Multidimensional Arrays in C++

A 2024 Compute Ontario Colloquia Presentation

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

SHARCNET / University of Windsor
Windsor, Ontario, Canada

Sept. 25, 2024

Land Acknowledgement

I am located at the University of Windsor.

The Univesity 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 detroits– of Detroit.

Abstract

The C++ 2023 standard has `std::mdspan` which provides a light-weight **non-owning multidimensional view** of a contiguous single-dimensioned **array**. This enables the reinterpretation of an underlying contiguous array as a multi-dimensional array with support for **different memory layouts** (e.g., C and Fortran) and **accessing elements** (e.g., directly, using atomics, etc.). As found in other programming languages, **subset views** / "slices" are also possible. One does not need to use the latest compiler tools or C++ standard: one can use **`mdspan`** using a reference implementation that has been backported to C++17 and C++14 as well with NVIDIA's CUDA and NVHPC tools. This talk will discuss how to **use `mdspan`** in C++ programs.

Presentation Overview

a) Overview of `mdspan`.

b) Features of `mdspan` and `submdspan`.

c) Example uses:
   - a simple `mdspan` program (C++23)
   - computing the transpose using `mdspan` (C++23, C++20, C++17, C++14)
   - matrix multiplication (`mdspan` and `submdspan`)
   - an NVIDIA HPC SDK example (`mdspan`)

Overview

An `mdspan`:

- is a view of a multidimensional array of elements
  - **Unless** one uses a **custom-written accessor policy** supporting such, **all elements** of the array **must be contiguous**.
- represents a multidimensional index space that is a Cartesian product of integer **intervals**
- each **interval** is a half-open range $[L_i, U_i)$ where L_i and U_i are the lower and upper bounds of the ith interval dimension
- has a **rank** which is the number of interval dimensions
- each dimension's **extent** (size) is:
  - $U_i - L_i$ when the `mdspan`'s rank exceeds 0
- has a **rank index** $[0, rank)$ (used to access each dimension's size)
- is **trivially copyable** if its `accessor_type`, `mapping_type`, and `data_handle_type` (which are determined by template parameters) are all trivially copyable

[1, 7, 9]

Overview (con't)

C++20 has a ***single*-dimension view** of an array called **`span`**:

- If one doesn't need multiple dimensioned access of a contiguous sequence of objects, consider using `span` instead.
- `span` is a pseudo-container complete with iterators.
  - `mdspan` does not have iterators
  - Why? How should one iterate over any multidimensional array?

C++17 has a **read-only *single*-dimension view** of an **array of characters** called `string_view`s designed for **string processing**.

- Seriously consider using `string_view`s when processing read-only strings.

Overview: ISO C++

- C++20 **deprecated** the (immediate) use of the **comma operator inside the array operator**
  - To use pre-C++23 array operators containing expressions with commas, e.g., `some_array[q,r,s]`, in C++23 or newer code, place the comma(s) expression inside of parentheses, e.g., `some_array[(q,r,s)]`.
- C++23:
  - defined `std::mdspan` in `#include <mdspan>`
  - changed **array operator overloading** to support **multiple indices**
  - **no changes** were made to array/vector **declarations**
- C++26 will define `std::submdspan()`
  - Supports **slices** (multidimensional subsets) of `std::mdspan`.
  - (Time ran out for `std::submdspan()` to be in C++23 so it should be in C++26.)

Overview: Concerning ISO C++26…

Although C++26 is not yet an ISO C++ standard, it is expected that C++26:

- will define additional layouts for `mdspan` supporting **BLAS object layouts**
- will have support for **linear algebra**
  - `#include <linalg>`
  - It will include types and calls able to use template types like the C++ Standard Library does.
    - Unlike BLAS where calls are hard-coded to use specific types.
    - It is possible that such will also have support for executors (sequential and parallel executions), e.g., like C++ parallel algorithms' `std::execution::par_unseq`, etc.

(NOTE: This talk is not about `#include <linalg>` and its implementations, e.g., [3, 4, 6].)

