MDSPAN
A DEEP DIVE SPANNING
C++, KOKKOS & SYCL

NEVIN “:-)” LIBER
Computer Scientist
nliber@anl.gov
WHO AM I?
WHO AM I?
Nevin “:-)” Liber

- Argonne National Laboratory
  - Computer Scientist
    - Argonne Leadership Computing Facility
- C++, Kokkos, SYCL
- Aurora
- WG21 - ISO C++ Committee
  - Vice Chair, Library Evolution Working Group Incubator (LEWGI / SG18)
  - Admin Chair
- INCITS/C++ - US C++ Committee
  - Vice Chair
- Khronos SYCL Committee Member
WHAT IS SYCL?

SYCL vs Kokkos

- SYCL
  - Performance portability
  - Vendor-provided toolchain
  - Language extensions allowed
  - Implicit data movement
  - Fixed layout* (C++/row major)
  - 3 dimensions*

- Kokkos
  - Performance portability
  - Leverages vendor toolchain
  - Pure C++ library
  - Explicit data movement
  - Parameterized layout
  - 8 dimensions
  - CUDA, HIP, SYCL, HPX, OpenMP, C++ threads backends
WHAT IS MDSPAN?
MDSPAN

- mdspan is a *non-owning* multidimensional array view for C++23

- Vocabulary type
  - Usage in interfaces
  - Usage across domains
MDSPAN

- `mdspan` is a *non-owning* multidimensional array view for C++23

```cpp
template<class ElementType, 
         class Extents, 
         class LayoutPolicy = layout_right, 
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
};
```
MDSPAN

**ElementType**

- `mdspan` is a *non-owning* multidimensional array view for C++23

```cpp
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
}
```
MDSPAN

ElementType

- The elements in the array
MDSPAN

Extents

- `mdspan` is a *non-owning* multidimensional array view for C++23

```cpp
template<class ElementType,
    class Extents,
    class LayoutPolicy = layout_right,
    class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
};
```
MDSPAN

Extents

- Extents describes the dimensions of the multidimensional array
MDSPAN

Extents

- Extents describes the dimensions of the multidimensional array

```cpp
template<
class IndexType, size_t... Es>
class extents;
```

```cpp
template<
class IndexType, size_t Rank>
using dextents = see below;
```
MDSPAN

Extents

- Extents describes the dimensions of the multidimensional array

```cpp
template<class IndexType, size_t... Es>
class extents;
```

```cpp
template<class IndexType, size_t Rank>
using dextents = see below;
```
MDSPAN

Extents

- Extents describes the dimensions of the multidimensional array

```cpp
template<class IndexType, size_t... Es>
class extents;
```
MDSPAN

**IndexType**

- Extents describes the dimensions of the multidimensional array

```cpp
template<class IndexType, size_t... Es>
class extents;
```

- The type used for the index (int, size_t, etc.)
MDSPAN

es...

- Extents describes the dimensions of the multidimensional array

```
template<class IndexType, size_t... Es>
class extents;
```

- Each dimension
  - `std::dynamic_extent` if the dimension is determined at runtime
  - Any other number is the (compile-time) static dimension
- `Es...` are `size_t` because `std::dynamic_extent` is `size_t`
MDSPAN

dextents

- Extents describes the dimensions of the multidimensional array

```cpp
template<class IndexType, size_t... Es>
class extents;
```

```cpp
template<class IndexType, size_t Rank>
using dextents = see below;
```
MDSPAN

**dextents**

```cpp
template<class IndexType, size_t Rank>
using dextents = see below;

dextents<int, 3>
```

- is an alias for

```cpp
extents<int, dynamic_extent, dynamic_extent, dynamic_extent, dynamic_extent>
```

- Corentin Jabot & I really tried to get `std::dynamic_extent` shortened to `std::dyn` in C++20
**MDSPAN**

**LayoutPolicy**

- `mdspan` is a *non-owning* multidimensional array view for C++23

```cpp
template<class ElementType, 
    class Extents, 
    class LayoutPolicy = layout_right, 
    class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
};
```
**MDSPAN**

**LayoutPolicy**

- Maps indices into offsets
  - `layout_right`
    - Rightmost extent is contiguous
  - Default
  - Row-major
  - C++ / C ordering

- `layout_left`
  - Leftmost extent is contiguous
    - \( A[i, j] == A[i + j \times N] \)
  - Column-major
  - Fortran ordering

- `layout_stride`
  - User-defined
    - Tiled, Symmetric, Sparse, Compressed, etc.
MDSPAN

LayoutPolicy::mapping

template<typename Extents>
struct layout_::mapping {
  //...
  constexpr const extents_type& extents() const noexcept;
  constexpr index_type required_span_size() const noexcept;

  template<class... Indices>
  constexpr index_type operator()(Indices...) const noexcept;

  static constexpr bool is_always_unique() noexcept;
  static constexpr bool is_always_exhaustive() noexcept;
  static constexpr bool is_always_strided() noexcept;

  static constexpr bool is_unique() noexcept;
  static constexpr bool is_exhaustive() noexcept;
  static constexpr bool is_strided() noexcept;

  constexpr index_type stride(rank_type) const noexcept;

  template<class OtherExtents>
  friend constexpr bool operator==(const mapping&, const mapping<OtherExtents>&) noexcept;
};
MDSPAN
AccessorPolicy

- mdspan is a *non-owning* multidimensional array view for C++23

```cpp
template<class ElementType, 
    class Extents, 
    class LayoutPolicy = layout_right, 
    class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
};
```
MDSPAN

**AccessorPolicy**

- Customize the pointer and reference types
- Add decorations like restrict
- Remote memory
- Compressed memory
- Atomic access
  - `std::atomic_ref`
MDSPAN

default_accessor

template<class ElementType>
struct default_accessor {
    using offset_policy = default_accessor;
    using element_type = ElementType;
    using reference = ElementType&;
    using data_handle_type = ElementType*;

    constexpr default_accessor() noexcept = default;

    template<class OtherElementType>
    constexpr default_accessor(default_accessor<OtherElementType>) noexcept {}

    constexpr reference access(data_handle_type p, size_t i) const noexcept {
        return p[i];
    }

    constexpr data_handle_type offset(data_handle_type p, size_t i) const noexcept {
        return p + i;
    }
};
MDSPAN

AccessorPolicy

- `mdspan` is a *non-owning* multidimensional array view for C++23

```cpp
template<class ElementType, class Extents, class LayoutPolicy = layout_right, class AccessorPolicy = default_accessor<ElementType>> struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
};
```

- Construct it with a pointer and a list of extents
MDSPAN

AccessorPolicy

- `mdspan` is a *non Owning* multidimensional array view for C++23

```cpp
template<class ElementType,  
    class Extents,  
    class LayoutPolicy = layout_right,  
    class AccessorPolicy = default_accessor<ElementType>> 
struct mdspan {  
    template<class... OtherIndexTypes>  
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts) = delete;  
    // ...  
    template<class... OtherIndexTypes>  
   constexpr reference operator[](OtherIndexTypes... indices) const;  
};
```

- Index it via `m[2, 3, 5]`
**MDSPAN**

- mdspan is a *non-owning* multidimensional array view for C++23

```cpp
template<class ElementType,
         class Extents,
         class LayoutPolicy = layout_right,
         class AccessorPolicy = default_accessor<ElementType>>
struct mdspan {
    template<class... OtherIndexTypes>
    explicit constexpr mdspan(data_handle_type p, OtherIndexTypes... exts);
    // ...
    template<class... OtherIndexTypes>
    constexpr reference operator[](OtherIndexTypes... indices) const;
};
```
HOW DID WE GET HERE?
HOW DID WE GET HERE?

