Practical Multidimensional Arrays and Linear Algebra in C++

SHARCNET Colloquium

Paul Preney, M.Sc, B.Ed., B.Sc.
Email: preney@sharcnet.ca

University of Windsor
Part of SHARCNET
Part of Compute Ontario
Part of Digital Research Alliance of Canada

Support Email: support@tech.alliancecan.ca

July 2, 2025

Land Acknowledgement

I am located at the University of Windsor.

The University of Windsor sits on the traditional territory of the Three Fires Confederacy of First Nations, which includes the Ojibwa, the Odawa, and the Potawatomi. We respect the longstanding relationships with First Nations people in this 100-mile Windsor-Essex peninsula region and the straits —les détroits— of Detroit.

Abstract

Support for multidimensional array accesses, array slicing, and linear algebra has been available for use in C++ compilers for a couple of years now. This very practical presentation will discuss how to make use of such (including in your existing sequential and parallel codes) on CPU and GPUs. Some aspects of performance will also be discussed for a variety of hardware including our current clusters and soon-to-be-running clusters.

Outline

Preliminaries: C++ Compilers

C++ compilers:

- Always prefer using the latest released stable compiler.
  - CPUs: Prefer using GCC or Clang.
  - CUDA GPUs: Prefer using the NVIDIA HPC SDK compiler.
- GCC: Prefer version 14 or newer. [7]
- Clang: Prefer version 17 or newer. [8]
- Intel C++: Prefer 2022.2 or newer. [1]
- NVIDIA HPC C++: Prefer 24.1 or newer. [3]
- MS Visual C++: Prefer 2022 version 17.14 or newer. [2]
- Apple Clang: Prefer version 16 or newer. [4]

Preliminaries: cppreference.com

In my opinion, https://cppreference.com/ is the best C++ online reference:

- it is a wiki
- it is *very* up-to-date
  - contains *all* ISO C++ standards
  - has many items from the next C++ standard
  - which standards are relevant are properly noted
- many entries have *working* code examples
- has C++ compilers' support with respect to each language and language library feature
- also has ISO C standard information

[9]

Preliminaries: Where To Find Existing Implementations

Existing implementations are available now!

- There is no need to wait until compilers implement them!

These implementations are:

- NVIDIA HPC SDK: Nov. 2022 release or newer [3]
  - If using an NVIDIA GPU or the NVIDIA compiler, prefer using this as it already has everything needed installed in it, e.g., both `mdspan` and `<linalg>`.
- `mdspan` and `submdspan` only:
  - `https://github.com/kokkos/mdspan` supporting C++14, C++17, C++20 and newer; CUDA and HIP; GCC and Clang; is header-only [6]
- `<linalg>` only:
  - `https://github.com/kokkos/stdBLAS` supporting C++17 or newer; GCC and Clang [5]

Preliminaries: Where To Find Existing Implementations (cont.)

The NVIDIA HPC SDK:

- allows one to compile ISO C++ and ISO Fortran code to run on CPUs and/or GPUs.
- Such features are enabled by using the `-stdpar` compiler option.
  - e.g., support for `mdspan`, `submdspan`, and `<linalg>`
- To use it on any of our clusters, load the `nvhpc` module (23.7 or newer), e.g.,
  - `module load nvhpc`
- iThe NVIDIA compiler requires one to use the function call operator instead of the multi-dimensional array operator, e.g.,
  - given an array called `ar`, use `ar(1,6,2)` instead of `ar[1,6,2]`

Preliminaries: Where To Find Existing Implementations (cont.)

Example NVIDIA HPC SDK build commands:

- `module load nvhpc`
- `nvc++ --c++23 -stdpar=multicore code.cpp` (CPU only)
- `nvc++ --c++23 -stdpar=gpu code.cpp` (CPU/GPU only)

Some CPU options: `-tp=haswell`, `-tp=skylake`, `-tp=zen2`, `-tp=zen3`

Some GPU minimum architecture options:

- Volta `-gpu=cu70`; Turing `-gpu=cu75`; Ampere `-gpu=cu80`; Hopper `-gpu=cu90`

Preliminaries: Where To Find Existing Implementations (cont.)

Building the Kokkos implementations:

- does *not* also need to install full Kokkos project package :-)
- is needed if other compilers than NVIDIA's need/want to be used
- likely to be header only
- compile `mdspan` to use the function call operator
  - i.e., set `MDSPAN_USE_PAREN_OPERATOR=1`
  - e.g., so your code will also work with the NVIDIA compiler
  - e.g., the `mdspan` appears to use the function call operator

Preliminaries: Where To Find Existing Implementations (cont.)

```
# A directory to work and install stuff in...
mkdir mydir
cd mydir

# Download and build Google Test...
git clone https://github.com/google/googletest.git
mkdir gtest.build
cd gtest.build
CC=gcc CXX=g++ cmake ../googletest \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=../installed \
  -DCMAKE_CXX_STANDARD=20 \
  -DCMAKE_CXX_EXTENSIONS=OFF
make
make install
cd ..
rm -rf gtest.build

# Download and build mdspan implementation...
```

Preliminaries: Where To Find Existing Implementations (cont.)

```
20 #   * Ensure C++20 is used and -DMDSPAN_USE_PAREN_OPERATOR=1 is set.
21 #   * Turn on desired items.
22 git clone https://github.com/kokkos/mdspan.git
23 mkdir mdspan.build
24 cd mdspan.build
25 CC=gcc CXX=g++ cmake ../mdspan \
26   -DCMAKE_BUILD_TYPE=Release \
27   -DCMAKE_INSTALL_PREFIX=../installed \
28   -DCMAKE_CXX_EXTENSIONS=OFF \
29   -DCMAKE_CXX_STANDARD=20 \
30   -DMDSPAN_CXX_STANDARD=20 \
31   -DMDSPAN_ENABLE_TESTS=OFF \
32   -DMDSPAN_ENABLE_BENCHMARKS=OFF \
33   -DMDSPAN_ENABLE_TESTS=OFF \
34   -DMDSPAN_ENABLE_EXAMPLES=OFF \
35   -DMDSPAN_ENABLE_BENCHMARKS=OFF \
36   -DMDSPAN_ENABLE_COMP_BENCH=OFF \
37   -DMDSPAN_ENABLE_CUDA=OFF \
38   -DMDSPAN_ENABLE_SYCL=OFF \
39   -DMDSPAN_ENABLE_HIP=OFF \
```

Preliminaries: Where To Find Existing Implementations (cont.)

```
40   -DMDSPAN_ENABLE_OPENMP=OFF \
41   -DMDSPAN_USE_SYSTEM_GTEST=OFF \
42   -DMDSPAN_USE_SYSTEM_BENCHMARK=OFF \
43   -DMDSPAN_INSTALL_STDMODE_HEADERS=ON \
44   -DMDSPAN_USE_BRACKET_OPERATOR=0 \
45   -DMDSPAN_USE_PAREN_OPERATOR=1
46 make
47 make install
48 cd ..
49 rm -rf mdspan.build
50
51 # Download and build stdBLAS implementation...
52 git clone https://github.com/kokkos/stdBLAS.git
53 mkdir stdblas.build
54 cd stdblas.build
55 CC=gcc CXX=g++ cmake ../stdBLAS \
56   -DCMAKE_BUILD_TYPE=Release \
57   -DCMAKE_INSTALL_PREFIX=../installed \
58   -DCMAKE_CXX_STANDARD=20 \
59   -DCMAKE_CXX_EXTENSIONS=OFF \
```

