HPC-BP Webinar Q&A

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

Date: 18 February 2026
Presented by: Christian Trott (Sandia National Laboratories)

(The slides are available via the link in the page's sidebar.)

Q: Please make some comments about how to limit the complexity (and improve the understandability) of compiler error messages through use of constrained templates and concepts.

From the audience: Modern compilers even interpret `std::enable_if` etc. with the same error messages that they would give for `requires` clauses or using concepts.

A: As said above: modern compilers already give decent error messages with std::enable_if (assuming you use it in canonical ways) which look very similar to error messages with require clauses. One thing concepts do for generic code is create error messages at the callside, instead of deep down inside implementation details. So that improves readability, by interrupting the template error mess early.

Q: Is inspecting mdspan objects in debuggers like gdb / lldb straight forward in their native form or have you found yourself using helper functions?

From the audience: mdspan exposes its three components via .data_handle(), .mapping(), and .accessor(). mdspan stores these three components. They don't need a unique address so the object might not actually exist in memory if it has size zero (that's normally the case for default_accessor, or for layout_left or layout_right with all compile-time extents).

A: Generally I didn't find the stuff to be too bad to read in debuggers. Normally the three members show up reasonably readable and they themselves are well structured. You kinda just need to get use to the structure.

Q: Would you consider using `dims` instead of `dextents` in future presentations? `dims` is shorter and has better defaults : -) . e.g., `dims<2>` instead of `dextents<size_t, 2>`. The reference mdspan implementation that Kokkos uses has a back-port of `dims` to C++ < 23 so it would work just fine for you. Also, `submdspan` is a C++26 feature and I'm sure you'll be talking about `submdspan` in talks : -) .

An exchange within the audience

- The layout describes the mapping from a multidimensional index i, j, k, ... to a 1-D offset. That has nothing to do with ownership.
- I'm thinking about preventing defining an mdspan with right layout when the backing array has left layout.
- How would you represent the backing array when defining the mdspan? If you're starting with a layout_left mdspan, the compiler forbids you from converting it to a layout_right mdspan.
- I suppose then, if some library hands me a pointer to a multi-dim array (or so it promises), I should first create an mdspan with the promised layout to refer to it, and later a sub-mdspan of that if I need it. Sound right?
- That's right! Applying an mdspan to a 1-D array asserts that the 1-D array can be interpreted as a multidimensional array with the mdspan's layout.

A: (The audience discussion doesn't seem to have anything to do with the question). Regarding dims: I know Mark likes it, I don't much because it hides the index type. Also in GPU land I am writing more often than not dextents<int, 2>, and yes you could write that as dims<2, int> but: mdspan doesn't have a dims() member function it has an extents() member function. It also doesn't have dim(i) it has extent(i) [to get a single extent]. So using extents or dextents as the spelling for that template parameter keeps the connection clearer in my mind.

Q: If mdspan is non-owning, why does it get to freely define a layout?

A: Its a description of memory. You allocate raw memory buffers and then you build an mdspan around it that describes how you interpret that buffer. That is actually a fairly common approach in HPC (i.e. central managed buffers, and then you build higher level datastructures around it). But basically you use mdspan to describe, and enforce how that memory should be interpreted.

Q: Does `std::mdspan` now work on GPUs?

From the audience: libcu++ (see https://github.com/NVIDIA/CCCL) has a beautiful mdspan implementation that works on GPUs if you're interested in trying it out! nvc++ also has an mdspan implementation that works in the parallel algorithms, that can run on the GPU.nvc++ also has an mdspan implementation that works in the parallel algorithms, that can run on the GPU.

A: GCC and LLVM versions actually kinda work too at least with ROCM, since its all constexpr functions, and ROCM just accepts those as device functions. That said the github.com/kokkos/mdspan is a very solid implementation which is getting tested with practically every toolchain used in HPC, and is used in production for everything that is build on top of Kokkos itself.

Q: What is SFINAE?

From the audience:
“Substitution failure is not an error” It’s a “soft’ error, as if the function didn’t exist. As opposed to a hard error, such as static_assert, which will stop your compilation. See https://en.cppreference.com/w/cpp/language/sfinae.html

Q:How does the “requires expression” provide stronger guarantees than standard templating? As far as I am aware syntactic validity is already a requirement for a template overload to materialize.

From the audience: There's a big difference between "function not considered for overload resolution" and "function is ill-formed." The latter means the compiler tries to compile the function (EXPENSIVELY) and the user gets a nasty big error message. The former means that the user immediately sees from the declaration of the function: "Hey, your input parameters don't have the right types to match this function."

A: (Audience answer is good) In standard templating you can’t put more complex requirements on something. Yes you can write something like
template<class T, class E, class L, class A>
void foo(mdspan<T,E,L,A> a)

To make sure only mdspan matches this, but adding the “only mdspan with rank-2, because I gonna index into it with two integers) is more cumbersome.

Q: What would be a non-invastive way of checking that concepts work as intended in unit tests? Is there an elegant way to check that the correct function was called? Or should outputs / side effects be tested instead?

From the audience: You can use a concept in a `static_assert`, just like a type trait. That's a good way to write unit tests for concepts. It’s not easy, but compile-time testing is possible in CMake, to verify (e.g.) that unwanted arguments are actually barred as expected. See also https://github.com/uliegecsm/reprospect

A: You want to hit the overload. Not sure there is a “right” way of checking you hit the right overload. In Kokkos we use the tools interface to test whether we go through the code paths we expect. It hooks into customization points of our runtime to provide output and check expected output. In general, that’s not a thing you can’t figure out which overload was called. Chat comment from Chris Green: Can do compile time testing. “Re: testing: it’s not easy, but compile-time testing is possible in CMake, to verify (e.g.) that unwanted arguments are actually barred as expected.” Christian: Yes, you can check for compile-time failure. Not possible to check which overload you hit. Good practice to check if you give the wrong thing at compile time that it errors out.

Q: Is C++20 moving to mostly header defined libraries similar to just in time compilation (e.g. like Julia)? Precompiling libraries seems to be a combinatory explosion.

A: C++ is almost all header-only. Very little precompiled because it’s templated; called STL for a reason.

Q:What is the main advantage of using mdspan?

A: Probably most concise and complete expression of how an actual multi-dimensional array is stored in memory. Full description. At compile time - you can reason about it and check at compile time with your function interfaces.

Q: The last release of https://github.com/kokkos/mdspan is about 3 years old. Would you mind creating a new release to get the latest fixes into spack, conda etc.?

A: We are planning to do that. Defacto release in Kokkos library where we repackage it.

Q: Would using https://github.com/kokkos/mdspan with more recent C++ standards (but with compilers who don’t have std::mdspan) allow to use the [ ] syntax?

A: Yes. In C++23 mode it’s enabled.