AN EIGHT YEAR MISSION...
N3851 MULTIDIMENSIONAL BOUNDS, INDEX AND ARRAY_VIEW
Łukasz Mendakiewicz & Herb Sutter (Microsoft)

- Based on C++Amp
- Only static extents

```
template <class ValueType, int Rank = 1>
struct array_view {
    constexpr array_view(bounds<Rank> bounds, ValueType* data) noexcept;

    // ...
    constexpr reference operator[](const index<Rank>& idx) const noexcept;
}
```

- strided_array_view
  - Contiguity in the least significant dimension is lifted
ARRAY_VIEW
Issaquah 2014

- Would like variadic operator[] but don't want to wait for language support
- Would like to mix static and dynamic extents
ARRAY_VIEW
Issaquah 2014

- Polls: Strongly Favor | Weakly in Favor | Neutral | Weakly Against | Strongly Against
  - Comfortable with [(1, 2)] syntax? 6 4 2 2 1
  - Comfortable with (1, 2) syntax instead of []? 3 2 3 5 1
  - Comfortable with 2 spellings? 0 1 3 1 10
  - Delay paper in ArraysTS until fixed_array_view? 0 0 4 4 8
  - array_view should have iterators with ValueType array_view? 0 0 3 7 4
  - Add a layout template parameter? 5 6 5 1 0
  - Hold up ArraysTS for layout? Unanimously NO
  - Take array_view for ArraysTS? 9 5 1 0 0
THANKS FOR LISTENING!
THANKS FOR LISTENING!*
THANKS FOR LISTENING!* 

*IF ONLY...
ARRAYS TS
Born 2013 (Chicago)

- Runtime-sized arrays with automatic storage duration
  - Stack arrays
  - No bounds checking, overrun the stack, etc.
  - Need a safe way to access it, iterators, etc.
## CONTIGUOUS TYPES & CONTAINERS

<table>
<thead>
<tr>
<th>Type</th>
<th>Models</th>
<th>Min Capacity</th>
<th>Max Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Exactly 1</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>optional&lt;T&gt;</td>
<td>Up to 1</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>array&lt;T,N&gt;</td>
<td>Exactly N</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>dynarray&lt;T&gt;</td>
<td>Exactly N</td>
<td>Run Time</td>
<td>Run Time</td>
</tr>
<tr>
<td>static_vector&lt;T,N&gt;</td>
<td>Up to N</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>fixed_capacity_vector&lt;T,N&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static_vector&lt;T,N&gt;</td>
<td>Up to N</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>inplace_vector&lt;T,N&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clump&lt;T,N,A&gt;</td>
<td>Indefinite</td>
<td>Compile Time</td>
<td>Run Time</td>
</tr>
<tr>
<td>small_vector&lt;T,N,A&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector&lt;T,A&gt;</td>
<td>Indefinite</td>
<td>Compile Time</td>
<td>Run Time</td>
</tr>
</tbody>
</table>
## CONTIGUOUS TYPES & CONTAINERS

<table>
<thead>
<tr>
<th>Type</th>
<th>Models</th>
<th>Min Capacity</th>
<th>Max Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Exactly 1</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>optional&lt;T&gt;</td>
<td>Up to 1</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>array&lt;T,N&gt;</td>
<td>Exactly N</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>dynarray&lt;T&gt;</td>
<td>Exactly N</td>
<td>Run Time</td>
<td>Run Time</td>
</tr>
<tr>
<td>static_vector&lt;T,N&gt;</td>
<td>Up to N</td>
<td>Compile Time</td>
<td>Compile Time</td>
</tr>
<tr>
<td>fixed_capacity_vector&lt;T,N&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static_vector&lt;T,N&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inplace_vector&lt;T,N&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clump&lt;T,N,A&gt;</td>
<td>Indefinite</td>
<td>Compile Time</td>
<td>Run Time</td>
</tr>
<tr>
<td>small_vector&lt;T,N,A&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vector&lt;T,A&gt;</td>
<td>Indefinite</td>
<td>Compile Time</td>
<td>Run Time</td>
</tr>
</tbody>
</table>
ARRAYS TS

- Runtime-sized arrays need a safe way to access it, iterators, etc.
  - `class dynarray`
    - Allocator not part of the type
      - Passed to constructors
    - How does `dynarray` “stack memory” work if embedded in another type?
      - And what if that aggregate type is on the heap?
      - Compiler writers do not know how to implement this

- **Impasse!**
  - We’ll have an Arrays TS (Technical Specification) to sort it out
ARRAYS TS
2014 (Issaquah)

- Let us add all the array like things
  - array_view
  - array_ref (span-like) and string_ref (string_view)
  - Multi-dimensional support to std::array
  - make_array
  - Extend shared_ptr and make_shared to support arrays
ARRAYS TS
Died 2016 (Jacksonville)

- Should we kill the Arrays TS?
  - 8 5 6 0 0
BACK TO 2014 AND ARRAY_VIEW
ARRAY_VIEW

N3976 (Revision 2) - Rapperswil

- Wording
- Some minor changes
- Send to Library Fundamentals TS v2
  - 10 4 1 2 0
ARRAY_VIEW
N4087 (Revision 3)

- Minor fixes
ARRAY_VIEW
N4177 (Revision 4) - Urbana

- Forward to formal motions?
  - 5 4 2 0 0
N4222 MINIMAL ADDITIONS TO THE ARRAY VIEW LIBRARY FOR PERFORMANCE AND INTEROPERABILITY

Rutger ter Borg & Jesse Perla

- Urbana (after meeting)
  - StorageOrder template tag (layout)
  - fixed_array_view (static extents as template parameters)
  - StrideType template parameter (layout)
  - Variadic operator() for index lookup
- Polls
  - Split 1D array_view from multidimensional one 1 1 4 5 0
  - If variadic operator[] comes along, use it (else operator()) 6 3 2 0 0
  - Allow operator[] for 1D case 4 4 3 0 0
D4300 ISSUES WITH ARRAY_VIEW

H. Carter Edwards (Sandia / Kokkos)

- Array layout has a significant impact on performance, including simd-vectorize
- Array layout performance considerations should include tiling and padding
- Strided should be a layout option
- Ability to mix compile-time and runtime dimensions has performance impact
- Index and bounds could be replaced by `std::array<ptrdiff_t, Rank>`
- `array_view<T, Rank>` inconsistent with `std::array<T, Length>`
- Interaction with memory management is not addressed

- Recommendation: delay N4177 to ArraysTS
ARRAY_VIEW

N4346 (Revision 5) - Cologne (L(E)WG only meeting)

- LWG changes requested in Urbana incorporated
- Many detailed comments
- “Looking in good shape to move in Lenexa”
ARRAY_VIEW

N4494 (Revision 6) - Lenexa

- LWG changes requested in Cologne incorporated
ARRAY_VIEW

N4512 (Revision 7) - Lenexa

- LWG changes requested in Lenexa (earlier in the week) incorporated
- LWG Motion 6 (formal motions straw poll page)
  - Apply N4512 to Library Fundamentals TSv2
  - Plenary discussion
    - Issaquah feedback never incorporated
    - Same issues raised in Rapperswil
    - Pointed out again in N4222 & D4300
      - LEWG supported direction; those authors will come back with proposal
- WG21 (yes no abstain) 16 14 26
- LWG Motion 6 was withdrawn
KOKKOS::VIEW