Preliminaries: Where To Find Existing Implementations (cont.)

```
60   -DLINALG_ENABLE_KOKKOS=OFF \
61   -DLINALG_ENABLE_TESTS=OFF \
62   -DLINALG_ENABLE_BLAS=OFF \
63   -DLINALG_ENABLE_EXAMPLES=OFF
64 # Notice the above command has ENABLE_BLAS turned off.
65 # If you've an installed BLAS it can be turned on.
66 # Our clusters load flexiblas by default so it can be turned on.
67 make
68 make install
69 cd ..
70 rm -rf stdblas.build
```

Preliminaries: Where To Find Existing Implementations (cont.)

To use these will need to add to your compiler's command line:

- `-I./mydir/installed/include -I./mydir/installed/include/experimental`
- `./mydir/installed` is where you installed the above libraries

If you specified the use of an existing BLAS implementation, remember to tell the compiler to link to the appropriate blas library, e.g.,

- `-lflexiblas`
- `-lblas`
- `-lblis`

etc. as is appropriate.

If using GCC and parallel algorithms, remember to:

- `module load tbb` on our clusters
- link the TBB library to your program with `-ltbb`

Preliminaries: Hiding Implementations in a Header File

Since `mdspan` is already in the C++ standard + `submdspan` and `<linalg>` will soon be in the standard, and, if you are using an implementation:

- it is worth considering hiding implementation-specific stuff in a header file

Preliminaries: Hiding Implementations in a Header File (cont.)

For example the header file could be:

myheader.hpp

```
#ifndef myheader_hpp_
#define myheader_hpp_

#include <version> // requires C++20 or newer

// include <mdspan> header...
//   * Define MDSPAN_USES_STD_MDSPAN macro only when std::mdspan is used.
//     Why? If this macro is defined one can only use array brackets to
//     access the multi-dimensional index. (The implementations permit
//      using the function call operator.)
#ifdef __has_include
#  if __has_include(<mdspan>) && \
      defined(__cpp_lib_mdspan) && __cpp_lib_mdspan ≥ 202207L && \
      defined(__cpp_lib_submdspan) && __cpp_lib_submdspan ≥ 202306L
#    include <mdspan>
#    define MDSPAN_NAMESPACE  ::std
#    define MDSPAN_USES_STD_MDSPAN
#  elif __has_include(<experimental/mdspan>)
```

Preliminaries: Hiding Implementations in a Header File (cont.)

```
19 #    include <experimental/mdspan>
20 #    ifdef MDSPAN_IMPL_STANDARD_NAMESPACE
21        // Kokkos' mdspan is in namespace MDSPAN_IMPL_STANDARD_NAMESPACE...
22 #      define MDSPAN_NAMESPACE  :: MDSPAN_IMPL_STANDARD_NAMESPACE
23 #    else
24 #      define MDSPAN_NAMESPACE  ::std::experimental
25 #    endif
26 #  else
27 #    error "Fix code to properly #include <mdspan>."
28 #  endif
29 #endif
30
31 #ifdef __has_include
32 #  if __has_include(<linalg>) && __cplusplus > 202311L /* Need C++26 */
33 #    include <linalg>
34 #    define LINALG_NAMESPACE  ::std::linalg
35 #  elif __has_include(<experimental/linalg>)
36 #    include <experimental/linalg>
37 #    if defined(MDSPAN_IMPL_STANDARD_NAMESPACE) && defined(MDSPAN_IMPL_PROPOSED_NAMESPACE)
38        // Kokkos' linalg is in namespace MDSPAN_IMPL_STANDARD_NAMESPACE :: MDSPAN_IMPL_PROPOSED_NAMESPACE
```

Preliminaries: Hiding Implementations in a Header File (cont.)

```
#      define LINALG_NAMESPACE  :: MDSPAN_IMPL_STANDARD_NAMESPACE :: MDSPAN_IMPL_PROPOSED_NAMESPACE::linalg
#    else
#      define LINALG_NAMESPACE  ::std::experimental::linalg
#    endif
#  else
#    error "Fix code to properly #include <linalg>."
#  endif
#endif

#endif // #ifndef myheader_hpp_
```

which can ease coding, e.g.,

```
// avoid having to specify the namespaces in code...
using namespace std;
using namespace MDSPAN_NAMESPACE;
using LINALG_NAMESPACE::matrix_vector_product;
// etc.
```

Multidimensional Arrays in C++: Overview

Q. Why does one want to use `mdspan`?

A. Manually calculating multidimensional indices into 1D arrays:

- can be error-prone
- can sometimes be annoying/hard
- can inefficient
- can result in code that is hard to modify later if there are changes

Multidimensional Arrays in C++: Overview (cont.)

Solution: Use `mdspan`!

`mdspan` supports:

- mapping dense 1D arrays to multidimensions
- each dimension's extent (size) can be specified at compile-time, or, run-time
  - NOTE: Although a little more typing when declaring an `mdspan`, compile-time specified extents can generate code that is a lot more efficient.
- multiple memory layouts are supported
  - e.g., row-major (default), column-major, etc.
- multuple access methods are supported
  - e.g., direct (default), using atomics/locks (custom code), etc.

Multidimensional Arrays in C++: Overview (cont.)

An example code fragment demonstrating its use:

```
1  vector<double> in(N*M);       // allocate memory for an N*M input array
2  vector<double> out(N*M);      // allocate memory for an N*M output array
3
4  mdspan A{in.data(), N, M};    // A is an N*M multidimensional view of in
5  mdspan B{out.data(), M, N};   // B is an M*N multidimensional view of out
6
7  // compute the Cartesian product of all (N,M) values, e.g.,
8  constexpr std::size_t ZERO{};
9  auto a = std::views::iota(ZERO,A.extent(0));  // C++20
10 auto b = std::views::iota(ZERO,A.extent(1));  // C++20
11 auto v = std::views::cartesian_product(a,b);  // C++23
12
13 // put the C++20 range v into a vector for parallel use...
14 //   * C++20 ranges are sequential only
15 //   * If wanting to run in parallel, at this time, put any C++20
16 //     range results into a vector.
17 vector v2(begin(v), end(v));
18
```

Multidimensional Arrays in C++: Overview (cont.)

```
19  // run std::for_each in parallel to compute the transpose...
20  for_each(
21    execution::par_unseq,
22    begin(v2), end(v2),
23    [=] (auto idx)              // notice capture by value with '='
24    {
25      auto [i, j] = idx;        // idx is a tuple pair (i,j)
26      B[j,i] = A[i,j];          // switch [] to () if implementation requires it
27    }
28  );
```

Multidimensional Arrays in C++: The mdspan Class

`mdspan` is a class template with the following parameters:

```
1 template <
2   typename T,
3   typename Extents,
4   typename LayoutPolicy = std::layout_right,
5   typename AccessorPolicy = std::default_accessor<T>
6 >
7 class mdspan;
```

- **`T`** is the type being stored in the multidimensional array
- **`Extents`** is used to specify the dimensions' sizes
- **`LayoutPolicy`** is used to specify the memory layout of the multidimensional array, i.e., it maps the multidimensional index space to an underlying 1D index
- **`AccessorPolicy`** is used to specify each element's memory access operation, i.e., it maps the underlying 1D index to a reference of type `T`

Multidimensional Arrays in C++: Element Type

The first template parameter of `std::mdspan` specifies the data type of the multidimensional array's elements.

