Modernizing C++ Interfaces with Concepts, Constraints and std::mdspan

Christian Trott

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

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.

Modern C++: Prevent & Detect Mistakes

Biggest philosophical change from C++98:

Abstractions that help avoid bugs

- Smart pointers instead of raw pointers
- Ranges instead of iterators
- Concepts and constraints instead of unconstrained templates

Enforced Safety **Checkable Conditions** **Early error detection**

This talk: 3 C++ Capabilities relevant to HPC

Archetypical HPC Software: BLAS

```
// Matrix
int    nRows  = ...;
int    nCols  = ...;
double * A = new double [nRows*nCols];
// Vectors
double * y = new double [nRows];
double * x = new double[nCols];

// y = 1.0*A*x;
dgemv('N', nRows, nCols, 1.0, A, nRows, x, 1, 0.0, y, 1);
```

- Lets unpack the 11 !! parameters to compute `y = A*x`:
  - `N`: the matrix is not transposed
  - `nRows`: Number of Rows (also length of `y`)
  - `nCols`: Number of Columns (also length of `x`)
  - `1.0`: scaling factor for `A`
  - `A`: pointer to the matrix values
  - `nRows`: stride of the rows of `A`
  - `x`: right hand side vector
  - `1`: stride of `x`
  - `0.0`: scaling of `y`
  - `y`: left hand side vector
  - `1`: stride of `y`

What is wrong with that?

- *Many parameters of the same type*
  - Easy to switch order
- *Parameters which only together describe an actual data structure*
  - Matrix == ptr + num_rows + num_cols + stride
- *Implicit assumption of a storage order*
  - The matrix better be in column major
- *Implicit assumption that object sizes match*
  - No separate values for size of A, x, and y storage
  - No way for implementation of dgemv to check validity of inputs

We need to do better!

***Three Modern C++ capabilities for interface design:* std::mdspan, constraints, *and* concepts**

Benefits of more explicit and checkable interfaces

Developers of libraries (and library like internals of applications):

- Self document and enforce requirements on functions
- Improved organization of overload sets

Users of libraries

- More well defined interfaces – specification as part of the interface instead of just documentation
- Catch mistakes at compile time instead of debugging code at runtime

AI coding assistants

- Enforced requirements in interfaces, guide code generation
- Compile time error feedback helps agents iterate
- No need to correctly connect documentation with code lines

mdspan – Multidimensional Arrays for C++

Multi dimensional array enabling the design features of Kokkos Views

- Compile time rank
- Mixed static and dynamic extents
- Configurable layout
- Non-owning – i.e. wraps existing allocations
- Reference semantics: a **const** mdspan of **non-const** ElementType has modifiable data

```
template<class ElementType, class Extents, class Layout, class Accessor>
class mdspan;
```

- **ElementType**: fundamental scalar type
- **Extents**: runtime and compile time extents
  - std::extents<int, 5,std::dynamic_extent,4> => 5xNx4 3D Array using int as index type
- **Layout**: the mapping from multi-dimensional index to memory offset
- **Accessor**: pointer type and how to generate element reference from pointer and offset

Reference implementation: https://github.com/kokkos/mdspan

Creating an mdspan

- Wraps existing allocation
- For many cases CTAD eliminates need to specify template args
- Designed for interoperability with any existing data allocations

```
double* ptr = new double[N*M];

// 2D layout_right (C-Layout), size_t as index type
mdspan a(ptr, N, M);

// N batched 3x3x3 tensors with unsigned as index_type
mdspan<double, extents<unsigned, dynamic_extent, 3,3,3>> a(ptr, N);

// 2D layout_left (Fortran Layout) with one compile time dimension
// using int for index calculations
mdspan<double, extents<int, dynamic_extent, 8>, layout_left> b(ptr, N);
```

Example Impact on Usage

Update C-like interface to mdspan

(takes matrix, row-major - contiguous storage)

Before:

```
double val = MyLib::matrix_norm(ptr, N, M);
```

After:

```
double val = MyLib::matrix_norm(mdspan(ptr, N, M));
```

Accessing Data and Assignment Rules

```
mdspan<double, dextents<int, 2>> matrix(ptr, N, M);

// access data with []
matrix[3, 7] = 5;

// assign non-const to const
mdspan<const double, dextents<int, 2>> const_matrix = matrix;

using mdspan_4x4 = mdspan<double, extents<int, 4, 4>>;
// will work if N and M are 4, otherwise throw
mdspan_4x4 m44 = matrix;

using mdspan_left = mdspan<double, dextents<int, 2>, layout_left>;
mdspan_left mleft = matrix; // will not compile
```

Assignment: if logical represents the same data and doesn’t violate pointer const rules assignment works!