- Multi-dimensional array of zero or more dimensions

```cpp
template <class DataType [, class LayoutType] [, class MemorySpace] [, class MemoryTraits]>
class View;
```
KOKKOS::VIEW

**DataType**

- Multi-dimensional array of zero or more dimensions

```cpp
template <class DataType
[, class LayoutType]
[, class MemorySpace]
[, class MemoryTraits]>
class View;
```
KOKKOS::VIEW

DataType

- Multi-dimensional array of zero or more dimensions

```cpp
template <class DataType
    [, class LayoutType]
    [, class MemorySpace]
    [, class MemoryTraits]>
class View;
```
KOKKOS::VIEW

**Data Type**

- Runtime (dynamic) and compile time (static) dimensions
- Const views
- Terse notation that must be valid C++ syntax
  - Requires runtime dimensions appear first
  - `View<double**>`
    - 2D View of double with 2 runtime dimensions
  - `View<const int***[5][3]>`
    - 5D View of int with 3 runtime and 2 compile time dimensions.
    - The data is read-only (const).
KOKKOS::VIEW

- Index with operator()
  - 1D Views can be indexed with operator[]
- Sometimes owning
  - (Kokkos parlance: managed or unmanaged)
- Sometimes reference counting
  - Not inside parallel_for, parallel_reduce, parallel_scan
KOKKOS::VIEW

**LayoutType**
- Multi-dimensional array of zero or more dimensions

```cpp
template <class DataType
            [, class LayoutType]
            [, class MemorySpace]
            [, class MemoryTraits]>

class View;
```
- Maps indices into offsets
  - LayoutLeft, LayoutRight, LayoutStride, LayoutTiled
KOKKOS::VIEW

MemorySpace

- Multi-dimensional array of zero or more dimensions

```
template <class DataType
    [, class LayoutType]
    [, class MemorySpace]
    [, class MemoryTraits]>
class View;
```

- Where the memory resides
  - CPU, GPU, etc.
KOKKOS::VIEW

MemoryTraits
- Multi-dimensional array of zero or more dimensions

template <class DataType

    [, class LayoutType]

    [, class MemorySpace]

    [, class MemoryTraits]>

class View;
KOKKOS::VIEW

MemoryTraits

- AccessorPolicy + Managed/Unmanaged
- Atomic
- RandomAccess
  - Hint
- C restrict
KOKKOS::VIEW

Under the covers

- Multi-dimensional array of zero or more dimensions

```
template <class DataType [, class LayoutType] [, class MemorySpace] [, class MemoryTraits]>

class View;
```
KOKKOS::VIEW

Under the covers

- Multi-dimensional array of zero or more dimensions

```cpp
template <class DataType
    [, class LayoutType]
    [, class MemorySpace]
    [, class MemoryTraits]>
class View;
```

```cpp
template <class DataType, class... Properties>
class View;
```
template<
class ArrayType,
   class ArrayLayout = void,
   class SizeType = size_t>
class shared_array;

template<
class ArrayType,
   class ArrayLayout = void,
   class SizeType = size_t>
class weak_array;
N4355: SHARED MULTIDIMENSIONAL ARRAY WITH POLYMORPHIC LAYOUT

**SizeType**

- Being able to customize the SizeType is important for performance!
N4355 : SHARED MULTIDIMENSIONAL ARRAY WITH POLYMORPHIC LAYOUT

**Lenexa**

- ArrayType

\[ T[ N0_{opt} ][ N1_{opt} ][ N2_{opt} ] \]

  - Language change!

- Allow operator[ ] for rank-1?
  - 2 6 1 2 1
Pub Quiz!
- This is the first appearance of
  ```cpp
def class byte : std::uint8_t {};  
```
- Bonus question: what are the differences between this and `byte` in C++17?
  ```cpp
def class byte : unsigned char {};  
```
- Type punning is allowed

- Constructing `array_view` with invalid values
  ```cpp```
  ```cpp
  std::terminate
  ```
  - Not undefined behavior
// class that represents a point in a multidimensional space
template <size_t Rank, typename ValueType = size_t>
class index;

// a random-access iterator over a static_bounds or strided_bounds object
// has the usual form so elided here for brevity of exposition
// comes in both const and non-const flavors
template <typename IndexType>
class bounds_iterator;

// static_bounds is a fixed set of extents
// in multidimensional space for an array_view
// this is one instance of the “bounds” conceptual type
template <typename SizeType, size_t FirstRange, size_t... RestRanges> class static_bounds;

template <size_t Rank, typename SizeType = size_t>
class strided_bounds;
P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

// a helper type that is useful to represent a dimension when creating and navigating strided/multidimensional arrays

```cpp
template <size_t DimSize = dynamic_range>
struct dim;

template <>
struct dim<dynamic_range>;
```

P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

// a helper type that can be passed to the ValueTypeOpt
// parameter of array_view, in which case the size_type
// member is used to determine the type used for measurement
// and index access into the array_view.

```cpp
template <typename ValueType, typename SizeType>
struct array_view_options
{
  struct array_view_traits
  {
    using value_type = ValueType;
    using size_type = SizeType;
  };
};
```
P0122R0 ARRAY_VIEW: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

Neil MacIntosh (Microsoft)

// a random-access iterator over an array_view or strided_array_view object
// has the usual form so elided here for brevity of exposition
// comes in both const and non-const flavors

```
template <typename IndexType>
class array_view_iterator;

template <typename ValueTypeOpt, size_t FirstDimension, size_t... RestDimensions>
class array_view;
```
My thoughts

- First time I can remember being present for an array_view discussion
- Way too complicated for mere mortals like me
- I’ll let the people who need this figure it out

- If only I knew I would be entering this field exactly three years after the date on this paper!…
P0122R1 SPAN: BOUNDS-SAFE VIEWS FOR SEQUENCES OF OBJECTS

- Removed multidimensional aspects from the proposal

- Unfortunately, I stopped paying attention to this proposal for a while…
I'm sorry

using size_type = size_t;
SPAN

using size_type = size_t;

- I’m sorry
  - I lead the charge
    - Up against well-known C++ luminaries
  - Before I was in HPC
    - And thought the performance differences were minor
      - *Which is what committee members say when they want a feature*
      - Don’t want a feature? *Can’t afford even one cycle*
  - More important was Interoperability with the rest of the standard library
Not just Kokkos folks

The essential issue with array_view
  - Did not fulfill C++’s zero-overhead abstraction
    - For both static and dynamic extents
    - Different memory layouts
      - Eigen
      - Matlab’s C++ interface
  - Would need another library for “direct mapping to the hardware”
P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

- Layout more general
  - Different orderings
  - Padding
- Interoperability with libraries using compile-time extents
- Zero-overhead abstraction for constexpr extents and strides
- Extensibility for view properties beyond dimensions and layouts
P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

Multiple implicit dimensions

template<class DataType, class... Properties>

struct view;

- View of multidimensional array with multiple implicit dimensions
  - Either pass a property, or…
  - “Requires slight language specification change for correction and relaxation of array declaration.”

view<int, view_property::implicit_dimensions<3>>;
view<int[][][][]>>;
Multiple implicit dimensions

view<int, view_property::implicit_dimensions<3>>;
view<int[][][][]>>;

- Equivalent-but-distinct types
- Issues when declaring the type in an interface