NOTE:

- `std::mdspan`s are **views**
  - i.e., they are **small** in size
  - i.e., they are **fast to copy**
  - i.e., they have **reference semantics**
  - views are "non-owning": they do *not* allocate and deallocate the data that can be accessed and have *reference semantics*
- `std::mdspan`s *not* containers
  - containers are "owning": they allocate and deallocate the data they contain and (almost always) have *value semantics*

Multidimensional Arrays in C++: Extents

The second template parameter is typically an instance of `std::extents`. This parameter:

- represents a multidimensional **index space** of **rank** equal to the **number of extents in this index space**, i.e., it specifies:
  - the **number of dimensions**, i.e., the `mdspan`'s rank
  - each **dimension's size**, i.e., its extent
    - if known at compile-time, then the dimension size must be an integer value
    - if known only at run-time, then the dimension size must be `std::dynamic_extent`
    - if all dimensions are known only at run-time, then `std::dextents` can be used to specify all dimensions
    - more efficient code is generated if static extents are used
- is regular and is trivially copyable

Multidimensional Arrays in C++: Extents (cont.)

Examples of the second template parameter:

- 5x3x15: `extents<size_t, 5, 3, 15>`
- 5x?x15: `extents<size_t, 5, dynamic_extent, 15>`
  - Remember to passing the unknown dimension when declarating the `mdspan` variable: `mdspan<float,extents<size_t,5,dynamic_extent,15>> m{ aptr, 3 };`
- 5x?x?: `extents<size_t, 5, dynamic_extent, dynamic_extent>`
  - e.g., `mdspan<float,extents<size_t,5,dynamic_extent,dynamic_extent>> m{ aptr, 3, 15 };`
- ?x?x?: `extents<size_t, dynamic_extent, dynamic_extent, dynamic_extent>`
  - e.g., `mdspan<float,extents<size_t,dynamic_extent,dynamic_extent,dynamic_extent>> m{ aptr, 5, 3, 15 };`
- ?x?x?: use `dextents<size_t, 3>` as the second parameter

Multidimensional Arrays in C++: Layout Policy

Every `mdspan` has a layout management policy type.

C++23 has three layout management policy types:

- **`std::layout_right`**: row-major multidimensional array layout, i.e., the rightmost extent has stride 1, e.g., C array layout
- **`std::layout_left`**: column-major multidimensional array layout, i.e., the leftmost extent has stride 1, e.g., Fortran array layout
- **`std::layout_stride`**: a layout mapping policy with user-defined strides

If not specified, `std::layout_right` is the default policy.

Multidimensional Arrays in C++: Accessor Policy

Every `mdspan` has an accessor policy type.

C++23 has one accessor policy defined:

- **`std::default_accessor`**: directly access the requested element

Why have accessors?

- They enable changing how memory is accessed.
  - e.g., a concurrent program may need to obtain a lock before modifying an element

std::submdspan

`std::submdspan` is a **function** that allows one to obtain a `std::mdspan` that is a **subset** of an existing `std::mdspan`

- e.g., a row or column of a matrix

std::submdspan (cont.)

```
1  auto row(auto matrix, size_t i)                    // matrix must be an mdspan
2    requires (decltype(matrix)::rank() == 2)
3  {
4    return submdspan(matrix, i, std::full_extent);
5  }
6
7  auto col(auto matrix, size_t i)                    // matrix must be an mdspan
8    requires (decltype(matrix)::rank() == 2)
9  {
10   return submdspan(matrix, std::full_extent, i);
11 }
```

std::submdspan (cont.)

Consider Strassen's matrix multiplication algorithm which requires square power-of-two rank input matrices A and B.

Given same-size input matrices A and B and a same-sized output matrix C, Strassen's algorithm subdivides A, B, and C as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}; B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}; C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

where each submatrix has the same size.

std::submdspan (cont.)

Strassen's algorithm evaluates such using:

$$
\begin{aligned}
Q_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q_2 &= (A_{21} + A_{22})B_{11} \\
Q_3 &= A_{11}(B_{12} - B_{22}) \\
Q_4 &= A_{22}(-B_{11} + B_{21}) \\
Q_5 &= (A_{11} + A_{12})B_{22} \\
Q_6 &= (-A_{11} + A_{21})(B_{11} + B_{12}) \\
Q_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

(where all matrix multiplications should be recursively done.)

std::submdspan (cont.)

To create A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, C_{11}, C_{12}, C_{21}, and C_{22} first determine the lower and upper index ranges from the input matrix A (which is assumed here to be an `mdspan`)

```
1 using index_type = decltype(A)::index_type;
2 using subset_type = tuple<index_type,index_type>;
3
4 subset_type const lower{ index_type{}, a.extent(0)/2 };
5 subset_type const upper{ a.extent(0)/2, a.extent(0) };
```

std::submdspan (cont.)

Then use these lower and upper ranges to determine all needed submatrices A_{11}, A_{12}, A_{21}, A_{22}, B_{11}, B_{12}, B_{21}, B_{22}, C_{11}, C_{12}, C_{21}, and C_{22}:

```
1  auto A11{ submdspan(a, lower, lower) };
2  auto A12{ submdspan(a, lower, upper) };
3  auto A21{ submdspan(a, upper, lower) };
4  auto A22{ submdspan(a, upper, upper) };
5  auto B11{ submdspan(b, lower, lower) };
6  auto B12{ submdspan(b, lower, upper) };
7  auto B21{ submdspan(b, upper, lower) };
8  auto B22{ submdspan(b, upper, upper) };
9  auto C11{ submdspan(c, lower, lower) };
10 auto C12{ submdspan(c, lower, upper) };
11 auto C21{ submdspan(c, upper, lower) };
12 auto C22{ submdspan(c, upper, upper) };
```

and use these to perform the rest of the calculations.

std::submdspan (cont.)

Cool! So what does `submdspan()` support?

std::submdspan (cont.)