Example Impact on Usage

Function that takes column major matrix of doubles:

```
double MyLib::matrix_norm(mdspan<const double, dims<2>, layout_left> a);
```

Valid:

```
mdspan<double, dextents<int, 2>, layout_left>
```

Converts non-const to const, and int index_type to size_t.

Valid:

```
mdspan<const double, extents<int, 16, 16>, layout_left>
```

Converts compile to runtime extents, and int index_type to size_t.

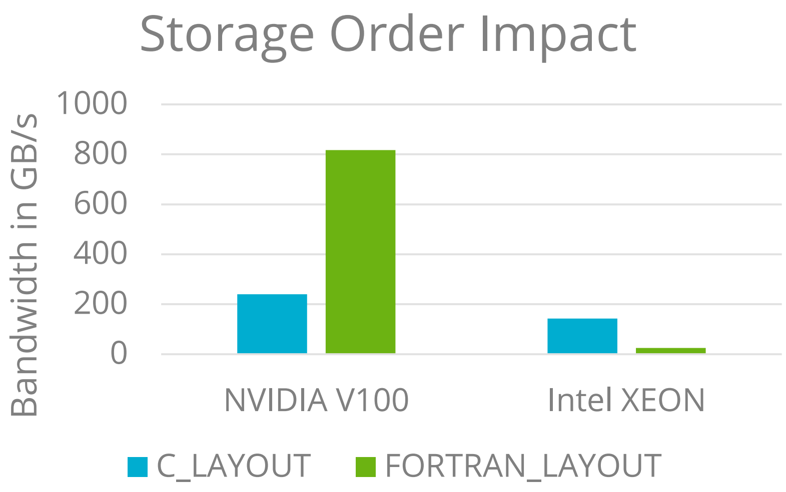
Invalid:

```
mdspan<const double, dims<2>, layout_right>
```

Incompatible assignment from layout_left to layout_right.

Why Layout Matters

```
using mdspan_t = mdspan<float, dextents<int, 2>, LAYOUT>;
void matrix_add( mdspan_t Z, mdspan_t X, mdspan_t Y) {
  Kokkos::parallel_for( Z.extent(0), KOKKOS_FUNCTION [=] (int i) {
    for(int j=0; j<Z.extent(1); j++) {
      Z[i,j] = X [i,j] + Y [i,j];
    }
  });
}
```



On GPUs “adjacent” threads want to do coalesced access – i.e. access elements on the same cache-line.

On CPUs threads rely on prefetching and avoiding of false sharing for performance – i.e. different threads should access different cache lines.

Optimal storage order for an algorithm depends on architecture!

Orthogonalizing Allocation and Access

```
struct SomeAccessor {
  using element_type = ...
  using pointer = ...
  using reference = ...
  using offset_policy = ...


  reference access(pointer ptr, size_t i);
  offset_policy::pointer offset(pointer ptr, size_t offset);
};
```

pointer is not necessarily element_type*

- Could be complex object, for example MPI Window handle …

Reference is not necessarily element_type&

- Could be proxy object with operators such as =,+=,*= defined etc.

access doesn't necessarily return ptr[i];

- The function returns reference *somehow* generated from ptr and i

Typesafe MemorySpace Access

Typesafe memory location

- C++ has no concept for non-accessible memory space
- But custom accessor templated or specific to memory space could do this

```
template<class T>
struct CudaSpaceAccessor {
  using element_type = T;  using pointer = T*;
  using reference    = T&; using offset_policy = CudaSpaceAccessor;


  reference access(pointer ptr, size_t i) {
    #ifdef __CUDA_ARCH__
      return ptr[i];
    #else
      throw std::runtime_error("Accessing CUDA allocation from host");
    #endif
  }
  offset_policy::pointer offset(pointer ptr, size_t offset) { return ptr + offset; }
};
```

Example Impact on Usage

Different overloads for GPU memory with CUDA, vs host memory:

```
using cuda_matrix_t =
  mdspan<const double, dims<2>, layout_left,
         CudaSpaceAccessor<const double>>;

double MyLib::matrix_norm(cuda_matrix_t a);


using host_matrix_t =
  mdspan<const double, dims<2>, layout_left,
         HostSpaceAccessor<const double>>;

double MyLib::matrix_norm(host_matrix_t a);
```