Overview: mdspan Reference Implementation

A working `mdspan` reference implementation is **available**:

- `git clone https://github.com/kokkos/mdspan.git` [5]
- There is also a Spack package, e.g., run `spack install mdspan` and activate such in your Spack environment.

Highlights:

- **C++14, C++17, C++20** and newer standards are **supported**
  - pre-C++23 code uses the function call operator instead of the array operator
    - e.g., `ar(1,6,2)` *versus* `ar[1,6,2]`
- To use the reference implementation:
  - `git clone https://github.com/kokkos/mdspan.git`, and, compile code with include path also referring to `mdspan/include/` subdirectory
  - `#include <experimental/mdspan>`
  - Before C++23, use namespace `std::experimental`.
- usable in **regular** C++ code as well in **CUDA** and **HIP** C++ codes
- Also provides should-be-in-C++26's **`submdspan`**.

Overview: NVIDIA NVHPC

NVIDIA's HPC SDK allows one to compile ISO C++ and Fortran code to run on CPU and/or GPU. [6]

Experimental features enabled by using a suitably set `-stdpar` compiler option:

- `mdspan` [7]
- `submdspan` [9]
- `mdarray` [8]
- linear algebra [3]
- senders and receivers / `std::execution` (C++20 limited support) [2]

with all of these having C++23 recommended and C++17 limited support except where indicated.

Overview: NVIDIA NVHPC (con't)

To use on our clusters:

- Load the module: `module load nvhpc/23.7` (or newer)
- Until C++23, etc. is sufficiently supported by the NVIDIA compiler:
  - use the function call operator to access multidimensional array elements
  - use the `<experimental/XXXX>` headers and `std::experimental` namespaces

Example build commands:

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

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

- Pascal `-gpu=cu60`; Volta: `-gpu=cu70`; Turing: `-gpu=cu75`; Ampere: `-gpu=cu80`; Hopper: `-gpu=cu90`

Features: mdspan

`mdspan` like most types in C++ is a class template with the following parameters:

```cpp
template <
  typename T,
  typename Extents,
  typename LayoutPolicy = std::layout_right,
  typename AccessorPolicy = std::default_accessor<T>
> class mdspan;
```

where:

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

[7]

Features: mdspan: Element Type

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

NOTE:

- **`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*
- **`mdspan`s are *not* containers**
  - containers are "owning": they allocate and deallocate the data they contain and (almost always) have *value semantics*
- C++23 `mdspan`s using C++23 defaults imply the underlying multidimensional array is **contiguous**
  - other layout and accessor policies permit the underlying array to be non-contiguous

Features: mdspan: 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, i.e., static, then the dimension size must be an integer
    - if known only at run-time, i.e., dynamic, then the dimension size must be `std::dynamic_extent`
    - if all dimensions are dynamic, then `std::dextents` can instead be used to specify all dimensions
- is `regular` and is **trivially copyable**

Features: mdspan: Extents (con't)

Examples:

- 5x3x15: `std::extents<std::size_t, 5, 3, 15>`
- 5x?x15: `std::extents<std::size_t, 5, std::dymamic_extent, 15>`
- 5x?x?: `std::extents<std::size_t, 5, std::dymamic_extent, std::dynamic_extent>`
- ?x?x?: `std::extents<std::size_t, std::dynamic_extent, std::dymamic_extent, std::dynamic_extent>`
- ?x?x?: `std::dextents<std::size_t, 3>`

Features: mdspan: 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 (rightmost extent has stride 1), e.g., C layout
- **`std::layout_left`:** column-major multidimensional array layout (leftmost extent has stride 1), e.g., Fortran layout
- **`std::layout_stride`:** a layout mapping policy with user-defined strides

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

Features: mdspan: Accessor Policy

Every `mdspan` has a policy type that defines how elements are accessed.

In C++23 there is one accessor policy:

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

Why have accessors?