A subset of a dimension can be specified as follows:

- using a **single integral value**
  - the value refers to the use of a **specific index** for that dimension
  - using such reduces the subset `mdspan` rank by one
- using a type convertible to `tuple<mdspan::index_type,mdspan::index_type>`
  - these values determine a **half-open range**, i.e., $[start, stop)$, of index values to use for that dimension
- using `std::strided_slice{a,b,c}`
  - `a` is the **start**ing index for that dimension
  - `b` is the **extent** (size) for that dimension
  - `c` is the **stride** (increment or step) starting at the starting index for that dimension
- using `std::full_extent`
  - this value uses **all index values** for that dimension

std::submdspan (cont.)

Other programming languages that have multidimensional arrays with slicing permit a stride/step parameter in addition to beginning and ending values:

- Fortran: `a(start:stop:step)`
- Numpy (Python): `a[start:stop:step]`
- MATLAB: `a(start:step:stop)`

but, for efficiency reasons, the design of `std::submdspan` is to use:

- the **start**ing index,
- the **extent** (size), and,
- the **stride** (step size).

i.e., in C++ this is effectively "`a[start:stop-start:step]`"

std::submdspan Examples

Given:

```
1 vector v{ 10, 0.0 };              // vector of 10 doubles
2 iota(v.begin(), v.end(), 0.0);    // assign 0.0 to 9.0 to elements
3 mdspan m{ v.data(), 10 };         // 1D mdspan
4 assert(m.rank() == 1);
```

e.g.,

- `auto sm = submdspan(m,4);`
  - 0D, i.e., scalar, mdspan result, i.e., the 5th element
  - `sm.rank() == 0`
  - `sm[] == m[4]`
  - `sm[] == 4.0`

std::submdspan Examples (cont.)

Given:

```
1 vector v{ 10, 0.0 };              // vector of 10 doubles
2 iota(v.begin(), v.end(), 0.0);    // assign 0.0 to 9.0 to elements
3 mdspan m{ v.data(), 10 };         // 1D mdspan
4 assert(m.rank() == 1);
```

e.g.,

- `auto sm = submdspan(m, tuple{2,5});`
  - `sm.rank() == 1`
  - `sm.extent(0) == 3`
  - `sm[0] == 2.0`
  - `sm[1] == 3.0`
  - `sm[2] == 4.0`

std::submdspan Examples (cont.)

Given:

```
1 vector v{ 10, 0.0 };              // vector of 10 doubles
2 iota(v.begin(), v.end(), 0.0);    // assign 0.0 to 9.0 to elements
3 mdspan m{ v.data(), 10 };         // 1D mdspan
4 assert(m.rank() == 1);
```

e.g.,

- `auto sm = submdspan(m, strided_slice{3,7,2});`
  - `sm.rank() == 1`
  - `sm.extent(0) == 4`
  - `sm[0] == 3.0`
  - `sm[1] == 5.0`
  - `sm[2] == 7.0`
  - `sm[3] == 9.0`

std::submdspan Examples (cont.)

Given:

```
1 vector v{ 10, 0.0 };              // vector of 10 doubles
2 iota(v.begin(), v.end(), 0.0);    // assign 0.0 to 9.0 to elements
3 mdspan m{ v.data(), 10 };         // 1D mdspan
4 assert(m.rank() == 1);
```

e.g.,

- `auto sm = submdspan(m, full_extent);`
  - `sm.rank() == 1`
  - `sm.extent(0) == m.extent(0)`
  - `sm[0] == 0.0`
  - `sm[1] == 1.0`
  - etc.

std::submdspan Examples (cont.)

Given:

```
1 vector v{ 10*10, 0.0 };           // vector of 100 doubles
2 iota(v.begin(), v.end(), 0.0);    // assign 0.0 to 99.0 to elements
3 mdspan m{ v.data(), 10, 10 };     // 2D mdspan
4 assert(m.rank() == 2);
```

e.g.,

- `auto sm = submdspan(m, full_extent, 3);`
  - `sm` is made up of `m`'s first dimension corresponding to the fourth `m`'s second dimension
  - `sm.rank() == 1`
  - `sm.extent(0) == m.extent(0)`
  - `sm[0] == m[i,3]` for some `i`

std::submdspan Examples (cont.)

Given:

```
1 vector v{ 10*10*10, 0.0 };        // vector of 1000 doubles
2 iota(v.begin(), v.end(), 0.0);    // assign 0.0 to 999.0 to elements
3 mdspan m{ v.data(), 10, 10, 10 }; // 3D mdspan
4 assert(m.rank() == 3);
```

e.g.,

- `auto sm = submdspan(m, strided_slice{3,7,2}, 0, full_extent);`
  - `sm.rank() == 2`
  - `sm.extent(0) == 4`
  - `sm.extent(1) == m.extent(2)`
  - `sm[i,j] == m[3+i*2,0,j]` for some `i` and `j`

Linear Algebra in C++

C++26 is likely to have the new `<linalg>` header implementing the BLAS 1, 2, and 3 functions.

The implementations mentioned already implement `<linalg>` using `mdspan` and `submdspan`.

The `<linalg>` functions are like the C++ standard library's algorithms:

- call then to use them
- instead of passing containers and ranges appropriately pass 1D or 2D mdspans
- if passed as the first argument `std::execution::unseq`, `std::execution::par`, or `std::execution::par_unseq` the code will run in parallel

Linear Algebra in C++ (cont.)

The BLAS 1, 2, 3, and, LAPACK routines implemented are:

- BLAS 1: `xLARTG`, `xROT`, `xSWAP`, `xSCAL`, `xCOPY`, `xAXPY`, `xDOT`, `xDOTU`, `xDOTC`, `xNRM2`, `SASUM`, `DASUM`, `SCASUM`, `DZASUM`, `IxAMAX`
- BLAS 2: `xGER`, `xGERU`, `xGEMV`, `xTRMV`, `xTPMV`, `xTRSV`, `xTPSV`, `xSYR`, `xSYR2`, `xSPR`, `xSPR2`, `xHER`, `xHER2`, `xHPR`, `xHPR2`
- BLAS 3: `xGEMM`, `xSYMM`, `xHEMM`, `xTRMM`, `xSYRK`, `xSYRK2`, `xHERK`, `xHERK2`, `xTSRM`
- LAPACK: `xLASSQ`
- Other routines in `<linalg>`: Frobenius norm, one norm, infinity norm

These routines are documented in this slide decks' appendices.

Some Examples

To print out any 2D `mdspan` “matrix”:

```
1  template <typename T>
2  requires (T::rank() == 2)     // require rank() == 2, i.e., 2D
3  void print_mdspan(T m)        // m is an mdspan
4  {
5    using std::size_t;
6    using std::cout;
7
8    // Like all C++ arrays, dimensions are indexed from 0.
9    // In m.extent(i), i is that dimension's index.
10   for (size_t i=0; i != m.extent(0); ++i)
11   {
12     for (size_t j=0; j != m.extent(1); ++j)
13       cout << m(i,j) << ' ';
14     cout << '\n';
15   }
16   cout << '\n';
17 }
```

Some Examples (cont.)

