CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Advanced C++ Course

std::mdspan

CSCS

Motivation

- How can we deal with multi-dimensional data with different layout: C-Layout vs Fortran-Layout

Every HPC software dealing with multi-dimensional data implements their own:

- allocation of multi-dimensional data (host/device, alignment, layout)
- accessing multi-dimensional data (abstract layout)
- iteration/algorithms over multi-dimensional data (in HPC mostly domain specific, highly optimized)

How to interface between libraries?

- Best case: well-defined/described concept that can be modelled by the *other* library
- Worst case: concrete class

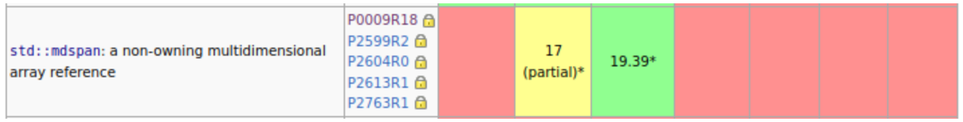
`md::span` **is**

- **a non-owning multi-dimensional array view**
- **meant to be used in interfaces**

What is std::mdspan?

`std::mdspan` is a non-owning multi-dimensional array view

- since C++23, see https://wg21.link/p0009 and https://eel.is/c++draft/views#multidim



- think of *pointer* and *metadata* (how to interpret the pointed-to memory)

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

- *TriviallyCopyable**: can be used in host/device interfaces
- allows different layouts

* under some costraints

std::mdspan<...>

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

```
std::vector<float> v(100);
auto v_span = std::mdspan(v.data(), std::extents{ 10, 10 });
```

- `T` is the element type (`my_mdspan::element_type`): `float`
- `Extents` describes number of dimensions and there sizes (required to be spezialization of `std::extents`)
- `LayoutPolicy` describes memory layout (`my_mdspan::layout_type`): default `std::layout_right` (C-layout)
- `AccessorPolicy` allows customization how we access the data (`my_mdspan::accessor_type`): think `std::default_accessor<T>` does pointer dereference of `T*`

```
v_span[2, 3] = 42.;
```

std::extents

- can describe run-time and compile-time extents
- compile-time extents are helpful for optimizations (explicit by library implementor or implicit by compiler)
- very easy (thanks to CTAD): `std::extents{ 10, 10 }`

```
template< class IndexType, std::size_t... Extents >
class extents;
```

- `IndexType` is a signed or unsigned integer type
- each element of `Extents` is either
  - `std::dynamic_extent` or
  - number representable in `IndexType` (compile-time extents)

std::extents

Examples

```
auto ext1 = std::extents<int, std::dynamic_extent, 3, std::dynamic_extent, 4>{ 42, 43 };
static_assert(decltype(ext1)::rank() == 4);
static_assert(decltype(ext1)::rank_dynamic() == 2);
static_assert(decltype(ext1)::static_extent(0) == std::dynamic_extent);
assert(ext1.extent(0) == 42);
static_assert(decltype(ext1)::static_extent(1) == 3);
```

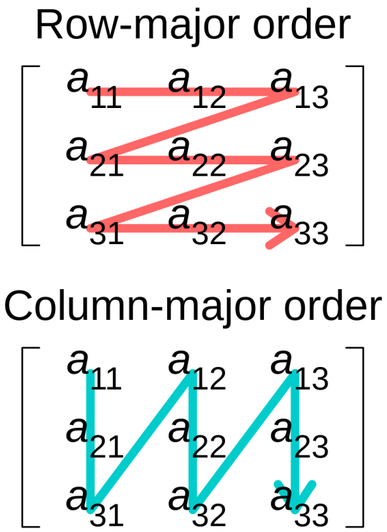
```
auto ext2 = std::extents<std::uint8_t, 3, 4>{};
static_assert(decltype(ext2)::static_extent(0) == 3);
static_assert(decltype(ext2)::static_extent(1) == 4);
```

```
auto ext3 = std::extents{ 42, 44 };
static_assert(decltype(ext3)::static_extent(42) == std::dynamic_extent);
assert(ext3.extent(0) == 42);
```

```
auto ext4 = std::dextents<int, 3>{ 42, 43, 44 };
```

LayoutPolicy

- provided policies:
  - `std::layout_right` (default, row-major, C-layout)
  - `std::layout_left` (column-major, Fortran-layout)
  - `std::layout_stride` generalization for arbitrary strides
- custom layout:
  - skip elements (e.g. tiling)
  - multiple indices to the same element




```cpp
auto s = std::mdspan(some_ptr, std::layout_stride::mapping(std::extents(2, 5, 10), std::array{ 5, 1, 10 }));

assert((some_ptr[1*5 + 2*1 + 3*10] == s[1, 2, 3]));
```