- They can provide information to the compiler concerning **data access**.
- They can provide information to the compiler concerning **special ways of accessing data**.
- They can be used to **enable heterogeneous memory access**, e.g., across distinct address spaces (GPU RAM, other nodes' RAM, PGAS models, etc.).

Features: mdspan: CTAD

- is an acronym for **Class Template Argument Deduction**
- enables the compiler, where applicable, to **deduce template parameters** from arguments' passed to the constructor being used
- is a language feature available with **C++17 or newer**
  - When this can be used, one is able to omit all template parameters when writing a declaration.
  - If this cannot be used, then one must specify the template parameters when writing a declaration of that type.

e.g.,

```
1 vector u(1000); // ERROR: Element type cannot be deduced.
2 vector v(2*3*4, 0.0); // e.g., v is vector<double>
3 mdspan mds{ v.data(), 2, 3, 4 };
```

Features: submdspan: Overview

`submdspan` is a **function** that allows one to obtain **an `mdspan`** that is a **subset** of an existing `mdspan`, e.g., a row/column of a matrix. [9]

NOTE: `submdspan` and associated types are expected to be in C++26 and is in the reference implementation referred to in this presentation.

Features: submdspan: Overview (con't)

A subset of a dimension can be specified as follows:

- using a **single integral value**
  - the value refers to the use of **that 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** $[start, stop)$ of index values to use for that dimension
- using `std::strided_slice{a,b,c}`
  - `a` is the **start**ing (offset) index for that dimension
  - `b` is the **extent** (size) for that dimension
  - `c` is the **stride** (increment/step) starting at the starting index for that dimension
- using `std::full_extent`
  - this value uses **all index values** for that dimension

Features: submdspan: Overview (con't)

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

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

however, the design of `submdspan` is to use:

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

i.e., C++ is equivalent to "`a( start : stop-start : step )`".

Features: submdspan: Overview (con't)

Why use the equivalent of (start:extent:step)?

- A **dynamic** starting value can be combined with a **static** extent which can be easily used to **generate** a **static** extent.

Features: submdspan: Example 1

Given:

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

Features: submdspan: Example 2

Given:

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

Features: submdspan: Example 3

Given:

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

Features: submdspan: Example 4

Given:

```
1 vector v{10, 0.0};              // vector of 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` … etc.

Features: submdspan: Example 5

Given:

```
1 vector v{10*10, 0.0};             // vector of 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[i] == m[i,3]` for some `i`

Features: submdspan: Example 6

Given:

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

Example 1: A simple use of mdspan: Using C++23

At the time of this presentation these major compilers support `std::mdspan` as an ISO C++23 library feature:

- Clang version 17 (partial)
- Clang version 18
- Microsoft Visual C++ version 19.39 (Windows only)
- XCode version 15.0.0 (Apple only)
- NVIDIA NVHPC version 22.11 (or newer)

otherwise one will need to use a reference implementation.

Let's first look at a simple C++23 example using the reference implementation.

Example 1: A simple use of mdspan: Using C++23 (con't)

ex01-refimpl-cpp23.cpp

```cpp
#include <cstddef>                      // for size_t
#include <vector>                       // for vector
#include <experimental/mdspan>          // for mdspan

using namespace std;                    // for simplicity below
constexpr size_t N = 100;

int main() {
  vector<double> in(N*N);               // allocate NxN linear array for input
  vector<double> out(N*N);              // allocate NxN linear array for output
  mdspan A{in.data(), N, N};            // A is a multidimensional view of in array
  mdspan B{out.data(), N, N};           // B is a multidimensional view of out array
  for (size_t i=0; i < N; ++i)
    for (size_t j=0; j < N; ++j)
      B[j,i] = A[i,j];                  // i.e., transpose element
}
```

Example 1: A simple use of mdspan: Using C++23 (con't)

Build the example code using a recent version of GCC:

- `git clone https://github.com/kokkos/mdspan.git`
- `g++ -std=c++23 -I./mdspan/include ex01-refimpl-cpp23.cpp`

Example 2: Transpose: Overview

This program:

- defines a `print()` function that outputs a two-dimensional `mdspan` passed to it
- defines a `transpose(in,out)` function where:
  - `in` is a **2D `mdspan`** representing an **input** matrix
  - `out` is a **2D `mdspan`** representing an **output** matrix
  - the function computes the matrix **transpose** of `in` storing the result in `out`
- `main()` has code that calls `print()` and `transpose()`

Example 2: Transpose: Reference Impl. Using C++23

- When using C++23, the `std` namespace is used to access `mdspan`.
- GCC compile: `g++ -std=c++23 -Ipath/to/mdspan/include ex02-refimpl-cpp23.cpp`

ex02-refimpl-cpp23.cpp

```cpp
#include <cassert>                  // for assert
#include <cstddef>                  // for size_t
#include <experimental/mdspan>      // for mdspan
#include <iostream>                 // for cout
#include <numeric>                  // for iota
#include <vector>                   // for vector
using namespace std;                // for simplicity below
void print(auto mat2d) {
  assert(mat2d.rank() == 2);
  for (size_t i=0; i < mat2d.extent(0); ++i) {
    for (size_t j=0; j < mat2d.extent(1); ++j)
      cout << mat2d[i,j] << ' ';
    cout << '\n';
  }
  cout << '\n';
}
```

Example 2: Transpose: Reference Impl. Using C++23 (con't)

```
17 auto transpose(auto in, auto out) {
18   assert(in.rank() == 2 && out.rank() == 2);
19   assert(in.extent(0) == out.extent(1) && in.extent(1) == out.extent(0));
20   for (size_t i=0; i < in.extent(0); ++i)
21     for (size_t j=0; j < in.extent(1); ++j)
22       out[j,i] = in[i,j];
23 }
24
25 int main() {
26   vector<double> in(5*3);                          // allocate linear 5 by 3 array
27   vector<double> out(5*3, 0.0);                    //   populate "out" with zero values
28   iota(in.begin(), in.end(), 0.0);                 //   populate "in" with values 0 to 14
29   mdspan A{ in.data(), 5, 3 };                     // treat in as a 5x3 2D matrix
30   mdspan B{ out.data(), A.extent(1), A.extent(0) }; // "out" is 3x5 2D matrix
31   print(A);                                        // output A mdspan
32   transpose(A,B);                                  // copy transpose of A into B
33   print(B);                                        // output B mdspan
34 }
```

Example 2: Transpose: Reference Impl. Using C++23 (con't)

ex02-refimpl-cpp23.exe.txt

```
0 1 2
3 4 5
6 7 8
9 10 11
12 13 14

0 3 6 9 12
1 4 7 10 13
2 5 8 11 14

```

Example 2: Transpose: Reference Impl. Using C++20

The same example with C++20:

- uses the function call operator instead of the array operator to access indices:
  - In `print()`:
    - C++23: `cout << mat2d[i,j] << ' ';`
    - pre-C++23: `cout << mat2d(i,j) << ' ';`
  - In `transpose()`:
    - C++23: `out[j,i] = in[i,j];`
    - pre-C++23: `out(j,i) = in(i,j);`
- use the `std::experimental` namespace for `mdspan`
  - e.g., add `using std::experimental::mdspan;`
- GCC compile: `g++ -std=c++20 -Ipath/to/mdspan/include ex02-refimplcpp20.cpp`

Example 2: Transpose: Reference Impl. Using C++17

The same example with C++17:

- has all changes in the C++20 example
- C++17 does not support `auto` function parameters so such needs to be replaced by template type placeholders
  - Each `mdspan` function parameter has a different type so distinct template type placeholders need to be used.
  - e.g., `template <typename T> void print(T mat2d)`
  - e.g., `template <typename T, typename U> void transpose(T in, T out)`
- GCC compile: `g++ -std=c++17 -Ipath/to/mdspan/include ex02-refimplcpp17.cpp`

Example 2: Transpose: Reference Impl. Using C++14

The same example with C++14:

- has all changes in the C++17 example
- C++14 does not support C++ template argument deduction with classes so the type `mdspan` requires its template arguments to be specified.
  - e.g., `mdspan<double,dextents<size_t,2>>` instead of `mdspan`
- since `dextents` is being used, ensure it is brought into the current name space, e.g.,
  - add `using std::experimental::dextents;`
- GCC compile: `g++ -std=c++14 -Ipath/to/mdspan/include ex02-refimplcpp14.cpp`

Example 3: Matrix Multiplication: Dot Product

ex03-matmul-cpp23.cpp

```cpp
// a and b are 1D mdspans...
auto dot_product(auto a, auto b)
  requires (decltype(a)::rank() == 1) && (decltype(b)::rank() == 1)
{
  assert(a.extent(0) == b.extent(0));

  using rettype =
    std::common_type_t<typename decltype(a)::element_type, typename decltype(b)::element_type>
  ;
  using a_index = typename decltype(a)::index_type;
  using b_index = typename decltype(b)::index_type;

  a_index ai{};
  b_index bi{};
  rettype retval{};
  for (; ai ≠ a.extent(0); ++ai, ++bi)
    retval += a[ai] * b[bi];
  return retval;
}
```

Example 3: Matrix Multiplication: Extract Row

ex03-matmul-cpp23.cpp

```cpp
// given a 2D mdspan called mds, return ith row as a 1D mdspan...
auto row(auto mds, typename decltype(mds)::index_type i)
  requires (decltype(mds)::rank() == 2)
{
  return submdspan(mds, i, full_extent);
}
```

Example 3: Matrix Multiplication: Extract Column

ex03-matmul-cpp23.cpp

```cpp
// given a 2D mdspan called mds, reutrn jth row as a 1D mdspan...
auto col(auto mds, typename decltype(mds)::index_type j)
  requires (decltype(mds)::rank() == 2)
{
  return submdspan(mds, full_extent, j);
}
```

Example 3: Matrix Multiplication: 2D Matrix Multiplication

ex03-matmul-cpp23.cpp

```cpp
// given input mdspan "matrices" a and b, output/return mdspan "matrix" c
// compute c = a*b (i.e., matrix multiplication) using dot_product()...
void matmul(auto c, auto a, auto b)
  requires (decltype(a)::rank() == 2) && (decltype(b)::rank() == 2) && (decltype(c)::rank() == 2)
{
  assert(a.extent(1) == b.extent(0));
  assert(c.extent(0) == a.extent(0) && c.extent(1) == b.extent(1));

  using a_index = typename decltype(a)::index_type;
  using b_index = typename decltype(b)::index_type;
  for (a_index i{}; i != a.extent(0); ++i)
  {
    for (b_index j{}; j != b.extent(1); ++j)
      c[i,j] = dot_product( row(a,i), col(b,j) );
  }
}
```

Example 3: Matrix Multiplication: main() function

ex03-matmul-cpp23.cpp

```
61 int main()
62 {
63   vector<double> a(4*3);                // 4x3 linear "matrix"
64   iota(a.begin(), a.end(), 0.0);        // set some values: 0.0 to 11.0
65   mdspan am{ a.data(), 4, 3 };          // 2D mdspan view of a
66
67   vector<double> b(3*7);                // 3x7 linear "matrix"
68   iota(b.begin(), b.end(), 0.0);        // set some values: 0.0 to 20.0
69   mdspan bm{ b.data(), 3, 7 };          // 2D mdspan view of b
70
71   vector c(4*7, 0.0);                   // 4x7 linear "matrix", all values zero
72   mdspan cm{ c.data(), 4, 7 };          // 2D mdspan view of c
73
74   matmul(cm,am,bm);                     // multiply am and bm, place result in cm
75 }
```