```cpp
void DoSomething(view<????> v);
```

- Separate overloads
- Pay (small) runtime conversion cost
- Stay in template-land
P0009R0 POLYMORPHIC MULTIDIMENSIONAL ARRAY VIEW

- **Layout**
  - `view<int[][][], view_property::layout_left>`

- **(Variadic) Properties** get you flexibility and extensibility
  - At the cost of many *equivalent-but-distinct* types

- **Polls**
  - Do we want static zero-length extents? 3 4 2 3 0
  - Do we want property lists in the template arguments? 3 6 3 0 0
  - Do we want per view bounds checking? 3 4 2 1 1
Bikeshed
  - array_ref

What about errors?
  - Contracts?
CONTRACTS

- C++ has only one knob that says “Here Be Dragons”
  - *Undefined Behavior*
  - Everything else is *defined* behavior
    - And developers *will* write code dependent on defined behavior
- Contracts will give us more knobs
  - C++26 hopefully?
view is now array_ref

Debate on signed vs. unsigned size_type
P0009R2 POLYMORPHIC MULTIDIMENSIONAL ARRAY REFERENCE
pre-Oulu 2016

- Add details for layout mapping
- Relaxed array declaration syntax moved to P0332
- Motivation and examples moved to P0331
Undesirable Extent Mechanism (B) Proposal

```cpp
template<
    size_t... IntegralExtent
>
struct extents;
```
- Renamed to `mdspan`
- Align with `span`
- Extents now part of this proposal
  - Still hoping for `mdspan<int>[][][][]`?

- Polls
  - We should be able to index with `span<int-type[N]>(in addition to an array)? 2 11 1 1 0`
  - We should be able to index with 1d `mdspan? 0 8 7 0 0`
  - Forward this to LWG for Library Fundamentals v3? Unanimous consent
LIBRARY FUNDAMENTALS V3
First working draft post-Rapperswil 2018

- Never got mdspan
- As for it shipping…
  - P2631R0 Publish TS Library Fundamentals 3 Now! - Alisdair Meredith, Bryce Adelstein Lelbach, Jonathan Wakely
    - Discussed on September 27th, 2022
      - Publish LFTSv3 - 3 5 3 4 2
      - Never publish LFTSv3 - 2 5 4 7 0
      - Don’t publish, but evaluate contents for IS (C++26 or later) - 4 10 2 2 1
      - Close LFTSv3 and stop maintaining working draft - 5 4 4 2 0
    - To be published later in 2023 (will be N4952; draft is N4948)
LIBRARY FUNDAMENTALS V3

P2708 No Further Fundamentals TSes

- Publish LFTSv3
  - No more LFTS after that
    - “The C++ committee intends to release new versions of this technical specification periodically, containing the library extensions we hope to add to a near-future version of the C++ Standard.”
  - Send P2708R0 (No Further Fundamentals TSes) to Library for Library Fundamentals TS v3 without a Library Evolution electronic poll: 8 7 1 0 0

- In the process of being published
  - Current (last?) draft is N4948; to be published as N4952
P0009R5 POLYMORPHIC MULTIDIMENSIONAL ARRAY
REFERENCE

Jacksonville 2018

- P0009R4 changes except `span<int-type [N]>` (weak support 2 1 1 0 0 0 & no proven need)
- P0009R5 not reviewed in Jacksonville
We want the customization of `basic_mdspan` to be two customization points `Mapper` and `Accessor` (akin to `Allocator` design)?

```cpp
basic_mdspan<T, Extents, Mapper, Accessor>
mdspan<T, N...>
```

- We don’t want too many types in the template argument list

We want the customization of `basic_mdspan` to be an arbitrary (and potentially user-extensible) list of properties (akin to `Executor` property design)?

```cpp
basic_mdspan<T, Extents, Properties...>
```

- I don’t want too many types in the template argument list
Replaced variadic property list with extents, layout mapping and accessor properties
- Added accessor policy concept
- Renamed `mdspan` to `basic_mdspan`

// Multidimensional span:
```cpp
template<typename ElementType, typename Extents, 
          typename LayoutPolicy = layout_right, 
          typename AccessorPolicy = accessor_basic>
class basic_mdspan;
```

```cpp
template <class T, ptrdiff_t... Extents>
using mdspan = basic_mdspan<T, extents<Extents...>>;
```
P0009R7 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE

post-Rapperswil 2018

- Wording
- How to refer to span (as that will be in C++20, not C++17)
P0009R8 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE
San Diego 2018

- Update based on reference implementation
NEVIN JOINS ARGONNE NATIONAL LABORATORY

- February 11th, 2019
P0009R9 MDSPAN: A NON-OWNING MULTIDIMENSIONAL ARRAY REFERENCE

Kona 2019

- Week of February 18th, 2019
  - Wording
CURRENT

array[x] // Ok
array[(x,y)] // Ok, uses y as index/key
array[x,y] // Ok, uses y as index/key

PROPOSED

array[x] // Ok
array[(x,y)] // Ok, uses y as index/key
array[x,y] // Deprecated,
            // uses y as index/key
Wording and paper cleanup

C++20 is done!
  - span is hot
  - mdspan is not

I accepted the LEWGI vice chair position

And then the pandemic hit…
I’m now an author on P0009!

- Changed all sizes from ptrdiff_t to size_t
- Explicit about trivially copyable
TRIVIALLY COPYABLE

- How do we copy objects in C++?
  - Copy constructor / copy assignment operator
    - Running code
    - Code may access both source and destination
  - Can we do the same for inter-device copying (host/device or device/device)?
    - Where would the code run?
    - May not be able to simultaneously access source and destination
- We can copy the bytes (object representation) that make up the object
- C++ **trivially copyable** used as a proxy for types where we can copy the bytes
Now L(E)WG wants the design decisions back in the paper *sigh*

Poll

- Prefer the IS over LFTSv3 as ship vehicle for P0009 (mdspan)
  - 10 6 1 0 0

Still hopeful for `mdspan<T[[]][]> mdspan<T[][64][]>` syntax
Because P1161 deprecated the use of comma expressions in subscript expressions in C++20

- Now make them ill-formed and give a new meaning to commas in subscript expressions
- \( a[x, y, z] \)
  - Eliminate workarounds
    - \( a(x, y, z) \)
    - \( a[x][y][z] \)
    - \( a\{x, y, z\} \)
- “We propose that \( \text{operator}[] \) should be able to accept zero or more arguments, including variadic arguments.”
P2589 STATIC OPERATOR[]

Kona 2022

- P2128
  - “Both its use and definition would match that of operator().”
    - Except P1169 static operator() changes fell through the cracks
- P2589 static operator[]
  - National body comments
    - US27-067, CA-065, GB-066
  - Forward P2589R0 to CWG for inclusion in C++23
    - 5 12 6 1 0
- CWG
  - “It would be a shame for Nevin to miss out on a chance to come to core.”
dextents type alias

Removed old `mdspan` and renamed `basic_mdspan` to `mdspan`