Calling such is straight-forward:

```
1  vector<double> a(4*3); iota(a.begin(), a.end(), 0.0); mdspan am{ a.data(), 4, 3 };
2  vector<double> b(3*7); iota(b.begin(), b.end(), 0.0); mdspan bm{ b.data(), 3, 7 };
3
4  cout << "Matrix A:\n"; print_mdspan(am);
5  cout << "Matrix B:\n"; print_mdspan(bm);
6
7  vector c(4*7, 0.0); mdspan cm{ c.data(), 4, 7 };
8  matrix_product(execution::par, am, bm, cm);
9
10 cout << "Matrix C:\n"; print_mdspan(cm);
```

Questions, etc.

Questions, etc.

Bibliography

[1] Intel Corporation. *Get Intel oneAPI DPC++/C++ Compiler*. 2025. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler-download.html.

[2] Microsoft Corporation. *Visual Studio: IDE and Code Editor for Software Developers and Teams*. 2025. URL: https://visualstudio.microsoft.com/.

[3] NVIDIA Corporation. *NVIDIA HPC SDK*. 2025. URL: https://developer.nvidia.com/hpc-sdk.

[4] Apple Inc. *Xcode*. 2025. URL: https://developer.apple.com/xcode/.

[5] LLC (NTESS) National Technology & Engineering Solutions of Sandia. *P1673 reference implementation*. 2023. URL: https://github.com/kokkos/stdBLAS.

[6] LLC (NTESS) National Technology & Engineering Solutions of Sandia. *Reference mdspan implementation*. 2022. URL: https://github.com/kokkos/mdspan.

[7] GNU Project. *GCC, the GNU Compiler Collection*. 2025. URL: https://gcc.gnu.org.

Bibliography (cont.)

[8] LLVM Project. *Clang: a C language family frontend for LLVM*. 2025. URL: https://clang.llvm.org/.

[9] C++ enthusiasts from around the world. *C++ reference*. 2025. URL: https://cppreference.com.

Appendix: BLAS 1: Given's Rotation

A Given's rotation is a rotation in a plane:

- `setup_givens_rotation_result<T> setup_givens_rotation(Real a, Real b)`
- `void apply_givens_rotation(InOutVec1 x, InOutVec2 y, Real c, Real s)`
- `void apply_givens_rotation(InOutVec1 x, InOutVec2 y, Real c, complex<Real> s)`
- `void apply_givens_rotation(ExecPolicy&&, InOutVec1 x, InOutVec2 y, Real c, Real s)`
- `void apply_givens_rotation(ExecPolicy&&, InOutVec1 x, InOutVec2 y, Real c, complex<Real> s)`

Result members:

- `std::linalg::setup_givens_rotation_result<T>` members: `T c, s, r;`
- `std::linalg::setup_givens_rotation_result<std::complex<T>>` members: `T c; std::complex<T> s, r;`

These functions correspond to the BLAS functions as follows:

- `setup_givens_rotation`: `xLARTG`
- `apply_givens_rotation`: `xROT`

Appendix: BLAS 1: Swap

Swap all corresponding elements of an `std::mdspan`:

- `void swap_elements(InOutObj1 x, InOutObj2 y)`
- `void swap_elements(ExecPolicy&&, InOutObj1 x, InOutObj2 y)`

These functions correspond to the BLAS functions `xSWAP`.

Appendix: BLAS 1: Scale

Multiply all elements by a scalar:

- `void scale(Scalar alpha, InOutObj x)`
- `void scale(ExecPolicy&&, Scalar alpha, InOutObj x)`
- `auto scaled(ScalingFactor alpha, mdspan<ElementType, Extents, Layout, Accessor> x)`
  - Returns a **read-only** `std::mdspan` that lazily scales each element of `x`.

These functions correspond to the BLAS functions `xSCAL`.

Appendix: BLAS 1: Copy

Copy all elements:

- `void copy(InObj x, OutObj y)`
- `void copy(ExecPolicy&&, InObj x, OutObj y)`

These functions correspond to the BLAS functions `xCOPY`.

Appendix: BLAS 1: Add

Add corresponding elements:

- `void add(InObj1 x, InObj2 y, OutObj z)`
- `void add(ExecPolicy&&, InObj1 x, InObj2 y, OutObj z)`

These functions correspond to the BLAS functions `xAXPY`.

Appendix: BLAS 1: Dot Product

Non-conjugated dot product of two vectors:

- `Scalar dot(InVec1 v1, InVec2 v2)`
- `Scalar dot(InVec1 v1, InVec2 v2, Scalar init)`
- `Scalar dot(ExecPolicy&&, InVec1 v1, InVec2 v2)`
- `Scalar dot(ExecPolicy&&, InVec1 v1, InVec2 v2, Scalar init)`

Conjugated dot product of two vectors:

- `Scalar dotc(InVec1 v1, InVec2 v2)`
- `Scalar dotc(InVec1 v1, InVec2 v2, Scalar init)`
- `Scalar dotc(ExecPolicy&&, InVec1 v1, InVec2 v2)`
- `Scalar dotc(ExecPolicy&&, InVec1 v1, InVec2 v2, Scalar init)`

These functions correspond to the BLAS functions: `xDOT`, `xDOTU`, and `xDOTC`.

Appendix: BLAS 1: Sum of Squares

Sum of squares of a vector's elements:

- `sum_of_squares_result<Scalar> vector_sum_of_squares(InVec v, sum_of_squares_result<Scalar> init)`
- `sum_of_squares_result<Scalar> vector_sum_of_squares(ExecPolicy&&, InVec v, sum_of_squares_result<Scalar> init)`

`std::libalg::sum_of_squares_result<T>` members:

- `T scaling_factor;`
- `T scaled_sum_of_squares;`

These functions correspond to the LAPACK functions `xLASSQ`.

Appendix: BLAS 1: Euclidean Norm

Euclidean norm of a vector:

- `auto vector_two_norm(InVec v)`
- `auto vector_two_norm(ExecPolicy&&, InVec v)`
- `Scalar vector_two_norm(InVec v, Scalar init)`
- `Scalar vector_two_norm(ExecPolicy&&, InVec v, Scalar init)`

These functions correspond to the BLAS functions `xNRM2`.

Appendix: BLAS 1: Sub of Absolute Values

Sum of absolute values of vector elements:

- `auto vector_abs_sum(InVec v)`
- `auto vector_abs_sum(ExecPolicy&&, InVec v)`
- `Scalar vector_abs_sum(InVec v, Scalar init)`
- `Scalar vector_abs_sum(ExecPolicy&&, InVec v, Scalar init)`

These functions correspond to the BLAS functions `SASUM`, `DASUM`, `SCASUM`, and `DZASUM`.

Appendix: BLAS 1: Index of Maximum Absolute Value

Index of maximum absolute value of vector elements:

- `typename InVec::size_type vector_idx_abs_max(InVec v)`
- `typename InVec::size_type vector_idx_abs_max(ExecPolicy&&, InVec v)`

These functions correspond to the BLAS functions `IxAMAX`.