Example 4: NVIDIA mdspan And linalg Use

ex04-nvhpc-cpp23.cpp

```cpp
//
// Example compilation lines...
//   * CPU: nvc++ -stdpar=multicore --c++23 ex04-nvhpc-linalg.cpp -lblas
//   * GPU: nvc++ -stdpar=gpu --c++23 ex04-nvhpc-linalg.cpp -cudalib=cublas
//
// NOTE: This program intentionally performs meaningless calculations and
//       only demonstrates the types and calls being used.
//

#include <array>                        // for std::array
#include <execution>                    // for std::execution
#include <iostream>                     // for std::cout
#include <numbers>                      // for mathematical constants
#include <vector>                       // for std::vector
#include <experimental/mdspan>          // for mdspan
#include <experimental/linalg>          // for linalg

using namespace std;                    // for simplicity below
namespace stdn = std::numbers;          // for brevity below
```

Example 4: NVIDIA mdspan And linalg Use (con't)

```
20 namespace stdex = std::experimental;      // for brevity below
21 namespace linalg = stdex::linalg;         // for brevity below
22
23 constexpr size_t N = 8;                   // an extent (e.g., # of rows)
24 constexpr size_t M = 4;                   // an extent (e.g., # of columns)
25
26 int main()
27 {
28   vector<double> a(N*M);
29   stdex::mdspan A(a.data(), N, M);
30   for (size_t i = 0; i < A.extent(0); ++i)
31     for (size_t j = 0; j < A.extent(1); ++j)
32       A(i,j) = stdn::e * i + j;           // demo but meaningless computation
33
34   vector<double> x(M);
35   stdex::mdspan X(x.data(), M);
36   for (size_t j = 0; j < X.extent(0); ++j)
37     X(j) = stdn::phi * j;                 // demo but meaningless computation
38
39   array<double,N> y(N);
```

Example 4: NVIDIA mdspan And linalg Use (con't)

```
40   stdex::mdspan Y(y.data(), N);
41   for (size_t i = 0; i < Y.extent(0); ++i)
42     Y(i) = -stdn::pi * i;                  // demo but meaningless computation
43
44   // compute Y = A*X
45   linalg::matrix_vector_product(A, X, Y);
46
47   // compute Y = (0.5*A)*X + (2.0*Y) in parallel
48   // e.g., linalg::matrix_vector_product(ExecPolicy, inMatrix, InVector1, InVector2, OutVector)
49   linalg::matrix_vector_product(execution::par, linalg::scaled(0.5, A), X, linalg::scaled(2.0, Y), Y);
50
51   // Output computed values...
52   for (size_t i = 0; i < N; ++i)
53     cout << "Y[" << i << "] = " << Y(i) << '\n';
54   return 0;
55 }
56
```

Example 4: NVIDIA mdspan And linalg Use (con't)

ex04-nvhpc-cpp23.exe.txt

```
Y[0] = 56.6312
Y[1] = 122.605
Y[2] = 188.579
Y[3] = 254.553
Y[4] = 320.528
Y[5] = 386.502
Y[6] = 452.476
Y[7] = 518.45
```

References

[1] cppreference.com. *C and C++ Reference*. URL: http://en.cppreference.com/w/ (visited on 09/12/2024) (cit. on p. 5).

[2] M. Dominiak, G. Evtushenko, L. Baker, L. Radu Teodorescu, L. Howes, K. Shoop, M. Garland, E. Niebler, and B. Adelstein Lelbach. *std::execution*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html (visited on 09/12/2024) (cit. on p. 10).

[3] M. Hoemmen, D. Hollman, C. Trott, D. Sunderland, N. Liber, A. Klinvex, L.-T. Lo, D. Lebrun-Gradie, G. Lopez, P. Caday, S. Knepper, P. Luszczek, and T. Costa. *A free function linear algebra interface based on the BLAS*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1673r13.html (visited on 09/12/2024) (cit. on pp. 8, 10).

[4] kokkos.org. *P1673 reference implementation*. URL: https://github.com/kokkos/stdBLAS (visited on 09/12/2024) (cit. on p. 8).

References (con't)

[5] kokkos.org. *Reference mdspan implementation*. URL: https://github.com/kokkos/mdspan (visited on 09/12/2024) (cit. on p. 9).

