Performance effects of future-proofing submdspan_mapping

Mark Hoemmen, mhoemmen@nvidia.com

Background

Multidimensional array view (C++23) → `mdspan<`

Const-qualified type of each element → `ElementType,`

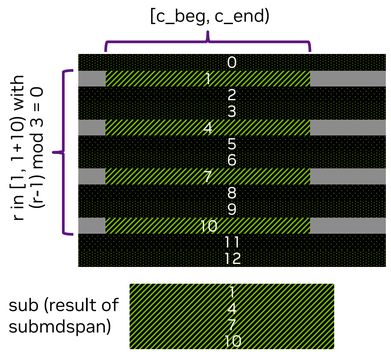
Type of (i,j,k,...) index space → `extents<IndexType, Exts...>,`

Maps (i,j,k,...) -> 1-D index → `Layout,`

Maps (1-D index, data handle) -> reference → `Accessor>`

submdspan: C++26 "slicing"

```
auto sub = submdspan(A,
  strided_slice{
    .offset = 1,
    .stride = 3,
    .extent = 10,
  },
  pair{c_beg, c_end}
);
```



submdspan calls submdspan_mapping

Definition of submdspan

```
auto smr = submdspan_mapping(A.mapping(), slices...);
return mdspan(
  A.accessor().offset(A.data_handle(), smr.offset),
  smr.mapping,
  Accessor::offset_policy(A.accessor()));
```

Users define submdspan_mapping for their custom layout mapping

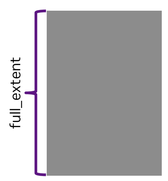
Valid slice types: 4 categories

Full

Anything convertible to full_extent_t

Means [0, extent(r))

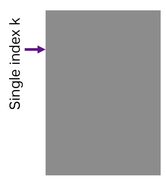
Like colon (:) in Fortran & Matlab



Single index

Anything convertible to mdspan::index_type

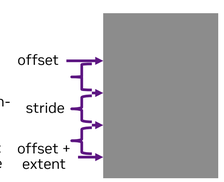
Integers, OR *integral-constant-like* ("compile-time integers") like constant_wrapper & integral_constant



Strided

strided_slice<Offset, Extent, Stride>

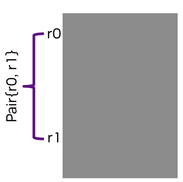
- Differs from Fortran, Matlab, & Python start:end:stride so run-time start can give a compile-time extent
- Offset, Extent, Stride: valid single index type



Contiguous subrange

index-pair-like, OR strided_slice with stride *integral-constant-like* 1

index-pair-like: *tuple-like* (exposition-only concept including std::{tuple,pair}) with tuple_size 2



Design issue & fix

FEATURE: Valid slice types are an open set

- Permit user-defined pair or tuple types (e.g., cuda::std::tuple)
- Index wrappers (to prevent mixing up indices in nested for loops

Later C++ versions may add new slice types

Accidentally, by expanding *tuple-like*

- Current definition: std::array<T,2>, std::complex, std::pair, std::tuple<X, Y>, std::ranges::subrange
- C++26 added std::complex, making it a valid slice type (unintended by submdspan authors)
- Proposals in flight (e.g., P2769) to redefine tuple-like as open set

Deliberately, e.g., by adding Fortran-style "start : end : stride"

BUG: Adding new slice type breaks user-defined submdspan_mapping customizations

- submdspan can call submdspan_mapping with any valid slice type
- Undefined behavior (UB) if submdspan calls it with invalid slice type
- submdspan_mapping NOT required to be ill-formed for invalid slice types
- Bad for custom layout mappings written to earlier C++ versions
- Users might exploit this: e.g., "Fake slice type" that makes custom submdspan_mapping print debug info & returns original mapping

FIX: submdspan "canonicalizes" slices

1. Full → full_extent_t
2. Single index → index_type or cw<Value>
3. Strided → strided_slice (apply single index to members)
4. Contiguous subrange → Strided with stride_type = cw<1>

cw<Value>: constant_wrapper, P2781 (in C++26)