```
template<class ElementType, class Extents, class LayoutPolicy, class AccessorPolicy>
class basic_mdspan { /* ... */ };   

template<class T, size_t... Extents>
using mdspan = basic_mdspan<T, extents<Extents...>>;
```

- Deduction guides (CTAD) [P2299]
- `operator[]`
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD) (AKA DEDUCTION GUIDES)
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

Deduction Guides (C++17)

- Class template parameters are deduced from the constructor arguments

\[\text{template}<\text{class } T1, \text{class } T2>\]
\[\text{pair}(T1, T2) \rightarrow \text{pair}<T1, T2>;;\]

- All parameters must be deduced
- Implicit and user defined ones
- Creates a different overload set
  - Exactly one match \(\text{pair}(\ldots)\) -> exactly one match \(\text{pair}<T1, T2>::\text{pair}(\ldots)\)

\[\text{pair}<\text{int, const char*}> t(2, "Three"); // \text{All pair ctors}\]
\[\text{pair } d(2, "Three"); // \text{Only CTAD ctors}\]
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

- No need to specify template parameters when declaring non-member variables
  - *Immense gain in usability* for types with lots of template parameters
    - Like `mdspan`
  - But it is a tradeoff
    - Still need to know the exact type with template parameters
      - Declaring member variables
      - Compile time debugging
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

**mdspan (from C++23)**

```cpp
template<class CArray>
requires(is_array_v<CArray> && rank_v<CArray> == 1)
mdspan(CArray&)
-> mdspan<remove_all_extents_t<CArray>, extents<size_t, extent_v<CArray, 0>>;  

template<class Pointer>
requires(is_pointer_v<remove_reference_t<Pointer>>)
mdspan(Pointer&)
-> mdspan<remove_pointer_t<remove_reference_t<Pointer>>, extents<size_t>>;  

template<class ElementType, class... Integrals>
requires((is_convertible_v<Integrals, size_t> && ...) && sizeof...(Integrals) > 0)
explicit mdspan(ElementType*, Integrals...)  
-> mdspan<ElementType, dextents<size_t, sizeof...(Integrals)>>;  

template<class ElementType, class OtherIndexType, size_t N>
mdspan(ElementType*, span<OtherIndexType, N>)  
-> mdspan<ElementType, dextents<size_t, N>>;  

template<class ElementType, class OtherIndexType, size_t N>
mdspan(ElementType*, const array<OtherIndexType, N>&)  
-> mdspan<ElementType, dextents<size_t, N>>;  

template<class ElementType, class IndexType, size_t... ExtentsPack>
mdspan(ElementType*, const extents<IndexType, ExtentsPack...>&)  
-> mdspan<ElementType, extents<IndexType, ExtentsPack...>>;  

template<class ElementType, class MappingType>
mdspan(ElementType*, const MappingType&)  
-> mdspan<ElementType, typename MappingType::extents_type,  
typename MappingType::layout_type>;  

template<class MappingType, class AccessorType>
mdspan(const typename AccessorType::data_handle_type&, const MappingType&,  
const AccessorType&)  
-> mdspan<typename AccessorType::element_type, typename MappingType::extents_type,  
typename MappingType::layout_type, AccessorType>;  

mdspan(from C++23)
```
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

mdspan(T*)  // pointer to one object

```cpp
template<class Pointer>
    requires(is_pointer_v<remove_reference_t<Pointer>>)
    mdspan(Pointer&&)
    -> mdspan<remove_pointer_t<remove_reference_t<Pointer>>>, extents<size_t>>;

// …

int i = 0;
int* p = &i;
mdspan d(p);
```

// mdspan(int**)&
// remove_reference_t<int&&> --> int*
// is_pointer_v<int*> == true
// remove_pointer_t<int*> --> int
// mdspan<int, extents<size_t>> -->
// mdspan<int, extents<size_t>, layout_right, default_accessor<int>>
// template<class... OtherIndexTypes>
// constexpr explicit mdspan<int, extents<size_t>>::mdspan(int*, OtherIndexTypes... exts);
// template<> constexpr explicit
```
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

mdspan (from C++23)

```cpp
template<class CArray>
requires(is_array_v<CArray> && rank_v<CArray> == 1)
mdspan(CArray&)
   -> mdspan<remove_all_extents_t<CArray>, extents<size_t, extent_v<CArray, 0>>>;

template<class Pointer>
requires(is_pointer_v<remove_reference_t<Pointer>>)
mdspan(Pointer&)
   -> mdspan<remove_pointer_t<remove_reference_t<Pointer>>, extents<size_t>>>;

template<class ElementType, class... Integrals>
requires((is_convertible_v<Integrals, size_t> && ...) && sizeof...(Integrals) > 0)
explicit mdspan(ElementType*, Integrals...)
   -> mdspan<ElementType, dextents<size_t, sizeof...(Integrals)>>;

template<class ElementType, class OtherIndexType, size_t N>
mdspan(ElementType*, span<OtherIndexType, N>)
   -> mdspan<ElementType, extents<size_t, N>>>;

template<class ElementType, class OtherIndexType, size_t N>
mdspan(ElementType*, const array<OtherIndexType, N>&)
   -> mdspan<ElementType, extents<size_t, N>>>;

template<class ElementType, class IndexType, size_t... ExtentsPack>
mdspan(ElementType*, const extents<IndexType, ExtentsPack...>&)
   -> mdspan<ElementType, extents<IndexType, ExtentsPack...>>>;

template<class ElementType, class MappingType>
mdspan(ElementType*, const MappingType&)
   -> mdspan<ElementType, typename MappingType::extents_type,
              typename MappingType::layout_type>;

template<class MappingType, class AccessorType>
mdspan(const typename AccessorType::data_handle_type&, const MappingType&,
        const AccessorType&)
   -> mdspan<typename AccessorType::element_type, typename MappingType::extents_type,
             typename MappingType::layout_type, AccessorType>;
```

mdspan (from C++23)
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

mdspan (from C++23)

```cpp
template<class CArray>
requires(is_array_v<CArray> && rank_v<CArray> == 1)
mdspan(CArray&) -> mdspan<remove_all_extents_t<CArray>, extents<size_t, extent_v<CArray, 0>>;

template<class Pointer>
requires(is_pointer_v<remove_reference_t<Pointer>>)
mdspan(Pointer&) -> mdspan<remove_pointer_t<remove_reference_t<Pointer>>, extents<size_t>>;

template<class ElementType, class... Integrals>
requires((is_convertible_v<Integrals, size_t> && ...) && sizeof...(Integrals) > 0)
explicit mdspan(ElementType*, Integrals...) -> mdspan<ElementType, dextents<size_t, sizeof...(Integrals)>>;

template<class ElementType, class OtherIndexType, size_t N>
mdspan(ElementType*, span<OtherIndexType, N>) -> mdspan<ElementType, extents<size_t, N>>;

template<class ElementType, class OtherIndexType, size_t N>
mdspan(ElementType*, const array<OtherIndexType, N>&) -> mdspan<ElementType, extents<size_t, N>>;

template<class ElementType, class IndexType, size_t... ExtentsPack>
mdspan(ElementType*, const extents<IndexType, ExtentsPack...>&) -> mdspan<ElementType, extents<IndexType, ExtentsPack...>>;

template<class ElementType, class MappingType>
mdspan(ElementType*, const MappingType&) -> mdspan<ElementType, typename MappingType::extent_type, typename MappingType::layout_type>;

template<class MappingType, class AccessorType>
mdspan(const typename AccessorType::data_handle_type&, const MappingType&, const AccessorType&) -> mdspan<typename AccessorType::element_type, typename MappingType::extent_type, typename MappingType::layout_type, AccessorType>;
```
CLASS TEMPLATE ARGUMENT DEDUCTION (CTAD)