Taking slices with submdspan

Slice and dice as in Fortran

```
auto sub = submdspan(data, slices ...);
```

```
mdspan matrix(ptr, N, M);

// get one row
auto row_i = submdspan(matrix, i, full_extent);

// get multiple columns
auto cols_0_5 = submdspan(matrix, full_extent, pair{0, 5});

// get compile time 4x4 submatrix
// offset type, extent type, stride type
using slice_t = strided_slice<int, std::integral_constant<int, 4>,
                              std::integral_constant<int, 1>>;
auto sub = submdspan(matrix, slice_t{i}, slice_t{j});
```

Slice Arguments

- single item: integral
- Range: pair
- Everything: full_extent
- Range with stride: strided_slice

Status of mdspan availability

C++23: std::mdspan, std::extents, std::layout_[left/right/stride]

- Implemented in LLVM 18, GCC 16, MSVC 2022-17.9

C++26: std::submdspan, std::layout_[left/right]_padded

- Implemented in GCC 16

Limitations in standard implementation: not available for GPU

If you don't want to wait:

- https://github.com/kokkos/mdspan :
  - backport to C++17 and C++20
  - standalone - doesn't require Kokkos itself
  - supports CUDA, HIP and SYCL
  - One difference: data access with ( ) instead of [ ] for C++17/20: a(i, j, k) = 5;

Kokkos Interlude: MDSpan Interop

Kokkos 5 brings interop of mdspan and Kokkos::View with new constructors and conversion functions:

```
explicit(traits::is_managed) View(const mdspan_type &mds);

template<class T, class E, class L, class A>
explicit(/*...*/) View(const mdspan<T, E, L, A> &mds);

template<class T, class E, class L, class A>
constexpr operator mdspan<T, E, L, A>();

template<class A = Kokkos::default_accessor<typename traits::value_type>>
constexpr auto to_mdspan(const A &other_accessor = OtherAccessorType{});
```

Conversion/assignment rules same as between Views or mdspans

Constraints and Concepts

Mechanisms to help with building function overload sets and enforce requirements on function and (class) template parameters.

Constraints:

- Replaces SFINAE mechanism for functions
- Makes it easier to build overload sets
- Can be used on-non-templated class member functions

Concepts:

- Express requirements for a type or multiple types in combination
- Can also replace SFINAE mechanism

Constraint: requires clause

Express constraints and requirements

Most important use: replace SFINAE

Example: function with one overload that takes rank-1 Views and one which takes rank-2 Views

SFINAE

```
template<class T, class E, class L, class A >
std::enable_if_t<mdspan<T, E, L, A>::rank() == 2>
print_elements(mdspan<T, E, L, A> a) { ... }
```

Concepts Requires clause

```
template<class T, class E, class L, class A >
requires(mdspan<T, E, L, A>::rank() == 2)
void print_elements(mdspan<T, E, L, A> a) { ... }
```

https://godbolt.org/z/K5PKfW3aa

Concept

- A “concept” lets you prepackage constraints into a name
- In some situations it can be used almost like a type

Who has written this:

```
template<class T, class E, class L, class A>
void foo(mdspan<T, E, L, A> a) { ... }
```

With Concepts we can do:

```
void foo(mdspan_instance auto a) { ... }
```

How do we get this concept? *Lets start with typetraits for “this is an mdspan”!*

```
template<class T>
constexpr bool is_mdspan_instance_v = false;


template<class T, class E, class L, class A>
constexpr bool is_mdspan_instance_v<mdspan<T,E,L,A> = true;
```

Actual Concepts Continued

The simplest concepts just make typetraits more useable

Without an actual concept we use requires with that typetrait:

```
template<class T>
requires(is_mdspan_instance_v<T>)
void take_mdspan(T) {}
```

Lets define a concept however:

```
template<class T>
concept mdspan_instance = is_mdspan_instance_v<T>;
```

Now we can do this:

```
template<mdspan_instance T>
void take_mdspan(T a) {}
```

And even shorter:

```
void take_mdspan(mdspan_instance auto a) {}
```

https://godbolt.org/z/c7chdeoMn

Actual Concepts with Real Convenience win

What if your function takes two potentially different specializations?

```
template<class T1, class E1, class L1, class A1,
        class T2, class E2, class L2, class A2 >
void take_two(mdspan<T1, E1, L1, A1> a, mdspan<T2, E2, L2, A2> a) { ... }
```

mdspan_instance *is a concept not a concrete class type!*

We can do this

```
template<mdspan_instance T1, mdspan_instance T2>
void take_mdspan(T1 a, T2 b) {}
```

And even without explicit template:

```
void take_mdspan(mdspan_instance auto a, mdspan_instance auto b) {}
```

And this works for templates of classes too!

https://godbolt.org/z/dqoqxG9fG

Concepts: Comparing if constexpr and requires

In C++17 we were able to avoid SFINAE with “if constexpr” expressions: when should we use what?