User-defined submdspan_mapping must fail to compile with anything else

Fixed definition of submdspan

```
auto [...canonical_slices] =
  submdspan_canonicalize_slices(src.extents(), slices...);
auto smr = submdspan_mapping(A.mapping(), canonical_slices...);
return mdspan(
  A.accessor().offset(A.data_handle(), smr.offset),
  smr.mapping,
  Accessor::offset_policy(A.accessor()));
```

User code only sees canonical slices

P3663 proposes this fix

- Passed C++ Standard Committee design review; awaits wording review
- Hope to apply to C++26 via either National Body comment or Defect Report

Performance

submdspan may be called in tight loops

```
using space_3d_t = extents<int, dynamic_extent, 3>;
mdspan<float, space_3d_t> points_in_3d_space = /* ... */;
mdspan<float, dims<2, int>> forces = /* ... */;
for (int r = 0; r < points_in_3d_space.extents(0); ++r) {
  auto point_r = submdspan(points_in_3d_space, r, full_extent);
  for (int c = r+1; c < points_in_3d_space.extents(1); ++c) {
    auto point_c = submdspan(points_in_3d_space, c, full_extent);
    forces[r, c] += force(point_r, point_c);
  }
}
```

Loop over N points in 3-D space

$O(N^2)$ calls to submdspan

submdspan MUST be fast or users will bypass it

- Point of mdspan is to write layout- & accessor-generic code
- Bypass → users get raw pointers & do error-prone index arithmetic

Canonicalization copies slices

- May consume hardware resources like GPU registers
- Increased code complexity may hinder inlining
- But fix may even improve run time by simplifying code
  - Fewer types → fewer function instantiations
  - strided_slice is an aggregate (unlike pair or tuple)

2 implementations

1. Fully C++26: pack indexing, structured bindings can introduce a pack
2. C++20 back-port

Both available in a pull request of the reference mdspan implementation: https://github.com/kokkos/mdspan

Benchmarks

- Iterate over all elements by slicing mdspan recursively until rank 1
- submdspan(x, k, uss, full, ..., full, uss) for every row k
  - uss ("unit-stride slice"): either std::pair or cuda::std::pair (for CUDA builds)
  - Force conversions: k → index_type, full → full_extent_t, pair → strided_slice
  - Stay layout_right_padded to avoid layout_stride (which computes & stores std::array)
- element_type is uint8_t: minimize memory; overflow-free integer arithmetic
- index_type={int, size_t} x all {static, dynamic} extents: {2, 2, 2, 2, 2, 2}
- Time in seconds, averaged over many iterations
- Hardware
  - NVIDIA H100 GPU
  - AMD EPYC 7232 CPU (8 cores, 2 threads/core)

Results

int with static extents	GCC 15.1	Clang 21	nvc++ 25.5	CUDA 13 + GCC 14.3
Baseline	1.13E-09	1.21E-07	2.25E-08	2.71E-08
C++26	1.16E-09	3.14E-09		
C++20	1.16E-09	3.14E-09	5.28E-08	2.75E-08

int with dynamic extents	GCC 15.1	Clang 21	nvc++ 25.5	CUDA 13 + GCC 14.3
Baseline	1.17E-07	2.09E-07	1.05E-07	6.83E-08
C++26	1.91E-07	7.01E-07		
C++20	1.93E-07	7.03E-07	5.37E-07	3.33E-08

size_t with static extents	GCC 15.1	Clang 21	nvc++ 25.5	CUDA 13 + GCC 14.3
Baseline	1.32E-09	3.14E-09	3.45E-09	3.22E-08
C++26	1.16E-09	3.14E-09		
C++20	1.16E-09	3.14E-09	4.76E-08	3.14E-08

size_t with dynamic extents	GCC 15.1	Clang 21	nvc++ 25.5	CUDA 13 + GCC 14.3
Baseline	1.23E-07	8.24E-08	9.70E-08	8.04E-08
C++26	1.50E-07	7.04E-08		
C++20	1.50E-07	7.06E-08	5.45E-07	3.31E-08