Appendix: BLAS 1: Frobenius Norm

Frobenius norm of a matrix:

- `auto matrix_frob_norm(InMat A)`
- `auto matrix_frob_norm(ExecPolicy&&, InMat A)`
- `Scalar matrix_frob_norm(InMat A, Scalar init)`
- `Scalar matrix_frob_norm(ExecPolicy&&, InMat A, Scalar init)`

These functions exist in the BLAS standard but are not part of the reference implementation.

Appendix: BLAS 1: One Norm

One norm of a matrix:

- `auto matrix_one_norm(InMat A)`
- `auto matrix_one_norm(ExecPolicy&&, InMat A)`
- `Scalar matrix_one_norm(InMat A, Scalar init)`
- `Scalar matrix_one_norm(ExecPolicy&&, InMat A, Scalar init)`

These functions exist in the BLAS standard but are not part of the reference implementation.

Appendix: BLAS 1: Infinity Norm

Infinity norm of a matrix:

- `auto matrix_inf_norm(InMat A)`
- `auto matrix_inf_norm(ExecPolicy&&, InMat A)`
- `Scalar matrix_inf_norm(InMat A, Scalar init)`
- `Scalar matrix_inf_norm(ExecPolicy&&, InMat A, Scalar init)`

These functions exist in the BLAS standard but are not part of the reference implementation.

Appendix: BLAS 2: General Matrix-Vector Product

General matrix-vector product:

- `void matrix_vector_product(InMat A, InVec x, OutVec y)`
- `void matrix_vector_product(InMat A, InVec1 x, InVec2 y, OutVec z)`
- `void matrix_vector_product(ExecPolicy&&, InMat A, InVec x, OutVec y)`
- `void matrix_vector_product(ExecPolicy&&, InMat A, InVec1 x, InVec2 y, OutVec z)`

These functions correspond to the BLAS functions `xGEMV`.

- Complexity: $O($`x.extent(0)*A.extent(1)`$)$
- Computes: $y = Ax$ and $z = y + Ax$, z may alias y

Appendix: BLAS 2: Symmetric Matrix-Vector Product

Symmetric matrix-vector product:

- `void symmetric_matrix_vector_product(InMat A, Triangle t, InVec x, OutVec y)`
- `void symmetric_matrix_vector_product(ExecPolicy&&, InMat A, Triangle t, InVec x, OutVec y)`
- `void symmetric_matrix_vector_product(InMat A, Triangle t, InVec1 x, InVec2 y, OutVec z)`
- `void symmetric_matrix_vector_product(ExecPolicy&&, InMat A, Triangle t, InVec1 x, InVec2 y, OutVec z)`

These functions correspond to the BLAS functions `xGEMV`.

- Complexity: $O($`x.extent(0)*A.extent(1)`$)$
- Computes: $y = Ax$ and $z = y + Ax$, z may alias y

Appendix: BLAS 2: Hermitian Matrix-Vector Product

Hermitian matrix-vector product:

- `void hermitian_matrix_vector_product(InMat A, Triangle t, InVec x, OutVec y)`
- `void hermitian_matrix_vector_product(ExecPolicy&&, InMat A, Triangle t, InVec x, OutVec y)`
- `void hermitian_matrix_vector_product(InMat A, Triangle t, InVec1 x, InVec2 y, OutVec z)`
- `void hermitian_matrix_vector_product(ExecPolicy&&, InMat A, Triangle t, InVec1 x, InVec2 y, OutVec z)`

These functions correspond to the BLAS functions `xGEMV`.

- Complexity: $O($`x.extent(0)*A.extent(1)`$)$
- Computes: $y = Ax$ and $z = y + Ax$, z may alias y

Appendix: BLAS 2: Triangular Matrix-Vector Product

Triangular matrix-vector product:

A) `void triangular_matrix_vector_product(InMat A, Triangle t, DiagonalStorage d, InVec x, OutVec y)`
`void triangular_matrix_vector_product(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InVec x, OutVec y)`

B) `void triangular_matrix_vector_product(InMat A, Triangle t, DiagonalStorage d, InOutVec y)`
`void triangular_matrix_vector_product(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutVec y)`

C) `void triangular_matrix_vector_product(InMat A, Triangle t, DiagonalStorage d, InVec1 x, InVec2 y, OutVec z)`
`void triangular_matrix_vector_product(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InVec1 x, InVec2 y, OutVec z)`

These functions correspond to the BLAS functions `xTRMV` and `xTPMV`.

A) Complexity: $O($`x.extent(0)*A.extent(1)`$)$; Computes: $y = Ax$
B) Complexity: $O($`y.extent(0)*A.extent(1)`$)$; Computes: $y = Ax$ (in-place)
   - Using this operation hinders parallelization although vectorization is still possible.
C) Complexity: $O($`x.extent(0)*A.extent(1)`$)$; Computes: $z = y + Ax$, z may alias y

Appendix: BLAS 2: Solve a Triangular Linear System

Solve a triangular linear system:

A)
```
void triangular_matrix_vector_solve(InMat A, Triangle t, DiagonalStorage d, InVec b, OutVec x)
void triangular_matrix_vector_solve(InMat A, Triangle t, DiagonalStorage d, InVec b, OutVec x, BinaryDivideOp divide)
void triangular_matrix_vector_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InVec b, OutVec x)
void triangular_matrix_vector_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InVec b, OutVec x, BinaryDivideOp divide)
```

A)
```
void triangular_matrix_vector_solve(InMat A, Triangle t, DiagonalStorage d, InOutVec b)
void triangular_matrix_vector_solve(InMat A, Triangle t, DiagonalStorage d, InOutVec b, BinaryDivideOp divide)
void triangular_matrix_vector_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutVec b)
void triangular_matrix_vector_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutVec b, BinaryDivideOp divide)
```

These functions correspond to the BLAS functions `xTRSV` and `xTPSV`.

A) Complexity: $O($`A.extent(1)*b.extent(0)`$)$; Computes: a vector x such that $b = Ax$
B) Complexity: $O($`y.extent(0)*A.extent(1)`$)$; Computes: a vector b such that $b = Ab$ (in-place)

- Using this operation hinders parallelization although vectorization is still possible.

Appendix: BLAS 2: Rank-1 Matrix Updates

Compute the non-symmetric non-conjugated rank-1 (outer product) update of a matrix:

- `void matrix_rank_1_update(InVec1 x, InVec2 y, InOutMat A)`
- `void matrix_rank_1_update(ExecPolicy&&, InVec1 x, InVec2 y, InOutMat A)`

These functions correspond to the BLAS functions `xGER`.

- Complexity: $O($`x.extent(0)*y.extent(0)`$)$
- Computes: a matrix A such that $A = A + xy^T$

Appendix: BLAS 2: Rank-1 Matrix Updates (cont.)

Compute the non-symmetric conjugated rank-1 (outer product) update of a matrix:

- `void matrix_rank_1_update_c(InVec1 x, InVec2 y, InOutMat A)`
- `void matrix_rank_1_update_c(ExecPolicy&&, InVec1 x, InVec2 y, InOutMat A)`

These functions correspond to the BLAS functions `xGERU`.

