concluding Remarks

- `mdspan` provides *standard* C++ multidimensional array support (finally)

Concluding Remarks

- `mdspan` provides *standard* C++ multidimensional array support (finally)
- The `LayoutPolicy` and `Accessor` customization points allow adaptation to a diverse set of use cases

Concluding Remarks

- `mdspan` provides *standard* C++ multidimensional array support (finally)
- The `LayoutPolicy` and `Accessor` customization points allow adaptation to a diverse set of use cases
- Benchmarks demonstrate that the abstraction comes with zero overhead in most scenarios and with most compilers

Concluding Remarks

- `mdspan` provides *standard* C++ multidimensional array support (finally)
- The `LayoutPolicy` and `Accessor` customization points allow adaptation to a diverse set of use cases
- Benchmarks demonstrate that the abstraction comes with zero overhead in most scenarios and with most compilers
- Implementation is available at github.com/kokkos/mdspan.

Concluding Remarks

- `mdspan` provides *standard* C++ multidimensional array support (finally)
- The `LayoutPolicy` and `Accessor` customization points allow adaptation to a diverse set of use cases
- Benchmarks demonstrate that the abstraction comes with zero overhead in most scenarios and with most compilers
- Implementation is available at github.com/kokkos/mdspan.
  - Our implementation is backported all the way to C++11 (though it will use C++14 or C++17 to improve compilation times if available)

Concluding Remarks

- `mdspan` provides *standard* C++ multidimensional array support (finally)
- The `LayoutPolicy` and `Accessor` customization points allow adaptation to a diverse set of use cases
- Benchmarks demonstrate that the abstraction comes with zero overhead in most scenarios and with most compilers
- Implementation is available at github.com/kokkos/mdspan.
  - Our implementation is backported all the way to C++11 (though it will use C++14 or C++17 to improve compilation times if available)
  - We plan to submit this implementation as pull requests to the three major standard library implementations upon final acceptance of `mdspan` into C++23

Concluding Remarks

- `mdspan` provides *standard* C++ multidimensional array support (finally)
- The `LayoutPolicy` and `Accessor` customization points allow adaptation to a diverse set of use cases
- Benchmarks demonstrate that the abstraction comes with zero overhead in most scenarios and with most compilers
- Implementation is available at github.com/kokkos/mdspan.
  - Our implementation is backported all the way to C++11 (though it will use C++14 or C++17 to improve compilation times if available)
  - We plan to submit this implementation as pull requests to the three major standard library implementations upon final acceptance of `mdspan` into C++23
  - Feedback is appreciated!

Questions?