**mdspan(T*)**  // pointer to one object

template<class Pointer>
    requires(is_pointer_v<remove_reference_t<Pointer>>)
    mdspan(Pointer&&)
    -> mdspan<remove_pointer_t<remove_reference_t<Pointer>>, extents<size_t>>;

// ...

int i = 0;
int* p = &i;
mdspan d(p);

// mdspan(int&&)
// remove_reference_t<int&&> --> int*
// is_pointer_v<int*> == true
// remove_pointer_t<int*> --> int
// mdspan<int, extents<size_t>> ->
// mdspan<int, extents<size_t>>, layout_right, default_accessor<int>>
// template<class... OtherIndexTypes>
// constexpr explicit mdspan<int, extents<size_t>>::mdspan(int*, OtherIndexTypes... exts);
// template<> constexpr explicit

121
P0009R14 MDSPAN

2021 telecons, telecons, telecons…

- Send P0009R14 (mdspan) to LWG for C++23 with priority P3 (to be confirmed with a Library Evolution electronic poll)
  - 9 7 0 0 0
  - 11/2021
P0009R15 MDSPAN

2022 telecons, telecons, telecons…

- LWG wording review
P0009R16 MDSPAN

2022 telecons, telecons, telecons…

- LWG wording review
P0009R17 MDSPAN
2022 telecons, telecons, telecons…

- submdspan moved to a separate paper
  - Not enough time to review it before C++23 feature freeze
  - I didn’t have enough time to review it in this talk either. :-(
P2553R2 MAKE MDSPAN SIZE_TYPE CONTROLLABLE

2022 telecons, telecons, telecons...

- Added SizeType template parameter for extents

```cpp
template<class SizeType, size_t... Es>
class extents;
```

- Initially constrained to unsigned types
- LEWG relaxed that constraint
  - Do not constrain extents size_type to unsigned_integral, allow for signed extents
  7-8-1-0-0
  - The concept of the size_type should be a Mandate rather than a Constraint 7-10-0-0-0
  - Send [P2553R1] Make mdspan size_type Controllable to Library Working Group for C+
    +23, classified as an improvement of an existing feature ([P0592R4] bucket 2 item)
  7-9-1-1-0
Throughout the standard, size_type stands for an unsigned type

Rename size_type to index_type

What should mdspan::size() return?

- P0009R16 returned old size_type
- P0009R17 returned size_t

```cpp
template<...> class extents { // ...
    using size_type = make_unsigned_t<index_type>;
};
```

```cpp
template<...> class mdspan { // ...
    using size_type = typename extents::size_type;
    constexpr size_type size() const noexcept;
};
```

Separate paper from P0009 to lessen risk of P0009 not making C++23
2022 telecons, telecons, telecons...

- Send P2599R0 (mdspan::size_type should be index_type) to Library for C++23 classified as an improvement (B2), to be confirmed with a Library Evolution electronic poll 2-7-2-2-0
  - SA: It's already a conscious choice by the user to use a signed type. So I don't think it will be surprising. The consistency of having it be called size_type is more important
- mdspan, extents, and layouts should have both an index_type (which is whatever the user provides for the first template parameter to extents) and a size_type (which is make_unsigned_t<index_type>) 3-9-1-1-0
  - WA: It's additional complexity
- Modify P2599R1 (mdspan::size_type should be index_type) such that mdspan::sizes return type is size_type, and send the modified paper to Library for C++23 classified as B2 - Improvement, to be confirmed with a Library Evolution electronic poll 5-8-0-1-0
  - WA: This is a late change.
- Put P2599R2 into C++23 pending LEWG approval 18-0-0
- Send [P2599R2] index_type & size_type In mdspan to Library Working Group for C++23, classified as an improvement of an existing feature ([P0592R4] bucket 2 item) 14-7-2-1-0
LWG review of P0009 wanted naming changes for problematic names

- **pointer** -> **data_handle_type**
  - Really is an opaque handle to data
  - Need not be dereferencable or indexable
  - Follows precedence of `std::thread::native_handle_type`

- Similar reasoning for `mdspan::data()` -> `mdspan::data_handle()`

- **contiguous** -> **exhaustive**
  - contiguous implies linear order, which isn’t necessarily true

Separate paper from P0009 to lessen risk of P0009 not making C++23

- 13-13-0-0-0
P2613R1 ADD THE MISSING `EMPTY` TO `MDSPAN`

2022 telecons, telecons, telecons…

- Add empty() to go along with size()
- Separate paper from P0009 to lessen risk of P0009 not making C++23
  - Almost didn’t make it, as P2613R0 had a wording bug
  - 10-10-1-1-0

[[nodiscard]] constexpr bool empty() const noexcept;

P0009R18 MDSPAN


- Apply the changes in P0009R18 (MDSPAN) to the C++ working paper
  - *Unanimous consent*
  - 2022-Jul-25 11:25 am CDT
BUT THAT WAS THE C++20 PRAGUE CELEBRATION
BUT THAT WAS THE C++20 PRAGUE CELEBRATION

HERE IS THE P0009 MDSPAN CELEBRATION:
Nevin 11:25 AM
Whoo hoo!

dalg24 11:26 AM
Hey!
crtrott 11:27 AM
it made it 😊

Mark Hoemann 1:21 PM
Congrats y’all!!!!!!
cartesian_product too! and the extended floating point paper!

Nevin 2:04 PM
Latest messages

Message: B isocpp
MDSPAN

http://eel.is/c++draft/views#mdspan.syn

- As of 2022-August-17:

```cpp
namespace std {
    // [mdspan.extents], class template extents
template<class Extents, size_t... Extents>
    class extents;

    // [mdspan.extents.dextents], alias template dextents
template<class IndexType, size_t Rank>
    using dextents = see below;

    // [mdspan.layout], layout mapping
    struct layout_left;
    struct layout_right;
    struct layout_stride;

    // [mdspan.accessor.default], class template default_accessor
template<class ElementType>
    class default_accessor;

    // [mdspan.mds], class template mds
    template<class ElementType, class Extents, class LayoutPolicy = layout_right,
             class AccessorPolicy = default_accessor<ElementType>>
    class mds;
}
```
Reference mdspan implementation

The ISO-C++ proposal [P0009](https://github.com/isl/teamiso) will add support for non-owning multi-dimensional array references to the C++ standard library. This repository aims to provide a production-quality implementation of the proposal as written (with a few caveats, see below) in preparation for the addition of `mdspan` to the standard. Please feel free to use this, file bugs when it breaks, and let us know how it works for you :)  

**Try it out on Godbolt**! [link]

During the final leg of the ISO C++ committee review process a number of breaking changes were proposed and accepted (issue #136). These are now merged into the stable branch.

Note: There is a tag `mdspan-0.3.0` which reflects the status of P0009 before R17 - i.e. it does not have the integral type template parameter for `extents`. Note: There is a tag `mdspan-0.4.0` which reflects the status of P0009 before

- renaming `pointer`, `data`, `is_contiguous` and `is_strides_contiguous`; and before
- renaming `size_type` to `index_type` and introducing a new `size_type = make_unsigned_t<index_type>` alias.

**Using mdspan**

A tutorial-style introduction to the basic usage of `mdspan` is provided on the project wiki. More advanced tutorials to come.