- Complexity: $O($`x.extent(0)*y.extent(0)`$)$
- Computes: a matrix A such that $A = A + xy^T$

Appendix: BLAS 2: Rank-1 Matrix Updates (cont.)

Compute the symmetric rank-1 (outer product) update of a matrix:

- `void symmetric_matrix_rank_1_update(InVec x, InOutMat A, Triangle t)`
- `void symmetric_matrix_rank_1_update(Scalar alpha, InVec x, InOutMat A, Triangle t)`
- `void symmetric_matrix_rank_1_update(ExecPolicy&&, InVec x, InOutMat A, Triangle t)`
- `void symmetric_matrix_rank_1_update(ExecPolicy&&, Scalar alpha, InVec x, InOutMat A, Triangle t)`

These functions correspond to the BLAS functions `xSYR` and `xSPR`.

- Complexity: $O($`x.extent(0)*x.extent(0)`$)$
- Computes: a matrix A such that $A = A + xx^T$ and $A = A + \alpha xx^T$
- There are overloads taking a scaling factor `alpha` since it would otherwise be impossible to express $A = A - xx^T$.

Appendix: BLAS 2: Rank-1 Matrix Updates (cont.)

Compute the Hermitian rank-1 (outer product) update of a matrix:

- `void hermitian_matrix_rank_1_update(InVec x, InOutMat A, Triangle t)`
- `void hermitian_matrix_rank_1_update(Scalar alpha, InVec x, InOutMat A, Triangle t)`
- `void hermitian_matrix_rank_1_update(ExecPolicy&&, InVec x, InOutMat A, Triangle t)`
- `void hermitian_matrix_rank_1_update(ExecPolicy&&, Scalar alpha, InVec x, InOutMat A, Triangle t)`

These functions correspond to the BLAS functions `xHER` and `xHPR`.

- Complexity: $O($`x.extent(0)*x.extent(0)`$)$
- Computes: a matrix A such that $A = A + xx^H$ and $A = A + \alpha xx^H$
- There are overloads taking a scaling factor `alpha` since it would otherwise be impossible to express $A = A - xx^H$.

Appendix: BLAS 2: Rank-2 Matrix Updates

Compute the symmetric rank-2 update of a matrix:

- `void symmetric_matrix_rank_2_update(InVec1 x, InVec2 y, InOutMat A, Triangle t)`
- `void symmetric_matrix_rank_2_update(ExecPolicy&&, InVec1 x, InVec2 y, InOutMat A, Triangle t)`

These functions correspond to the BLAS functions `xSYR2` and `xSPR2`.

- Complexity: $O($`x.extent(0)*y.extent(0)`$)$
- Computes: a matrix A such that $A = A + xy^T + yx^T$

Appendix: BLAS 2: Rank-2 Matrix Updates (cont.)

Compute the Hermitian rank-2 update of a matrix:

- `void hermitian_matrix_rank_2_update(InVec1 x, InVec2 y, InOutMat A, Triangle t)`
- `void hermitian_matrix_rank_2_update(ExecPolicy&&, InVec1 x, InVec2 y, InOutMat A, Triangle t)`

These functions correspond to the BLAS functions `xHER2` and `xHPR2`.

- Complexity: $O($`x.extent(0)*y.extent(0)`$)$
- Computes: a matrix A such that $A = A + xy^H + yx^H$

Appendix: BLAS 3: General Matrix-Matrix Product

General matrix-matrix product:

- `void matrix_product(InMat1 A, InMat2 B, OutMat C)`
- `void matrix_product(InMat1 A, InMat2 B, InMat3 E, OutMat C)`
- `void matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, OutMat C)`
- `void matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, InMat3 E, OutMat C)`

These functions correspond to the BLAS functions `xGEMM`.

- Complexity: $O($`A.extent(0)*A.extent(1)*B.extent(1)`$)$
- Computes: $C = AB$ and and $C = E + AB$, C may alias E

Appendix: BLAS 3: Symmetric Matrix-Matrix Product

Symmetric matrix-matrix product:

- `void symmetric_matrix_product(InMat1 A, Triangle t, InMat2 B, OutMat C)`
- `void symmetric_matrix_product(InMat1 A, InMat2 B, Triangle t, OutMat C)`
- `void symmetric_matrix_product(InMat1 A, Triangle t, InMat2 B, InMat3 E, OutMat C)`
- `void symmetric_matrix_product(InMat1 A, InMat2 B, Triangle t, InMat3 E, OutMat C)`
- `void symmetric_matrix_product(ExecPolicy&&, InMat1 A, Triangle t, InMat2 B, OutMat C)`
- `void symmetric_matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, Triangle t, OutMat C)`
- `void symmetric_matrix_product(ExecPolicy&&, InMat1 A, Triangle t, InMat2 B, InMat3 E, OutMat C)`
- `void symmetric_matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, Triangle t, InMat3 E, OutMat C)`

These functions correspond to the BLAS functions `xSYMM`.

- Complexity: $O($`A.extent(0)*A.extent(1)*B.extent(1)`$)$
- Computes: $C = AB$ and and $C = E + AB$, C may alias E

Appendix: BLAS 3: Hermitian Matrix-Matrix Product

Hermitian matrix-matrix product:

- `void hermitian_matrix_product(InMat1 A, Triangle t, InMat2 B, OutMat C)`
- `void hermitian_matrix_product(InMat1 A, InMat2 B, Triangle t, OutMat C)`
- `void hermitian_matrix_product(InMat1 A, Triangle t, InMat2 B, InMat3 E, OutMat C)`
- `void hermitian_matrix_product(InMat1 A, InMat2 B, Triangle t, InMat3 E, OutMat C)`
- `void hermitian_matrix_product(ExecPolicy&&, InMat1 A, Triangle t, InMat2 B, OutMat C)`
- `void hermitian_matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, Triangle t, OutMat C)`
- `void hermitian_matrix_product(ExecPolicy&&, InMat1 A, Triangle t, InMat2 B, InMat3 E, OutMat C)`
- `void hermitian_matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, Triangle t, InMat3 E, OutMat C)`

These functions correspond to the BLAS functions `xHEMM`.

- Complexity: $O($`A.extent(0)*A.extent(1)*B.extent(1)`$)$
- Computes: $C = AB$ and and $C = E + AB$, C may alias E

Appendix: BLAS 3: Triangular Matrix-Matrix Product

Triangular matrix-matrix product:

- `void triangular_matrix_product(InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat C)`
- `void triangular_matrix_product(InMat1 A, InMat2 B, Triangle t, DiagonalStorage d, OutMat C)`
- `void triangular_matrix_product(InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, InMat3 E, OutMat C)`
- `void triangular_matrix_product(InMat1 A, InMat2 B, Triangle t, DiagonalStorage d, InMat3 E, OutMat C)`
- `void triangular_matrix_product(ExecPolicy&&, InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat C)`
- `void triangular_matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, Triangle t, DiagonalStorage d, OutMat C)`
- `void triangular_matrix_product(ExecPolicy&&, InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, InMat3 E, OutMat C)`
- `void triangular_matrix_product(ExecPolicy&&, InMat1 A, InMat2 B, Triangle t, DiagonalStorage d, InMat3 E, OutMat C)`

These functions correspond to the BLAS functions `xTRMM`.