[6] nvidia.com. *NVIDIA HPC SDK Release Notes*. URL: https://docs.nvidia.com/hpc-sdk/archive/22.11/pdf/hpc-sdk2211rn.pdf (visited on 09/12/2024) (cit. on pp. 8, 10).

[7] C. Trott, D. Hollman, D. Lebrun-Grandie, M. Hoemmem, D. Sunderland, H. C. Edwards, B. Adelstein Lelbach, M. Bianco, B. Sander, A. Iliopoulos, J. Michopoulos, and N. Liber. *MDSPAN (P0009r18)*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0009r18.html (visited on 09/12/2024) (cit. on pp. 5, 10, 12).

References (con't)

[8] C. Trott, D. Hollman, M. Hoemmem, D. Sunderland, and D. Lebrun-Grandie. *mdarray: An Owning Multidimensional Array Analog of mdspan*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p1684r5.html (visited on 09/12/2024) (cit. on p. 10).

[9] C. Trott, D. Lebrun-Grandie, M. Hoemmen, and N. Liber. *Submdspan (P2630r4)*. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2630r4.html (visited on 09/12/2024) (cit. on pp. 5, 10, 19).

Part I

Appendix

Example Layout Demonstration

The `std::layout_stride` mapping can be initialized to do what `std::layout_right` and `std::layout_left` do. For example, the following program demonstrates uses `std::layout_stride` and `std::layout_right` layouts in an equivalent manner:

layout01-cpp23.cpp

```cpp
#include <cstddef>                      // for size_t
#include <iostream>                     // for cout
#include <numeric>                      // for iota
#include <vector>                       // for vector
#include <experimental/mdspan>          // for mdspan

using namespace std;                    // for simplicity below

void print(auto mat)
  requires (decltype(mat)::rank() == 3)
{
  cout << "[ ";
  for (size_t i=0; i < mat.extent(0); ++i)
  {
    cout << "[ ";
```

Example Layout Demonstration (con't)

```
16      for (size_t j=0; j < mat.extent(1); ++j)
17      {
18        cout << "[ ";
19        for (size_t k=0; k < mat.extent(2); ++k)
20          cout << mat[i,j,k] << ' ';
21        cout << "]";
22      }
23      cout << " ]";
24    }
25    cout << " ]\n";
26  }
27
28  int main()
29  {
30    vector<int> v(2*3*4);                    // allocate 2x3x4 linear array
31    iota(v.begin(), v.end(), 0);             // populate v with [0..v.size())
32
33    mdspan mdv1{v.data(), 2, 3, 4};          // uses CTAD and default layout
34
35    // explicitly specified mdspan parameters...
```

Example Layout Demonstration (con't)

```
36   mdspan<int, dextents<size_t,3>, layout_right> mdv2{ v.data(), 2, 3, 4 };
37   mdspan<int, dextents<size_t,3>, layout_stride> mdv3{
38     v.data(), { dextents<size_t,3>{2,3,4}, array<size_t,3>{3*4,4,1} }
39   };
40
41   cout << "mdv1... "; print(mdv1);
42   cout << "mdv2... "; print(mdv2);
43   cout << "mdv3... "; print(mdv3);
44 }
```

When run, the above program outputs the same values for `mdv1`, `mdv2`, and `mdv3`:

layout01-cpp23.exe.txt

```
1 mdv1... [ [ [ 0 1 2 3 ][ 4 5 6 7 ][ 8 9 10 11 ] ][ [ 12 13 14 15 ][ 16 17 18 19 ][ 20 21 22 23 ] ] ]
2 mdv2... [ [ [ 0 1 2 3 ][ 4 5 6 7 ][ 8 9 10 11 ] ][ [ 12 13 14 15 ][ 16 17 18 19 ][ 20 21 22 23 ] ] ]
3 mdv3... [ [ [ 0 1 2 3 ][ 4 5 6 7 ][ 8 9 10 11 ] ][ [ 12 13 14 15 ][ 16 17 18 19 ][ 20 21 22 23 ] ] ]
```