**Features in Addition To C++ Standard**

- C++17 backport (e.g. concepts not required)
- C++14 backport (e.g. fold expressions not required)
- Compile times of this backport will be substantially slower than the C++17 version
- Macros to enable, e.g. `__device__` marking of all functions for CUDA compatibility

**Building and Installation**

This implementation is header-only, with compiler features detected using feature test macros, so you can just use it directly with no building or installation. If you would like to run the included tests or benchmarks, you’ll need CMake.
Adding `mdspan` reference implementation #299

Youyu3 commented 13 days ago

- Pulls the `mdspan` reference implementation from branch "stable" of the kokkes repo, https://github.com/kokkes/mdspan, up to PR 172.
- Unified internal identifiers and made some naming convention updates.

Youyu3 added 7 commits 14 days ago

- Pulls the `mdspan` reference implementation from branch "stable" from t...948
- Uglification and naming convention updates
- More naming convention updates
- More uglification and naming convention updates
- More uglification of function parameter names
- Fix some uglifications
- Move `mdspan` as the directory tree

Mhoommen reviewed 6 days ago

```
#include/cuda/std/detail/ittbx/include/experimental/_block_bits/config.hpp
```

229  #endif
230  
277  #if(__hasmdspan_use_alias_template_argument_deduction)
278  
279  #endif

```
View changes
```
SYCL & MDSPAN
SYCL & MDSPAN

A small subgroup of the Khronos SYCL Committee started thinking about and fleshing out how we can take advantage of mdspan in SYCL about a year ago

- First thoughts
  - Accessors
    - Unified Shared Memory (USM)
  - Strides, offsets and sub-buffers
  - More than three dimensions
  - Unify buffers and images

Tom Deakin, Dániel Berényi, Nevin Liber
Ronan Keryell, Roland Schulz, Thomas Applencourt, James Brodman, Aksel Alpay, Gregory Lueck, Gordon Brown, Tadej Ciglarič
SYCL ACCESSORS

```cpp
template <typename DataT, int Dimensions, access_mode AccessMode, target AccessTarget, ...>
class accessor { /* ... */ };
```

- accessor is the non-owning view of `sycl::buffer`
  - `access_mode`: read, write, read_write
  - `target`: device, host_task

- mdspan improvements over accessors
  - `LayoutPolicy`: flexibility for order data is stored
  - `AccessPolicy`: restrict, atomic, volatile, etc.
SYCL ACCESSORS

template <typename DataT, int Dimensions, access_mode AccessMode, target AccessTarget, ...>
class accessor { /* ... */ };  

- mdspan improvements over accessors (continued)
  - Rectangular copies
    - USM copies from host
    - Async copy to/from all memory space combinations

- Alternative
  - Add these features to SYCL accessor
SYCL & MDSPAN

- `embedded_ptr` (hipSYCL)
  - Lightweight (compared with accessor) to get pointers to data inside kernels
  - In general we can’t use raw pointers
    - Can’t always share between host and device
  - We can use the `embedded_ptr` to directly create construct an `mdspan`

```cpp
sycl::buffer<double, 2> A = {N, P};
embedded_ptr p_A {A, sycl::read_only};
cgh.parallel_for(...) {
    std::mdspan md_A {p_A, {N, P}};
}
```
SYCL & MDSPAN

Current Status

- Grew the subgroup to include implementers
- Working on proposal for next Khronos F2F Meeting (eight days from today)
  - Buffer accessor `mdspan`
  - `embedded_ptr mdspan`
  - USM `mdspan`
  - C++23 baseline?
    - `mdspan::operator[]` requires it

- Stay tuned!
KOKKOS

- Refactoring View to use `mdspan`
- Papers targeting C++26
  - P1673 A free function linear algebra interface based on the BLAS
  - P1684 `mdarray`
  - P2630 `submdspan`
  - P2642 Padded `mdspan` layouts
  - P2689 Atomic Refs Bound to Memory Orderings & Atomic Accessors
  - P2763 `layout_stride` static extents default constructor fix (C++23)
  - P2798 Fix layout mappings all static extent default constructor (C++23)
  - P2897 `aligned_accessor`: An `mdspan` accessor expressing pointer overalignment
P1684R0 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

**mdarray**

```cpp
template<class ElementType, class Extents, class LayoutPolicy = layout_right, class ContainerPolicy = see-below>
class mdarray;
```

- Adaptor
  - `stack, queue, priority_queue, flat_set, flat_map`
template<
class ElementType,
class Extents,
class LayoutPolicy = layout_right,
class ContainerPolicy = see-below>
class mdarray;

▪ “Replaces” AccessorPolicy from mdspan
  ▪ Generalization of needed contiguous container features
  ▪ create() method
P1684R0 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

Cologne 2019

template<class ElementType,
class Extents,
class LayoutPolicy = layout_right,
class ContainerPolicy = see-below>
class mdarray;

- Polls
  - Do this as containers (md_array, md_vector?) instead of as adaptor? 0 7 2 2 3
  - Continue work and come back (we believe this is a problem the standard should solve? 8 8 3 2 0
P1684R0 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

Container Adaptor

template<class ElementType,
        class Extents,
        class LayoutPolicy = layout_right,
        class ContainerPolicy = see-below>

    class mdarray;

- Why an adaptor?
  - array<T,N>, vector<T,A>, inplace_vector<T,N>, small_vector<T,N,A>
  - Device specific containers (sometimes C++ Standard Library containers won’t work)
  - User defined containers
template<
class ElementType,
class Extents,
class LayoutPolicy = layout_right,
class ContainerPolicy = see-below>
class mdarray;

- Uses Container, not ContainerPolicy
  - Defaults to array when all static extents
  - Otherwise, defaults to vector
- Poll: The default container should be std::vector? 3 4 3 0 0
template<class ElementType,
class Extents,
class LayoutPolicy = layout_right,
class ContainerPolicy = vector<ElementType>>
class mdarray;

- Poll:
  - We support the presented container adapter design 8 8 0 1 0
P1684R3 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

```cpp
template<class ElementType, 
    class Extents,  
    class LayoutPolicy = layout_right,  
    class ContainerPolicy = vector<ElementType>> 
    class mdarray;
```

- Consistent with C++23 `mdspan`
- Added size constructible container requirements
P1684R4 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

- Drop the “size constructible container” requirements and simply use preconditions on relevant constructors
P1684R5 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

- Possibly simplify constructors
- Disambiguate deduction guides
- Add precondition to relevant functions about container size being large enough
  - Fixes issues in moved-from state (maybe?)
P1684 AN OWNING MULTIDIMENSIONAL ARRAY ANALOG OF MDSPAN

Strong Invariants

- `mdarray` with all static extents is not default constructible
  - Move assignment modifies both the source and destination underlying containers
  - What is the moved-from state of an `mdarray` of all static extents?
    - Valid-but-unspecified state of the underlying container isn’t sufficient
  - What is the state of both the source and destination if move assignment throws?
  - Common standard containers:
    - `array` - no problem (array elements in moved-from state won’t break `mdarray` invariants)
    - `vector::clear()` isn’t sufficient to maintain the invariant
    - User defined containers?
  - Other adaptors with invariants (such as `priority_queue`) don’t answer these questions either
    - NB comment rejected for `flat_map` and `flat_multimap`
    - Just to be clear, `clear()` can solve this for `flat_map` and `flat_multimap`
P2630 SUBMDSPAN

template<class T, class E, class L, class A, class... SliceArgs)
auto submdspan(mdspan<T,E,L,A> x, SliceArgs... args);

- Returns a different mdspan type
- Originally part of P0009
- Added customization points so that submdspan can work with user-defined policies
- Added the ability to specify slices as compile time values
P2630 SUBMDSPAN