- Complexity: $O($`A.extent(0)*A.extent(1)*B.extent(1)`$)$
- Computes: $C = AB$ and and $C = E + AB$, C may alias E

Appendix: BLAS 3: In-place Triangular Matrix-Matrix Product

In-place triangular matrix-matrix product:

- `void triangular_matrix_left_product(InMat A, Triangle t, DiagonalStorage d, InOutMat C)`
- `void triangular_matrix_right_product(InMat A, Triangle t, DiagonalStorage d, InOutMat C)`
- `void triangular_matrix_left_product(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutMat C)`
- `void triangular_matrix_right_product(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutMat C)`

These functions correspond to the BLAS functions `xTRMM`.

- Complexity: $O($`A.extent(0)*A.extent(1)*C.extent(1)`$)$
- Left product computes: $C = AC$
- Right product computes: $C = CA$

Appendix: BLAS 3: Rank-k Matrix Updates

Rank-k update of a symmetric matrix:

- `void symmetric_matrix_rank_k_update(InMat A, InOutMat C, Triangle t)`
- `void symmetric_matrix_rank_k_update(Scalar alpha, InMat A, InOutMat C, Triangle t)`
- `void symmetric_matrix_rank_k_update(ExecPolicy&&, InMat A, InOutMat C, Triangle t)`
- `void symmetric_matrix_rank_k_update(ExecPolicy&&, Scalar alpha, InMat A, InOutMat C, Triangle t)`

These functions correspond to the BLAS functions `xSYRK`.

- Complexity: $O($`A.extent(0)*A.extent(1)*C.extent(0)`$)$
- Computes: $C = C + AA^T$ or $C = C + \alpha AA^T$, α (i.e., `alpha`) is a scalar

Appendix: BLAS 3: Rank-k Matrix Updates (cont.)

Rank-k update of a Hermitian matrix:

- `void hermitian_matrix_rank_k_update(InMat A, InOutMat C, Triangle t)`
- `void hermitian_matrix_rank_k_update(Scalar alpha, InMat A, InOutMat C, Triangle t)`
- `void hermitian_matrix_rank_k_update(ExecPolicy&&, InMat A, InOutMat C, Triangle t)`
- `void hermitian_matrix_rank_k_update(ExecPolicy&&, Scalar alpha, InMat A, InOutMat C, Triangle t)`

These functions correspond to the BLAS functions `xHERK`.

- Computes: $C = C + AA^H$ or $C = C + \alpha AA^H$, α (i.e., `alpha`) is a scalar

Appendix: BLAS 3: Rank-2k Matrix Updates

Rank-2k update of a symmetric matrix:

- `void symmetric_matrix_rank_2k_update(InMat1 A, InMat2 B, InOutMat C, Triangle t)`
- `void symmetric_matrix_rank_2k_update(ExecPolicy&&, InMat1 A, InMat2 B, InOutMat C, Triangle t)`

These functions correspond to the BLAS functions `xSYRK2`.

- Complexity: $O($`A.extent(0)*A.extent(1)*C.extent(0)`$)$
- Computes: $C = C + AB^T + BA^T$

Appendix: BLAS 3: Rank-2k Matrix Updates (cont.)

Rank-2k update of a Hermitian matrix:

- `void hermitian_matrix_rank_2k_update(InMat1 A, InMat2 B, InOutMat C, Triangle t)`
- `void hermitian_matrix_rank_2k_update(ExecPolicy&&, InMat1 A, InMat2 B, InOutMat C, Triangle t)`

These functions correspond to the BLAS functions `xHERK2`.

- Complexity: $O($`A.extent(0)*A.extent(1)*C.extent(0)`$)$
- Computes: $C = C + AB^T + BA^T$

Appendix: BLAS 3: Triangular Matrix-Matrix Solve

Triangular matrix-matrix left solve:

- `void triangular_matrix_matrix_left_solve(InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X)`
- `void triangular_matrix_matrix_left_solve(InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X, BinaryDivideOp divide)`
- `void triangular_matrix_matrix_left_solve(ExecPolicy&&, InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X)`
- `void triangular_matrix_matrix_left_solve(ExecPolicy&&, InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X, BinaryDivideOp divide)`

These functions correspond to the BLAS functions `xTSRM`.

- Complexity: $O($`A.extent(0)*X.extent(1)*X.extent(1)`$)$
- Computes: X such that $AX = B$

Appendix: BLAS 3: Triangular Matrix-Matrix Solve (cont.)

Triangular matrix-matrix right solve:

- `void triangular_matrix_matrix_right_solve(InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X)`
- `void triangular_matrix_matrix_right_solve(InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X, BinaryDivideOp divide)`
- `void triangular_matrix_matrix_right_solve(ExecPolicy&&, InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X)`
- `void triangular_matrix_matrix_right_solve(ExecPolicy&&, InMat1 A, Triangle t, DiagonalStorage d, InMat2 B, OutMat X, BinaryDivideOp divide)`

These functions correspond to the BLAS functions `xTSRM`.

- Complexity: $O($`B.extent(0)*B.extent(1)*A.extent(1)`$)$
- Computes: X such that $XA = B$

Appendix: BLAS 3: In-place Triangular Matrix-Matrix Solve

In-place triangular matrix-matrix left solve:

- `void triangular_matrix_matrix_left_solve(InMat A, Triangle t, DiagonalStorage d, InOutMat B)`
- `void triangular_matrix_matrix_left_solve(InMat A, Triangle t, DiagonalStorage d, InOutMat B, BinaryDivideOp divide)`
- `void triangular_matrix_matrix_left_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutMat B)`
- `void triangular_matrix_matrix_left_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutMat B, BinaryDivideOp divide)`

These functions correspond to the BLAS functions `xTSRM`.

- Complexity: $O($`A.extent(0)*A.extent(1)*B.extent(1)`$)$
- Computes: X such that $AX = B$

Appendix: BLAS 3: In-place Triangular Matrix-Matrix Solve (cont.)

In-place triangular matrix-matrix right solve:

- `void triangular_matrix_matrix_right_solve(InMat A, Triangle t, DiagonalStorage d, InOutMat B)`
- `void triangular_matrix_matrix_right_solve(InMat A, Triangle t, DiagonalStorage d, InOutMat B, BinaryDivideOp divide)`
- `void triangular_matrix_matrix_right_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutMat B)`
- `void triangular_matrix_matrix_right_solve(ExecPolicy&&, InMat A, Triangle t, DiagonalStorage d, InOutMat B, BinaryDivideOp divide)`

These functions correspond to the BLAS functions `xTSRM`.

- Complexity: $O($`A.extent(0)*A.extent(1)*B.extent(1)`$)$
- Computes: X such that $XA = B$