Layout example: matrix vector multiply*

```
using layout = /* see-below */;

std::mdspan<double, std::extents<int, N, M>, layout> A = ...;
std::mdspan<double, std::extents<int, N>> y = ...;
std::mdspan<double, std::extents<int, M>> x = ...;

std::ranges::iota_view range{0, N};

std::for_each(std::execution::par_unseq,
  std::ranges::begin(range), std::ranges::end(range),
  [=](int i) {
     double sum = 0.0;
     for(int j = 0; j < M; ++j) {
       sum += A[i, j] * x[j];
     }
     y[i] = sum;
  });
```

- on CPUs: C-layout aka row-major aka std::layout_right performs well (vectorized inner loop)
- on GPUs: Fortran-layout aka column-major aka std::layout_left performs well (coalesced memory load)

* from https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p0009r18.html 2.6

AccessorPolicy

Example

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

  // some ctors

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

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

Custom Accessor Example: protect host/device access

```
template <class ElementType, DeviceType Device>
struct host_device_protector {
    using offset_policy = host_device_protector; using element_type = ElementType;
    using reference = ElementType&; using data_handle_type = ElementType*;

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

    constexpr reference access(data_handle_type p, size_t i) const noexcept {
#ifdef __CUDA_ARCH__
        static_assert(Device == DeviceType::CUDA);
#else
        static_assert(Device == DeviceType::CPU);
#endif
        return p[i];
    }
};
```

```
void test_host_device_protector() {
    float* dev_ptr = allocate_some_cuda_memory<float>(4);
    auto s = std::mdspan<float, std::dextents<int, 2>, std::layout_right, host_device_protector<float, DeviceType::CPU>>{ dev_ptr, std::dextents<int, 2>{ 2, 2 } };
    std::cout << s[1, 2] << std::endl;
}
```

```
> error: static assertion failed due to requirement '(DeviceType)1 == DeviceType::CPU'
```

Status of multi-dimensional C++

✅ accessing multi-dimensional data (mdspan)
❌ allocating multi-dimensional data
❌ iterating multi-dimensional data / multi-dimensional algorithms (see *)

In C++26 we will get:

- very likely: `std::submdspan`
- likely: `std::mdarray`

* Multidimensional C++, Bryce A. Lelbach at CppNorth 2022

Example: mdspan from custom mdarray*

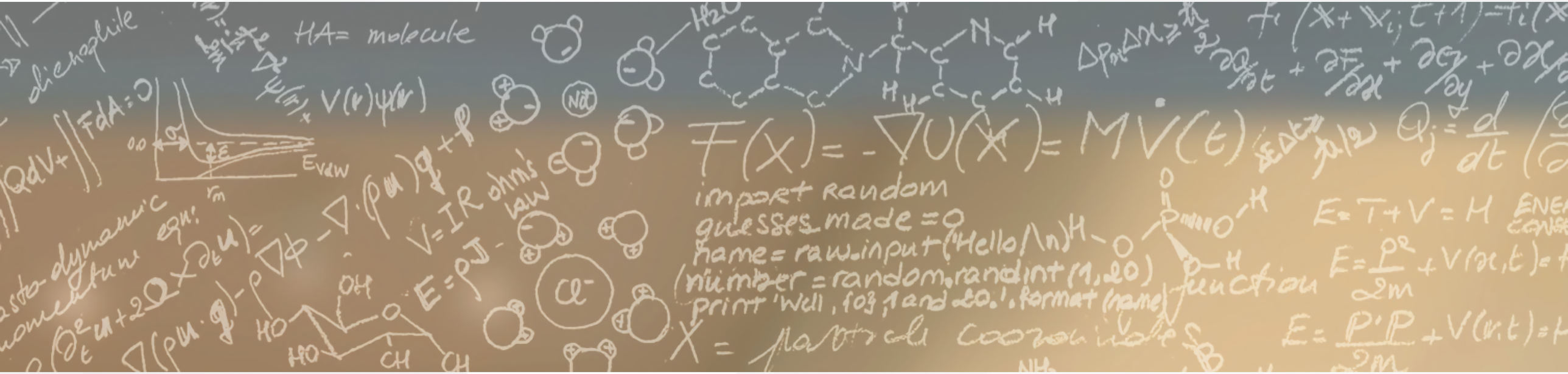
```
template <class T, std::size_t N>
struct my_mdarray {
    std::vector<T> data_;
    std::array<int, N> sizes_;

    my_mdarray(std::convertible_to<int> auto... sizes) : data_((sizes * ...)), sizes_{ sizes... } {
    }

    std::mdspan<T, std::dextents<int, N>> mdspan() {
        return { data_.data(), std::dextents<int, N>{ sizes_ } };
    }

    operator std::mdspan<T, std::dextents<int, N>>() {
        return { data_.data(), std::dextents<int, N>{ sizes_ } };
    }
};
```

* std::mdarray is proposed in https://wg21.link/p1684


HA= molecule
V(r)ψ(r)
F(X)= -∇U(X)= MV(t)
import random
guesses_made=0
name=raw_input("Hello!\n)H
(number=random.randint(1,20)
print 'Well, fog, 1 and 20.', format(name)
E=T+V=H
E=p²/2m +V(r,t)
E=P'P/2m +V(r,t)
E=ρJ
V=IR ohms law
Evdw

Questions?