SliceArgs

- **index_type** (from input mdspan)
  - Rank of resulting mdspan is one less than the input mdspan
  - Contains only elements where the index matches this slice specifier

- **tuple<index_type, index_type>**
  - Begin to end subrange of elements

- **full_extent_t**
  - Full range of indices

- **strided_index_range<OffsetType, ExtentType, StrideType>{{.offset, .extent, .stride}}**

- If any of **index_type, OffsetType, ExtentType, StrideType** is integral_constant
  - Compile time constant baked into the mdspan return type
strided_index_range{.offset, .extent, .stride}

- offset
  - The start index
- extent
  - Length of the subrange (not the end index)
- stride
  - Stride within that subrange
P2630 SUBMDSPAN

Customization Points

template<class Mapping, class ... SliceArgs>
auto submdspan_mapping(const Mapping&, SliceArgs...) { /* ... */ }

template<class Mapping, class ... SliceArgs>
size_t submdspan_offset(const Mapping&, SliceArgs...) { /* ... */ }

template<class T, class E, class L, class A, 
class ... SliceArgs)
auto submdspan(const mdspan<T,E,L,A>& src, SliceArgs ... args) {
  size_t sub_offset = submdspan_offset(src.mapping(), args...); // ADL
  auto sub_map = submdspan_mapping(src.mapping(), args...); // ADL
  typename A::offset_policy sub_acc(src.accessor());
  typename A::offset_policy::data_handle_type
    sub_handle = src.accessor().offset(src.data_handle(), sub_offset);
  return mdspan(sub_handle, sub_map, sub_acc); // Customizations
}
P2630 SUBMDSPAN

Varna 2023

- 2023-05 Library Evolution Electronic Poll (before Varna)
  - Send P2630R3 submdspan to LWG for C++26? 10 4 1 0 1

- Varna 2023 LWG
  - Put P2630R4 into C++26? 6 2 0
P2642 PADDED MDSPAN LAYOUTS
P2642 PADDED MDSPAN LAYOUTS
Mark Hoemmen, Christian Trott, Damien Lebrun-Grandie, Malte Förster, Jiaming Yuan

- layout_left_padded & layout_right_padded
  - Array layouts that are contiguous in one dimension
    - Commonly supported in BLAS and LAPACK
  - Padded storage for over aligned access of the start of every contiguous segment of the array

- The proposed tagged type (layout_left_padded) should be a nested type (layout_left::padded) 0 0 1 6 1
- An assume_layout customisation point should be provided for layout conversions. 0 0 2 5 0
P2689 ATOMIC REFS BOUND TO MEMORY ORDERINGS & ATOMIC ACCESSORS
SG1 (Issaquah 2023)
- We like the bound memory_order versions of atomic_ref 1 1 1 0 0
- SX X N Y SY 1 1 3 4 2
  - X - The meaning of the bound memory_order is that it is the default order
  - Y - The meaning of the bound memory_order is that it is the only order

LEWG (Issaquah 2023)
- Do we want a more generic (non-exposition only) atomic_ref_\* with an additional memory_order template parameter which these are template aliases for?
### ATOMIC REFS BOUND TO MEMORY ORDERINGS & ATOMIC ACCESSORS

Christian Trott, Damien Lebrun-Grandie, Mark Hoemmen, Daniel Sunderland, Nevin Liber

<table>
<thead>
<tr>
<th>Atomic Ref</th>
<th>memory_order</th>
<th>Loads</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_ref_relaxed</td>
<td>memory_order_relaxed</td>
<td>memory_order_relaxed</td>
<td>memory_order_relaxed</td>
</tr>
<tr>
<td>atomic_ref_acq_rel</td>
<td>memory_order_acquire</td>
<td>memory_order_acquire</td>
<td>memory_order_release</td>
</tr>
<tr>
<td>atomic_ref_seq_cst</td>
<td>memory_order_seq_cst</td>
<td>memory_order_seq_cst</td>
<td>memory_order_seq_cst</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Accessor</th>
<th>reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_accessor</td>
<td>atomic_ref</td>
</tr>
<tr>
<td>atomic_accessor_relaxed</td>
<td>atomic_ref_relaxed</td>
</tr>
<tr>
<td>atomic_accessor_acq_rel</td>
<td>atomic_ref_acq_rel</td>
</tr>
<tr>
<td>atomic_accessor_seq_cst</td>
<td>atomic_ref_seq_cst</td>
</tr>
</tbody>
</table>

- And plugging `atomic_accessor*` into `mdspan` just works!
P2763 LAYOUT_STRIDE STATIC EXTENTS DEFAULT CONSTRUCTOR FIX

Christian Trott, Damien Lebrun-Grandie, Mark Hoemmen, Nevin Liber

- layout_stride default constructor
  - Was default constructed
  - Now initializes extents_ and strides_

- Issaquah 2023 for C++23
  - LEWG: Send P2763R0 (layout_stride static extents default constructor fix) to Library for C++23 as the resolution to LWG3861, classified as B2 - improvement.
  6 4 2 0 0
  - LWG: put P2763r1 into C++23? 12 1 0
P2798 FIX LAYOUT MAPPINGS ALL STATIC EXTENT DEFAULT CONSTRUCTOR
P2798 FIX LAYOUT MAPPINGS ALL STATIC EXTENT DEFAULT CONSTRUCTOR

Christian Trott, Damien Lebrun-Grandie, Mark Hoemmen

- When default constructing a `layout_left`, `layout_right` or `layout_stride` mapping with all static extents, there is currently no precondition check guarding against overflow of the `index_type`.

Mandates: If `Extents::rank_dynamic() == 0` is true, then the size of the multidimensional index space `Extents()` is representable as a value of type `typename Extents::index_type`.

- Addresses LWG3876 Default constructor of `std::layout_XX::mapping` misses precondition
- Approved Issaquah 2023 for C++23
P2897 ALIGNED_ACCESSOR: AN MDSPAN ACCESSOR EXPRESSING POINTER OVERALIGNMENT
P2897 ALIGNED_ACCESSOR: AN MIDSPAN ACCESSOR EXPRESSING POINTER OVER ALIGNMENT

Damien Lebrun-Grandie, Mark Hoemmen, Nicolas Morales, Christian Trott

- mdspan accessor policy that uses C++20 assume_aligned to decorate pointer access.
  - Wrapper around assume_aligned like atomic_accessor* is wrapper around atomic_ref*.

```cpp
template< std::size_t N, class T >
[[nodiscard]]
constexpr
T* assume_aligned( T* ptr );
```

- Informs the implementation that the object ptr points to is aligned to at least N.

- Scheduled to be discussed 20 days from now in LEWG telecon
CONCLUSIONS
CONCLUSIONS

- `mdspan`
  - A *non-owning* multidimensional array view vocabulary type
  - Took eight years to be designed, refined and standardized
    - Based on many more years of practical deployment experience
    - Buy-in from a wide variety of interested parties
    - Not Designed By Committee
      - Rather *Consensus* By Committee
  - The result is something *really good*
    - Flagship library of C++23!
SPECIAL THANKS:
ARGONNE NATIONAL LABORATORY
KOKKOS TEAM
C++ COMMITTEE
KHRONOS SYCL COMMITTEE

...AND A CAST OF TENS? HUNDREDS?

(THANK THEM / BLAME ME)
This was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative. Additionally, this research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.