Example: different code paths in a function **foo** dependent on rank of an mdspan

```
template<mdspan_instance T>
auto foo(T a) {
  static_assert(T::rank() < 3);

  if constexpr (T::rank() == 0) {
    return a[];
  } else if constexpr (T::rank() == 1) {
    return a[0];
  } else {
    return a[0, 0];
  }
}
```

- Enables ranking of choice
- Better control over all variants
- Avoids duplication of common code

```
template<mdspan_instance T>
requires(T::rank() == 0)
auto foo(T a) { return a[]; }

template<mdspan_instance T>
requires(T::rank() == 1)
auto foo(T a) { return a[0]; }

template<mdspan_instance T>
requires(T::rank() == 2)
auto foo(T a) { return a[0, 0]; }
```

- Requires fully disjoint conditions
- Enables overloads in different files
- Reduces spaghetti code

https://godbolt.org/z/v71MK87r1

Concepts: C++20 Defined Concepts

Core language concepts:
- same_as - specifies two types are the same.
- derived_from - specifies that a type is derived from another type.
- convertible_to - specifies that a type is implicitly convertible to another type.
- common_with - specifies that two types share a common type.
- integral - specifies that a type is an integral type.
- default_constructible - specifies that an object of a type can be default-constructed.

Comparison concepts:
- boolean - specifies that a type can be used in Boolean contexts.
- equality_comparable - specifies that operator== is an equivalence relation.

Object concepts:
- movable - specifies that an object of a type can be moved and swapped.
- copyable - specifies that an object of a type can be copied, moved, and swapped.
- semiregular - specifies that an object of a type can be copied, moved, swapped, and default constructed.
- regular - specifies that a type is *regular*, that is, it is both semiregular and equality_comparable.

Callable concepts:
- invocable - specifies that a callable type can be invoked with a given set of argument types.
- predicate - specifies that a callable type is a Boolean predicate.

Concepts: Requires Expression

So far we only checked boolean expressions – but concepts can do more!

Requires Expression can check for the (syntactic) validity of an expression.

- requires expression inside requires clause
- takes variable definitions inside parenthesis, and expression inside curlies

```
requires( requires ( variable declaration )  { expression; } )
```

Example: check that something can be indexed into and assigned to:

```
template<class ViewLike, class T>
requires( requires(ViewLike view, T value, int i, int j) { view(i, j) = value; })
void set_elements(ViewLike v, T a) {
  v(0,0) = a;
}
```

https://godbolt.org/z/TP8Ex4Pbn

More complex concepts

Actual concepts can leverage requires expressions

Example: allow any types that can be added to each other

```
template<class T, class U>
concept addable = requires(T a, U b) { a+=b; };
```

You can combine concepts

```
template<class MT1, class MT2>
concept addable_mdspan = mdspan_instance<MT1> && mdspan_instance<MT2> &&
                addable<typename MT1::element_type, typename MT2::element_type>;
```

Or have just multiple conditions:

```
template<class MT1, class MT2>
concept addable_mdspan =
  is_mdspan_instance_v<MT1> && is_mdspan_instance_v<MT2> &&
  requires(typename MT1::element_type a, typename MT2::element_type b) { a+=b; };
```

https://godbolt.org/z/T5MzrEMcz

Pitfalls of concepts

Often concepts have the effect of introducing public customization points!

- Anything fulfilling the requirements matches!
- Think hard about whether that is the intent!

Modifying an existing concept can change overload sets!

- Both adding and removing requirements is problematic
- Possibly introduce new ambiguity
- Could make user types not match anymore

Generally: concepts that include “this is a specialization of this template” are safer.

C++20 things you may want to look into (or not)

Coroutines: express asynchronicity differently

- Generators, Task systems, event based systems
- Limited support for GPUs
- As with any asynchronous programming concepts: adds complexity

Ranges Library: composable iteration over collection of elements

- Pretty huge capability, needs its own presentation

std::format: write formatted output and define

- Format string + arguments like printf
- customization points defined for printing custom types
- Typesafe and doesn't have overflow problems etc.

Math constants: avoid everyone having defined "Pi" – actually

std::span: super simple 1D mdspan: pointer + (static) extent

Modules: could help with compile times –15% improvement for Kokkos Tests in Kokkos 5

Questions, want to use kokkos/mdspan etc.?

Join the kokkos slack:
https://kokkos.org/community/chat/