SAND2019-10501C

mdspan in C++: A Case Study in the Integration of Performance Portable Features into International Language Standards

David S. Hollman
Scalable Modeling & Analysis
Sandia National Laboratories
Livermore, CA, USA
dshollm@sandia.gov

Bryce Adelstein-Lelbach
NVIDIA Corp.
Santa Clara, CA, USA
blelbach@nvidia.com

H. Carter Edwards
NVIDIA Corp.
Santa Clara, CA, USA
hedwards@nvidia.com

Mark Hoemmen
Engineering Sciences
Sandia National Laboratories
Albuquerque, NM, USA
mhoemme@sandia.gov

Daniel Sunderland
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
dsunder@sandia.gov

Christian R. Trott
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
crtrott@sandia.gov

***Abstract*—Multi-dimensional arrays are ubiquitous in high-performance computing (HPC), but their absence from the C++ language standard is a long-standing and well-known limitation of their use for HPC. This paper describes the design and implementation of `mdspan`, a proposed C++ standard multidimensional array view (planned for inclusion in C++23). The proposal is largely inspired by work done in the Kokkos project—a C++ performance-portable programming model deployed by numerous HPC institutions to prepare their code base for exascale-class supercomputing systems. This paper describes the final design of mdspan after a five-year process to achieve consensus in the C++ community. In particular, we will lay out how the design addresses some of the core challenges of performance-portable programming, and how its customization points allow a seamless extension into areas not currently addressed by the C++ Standard but which are of critical importance in the heterogeneous computing world of today's systems. Finally, we have provided a production-quality implementation of the proposal in its current form. This work includes several benchmarks of this implementation aimed at demonstrating the zero-overhead nature of the modern design.**

***Index Terms*—programming models, C++, performance portability, programming languages, stardardization, multidimensional array, data structures**

Introduction

One of the primary concerns of the high-performance computing (HPC) community [1] is performance portability. *Performance portability* means that a single code base can perform well on many different platforms. Over the last decade in particular, numerous projects [2]–[6] have tried to address various challenges associated with it. The recent announcement of the first exascale-class platforms, introducing architectures which were previously not deployed in the HPC community, has increased the urgency of finding solutions to performance portability concerns. One of the projects which has found significant success in adoption is Kokkos, [6], [7] a C++ performance-portable programming model originally developed at Sandia National Laboratories, but now maintained by a group spanning four United States National Laboratories as well as the Swiss National Supercomputing Centre.

Arguably the most significant innovation of the Kokkos project was its `View` data structure, a multi-dimensional array abstraction which addresses concerns of performance portability such as data layout and data access customization. This array abstraction is now used at the heart of many HPC software projects [8], and is proving to be critical for meeting the challenges of preparing code bases for the exascale era. While maintaining these capabilities in an HPC-specific solution is workable for now, there are a number of reasons why it would be beneficial to have the core capabilities become part of international programming language standards. Doing so would enable tighter integration into other language and library capabilities, such as the proposed ISO C++ Linear Algebra library [9]. It would simplify interface compatibility between different HPC products, and would further seamless integration with external products used in applications not specific to HPC. For example, the proposed ISO C++ Audio library [10] has expressed interest in using this abstraction.

This work was carried out in part at Sandia National Laboratories. Sandia National Laboratories is a multi- mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidary of Honeywell International Inc., for the U. S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

To that end the Kokkos team initiated a collaboration with other stakeholders to design a multi-dimensional array for the ISO C++ Standard, that also addresses the concerns of performance portability addressed by Kokkos' `View`. The result of this over five-year process is `std::mdspan`, described herein and proposed to the ISO C++ standard in the proposal P0009 [11]. The design allows for a mix of static and dynamic array dimensions, enables control of the data layout, and has customization points to control how data are accessed. The latter includes use cases that involve hardware-specific special load and store paths.

In this work we describe each of the design aspects of `mdspan`, with examples demonstrating their impact for performance and portability concerns, as well as benchmarks of the production-quality reference implementation developed by the authors.

Design

`mdspan` provides a class template for creating types of objects that represent, but do not own, a contiguous piece (or "span") of memory that is to be treated as a multi-dimensional entity with one or more dimensional constraints. Together, these dimensional constraints form a *multi-index domain*. In the simple case of a two-dimensional entity, for instance, this multi-index domain encompasses the row and column indices of what is typically called a matrix. For instance,

```
void some_function(float* data) {
  auto my_matrix =
    mdspan<float, dynamic_extent, dynamic_extent>(
      data, 20, 40
    );
  /* ... */
}
```

says to create an object that interprets memory starting at the pointer `data` as a matrix with the shape 20 rows by 40 columns. Extents can be provided either statically (i.e., at compile time) or dynamically (as shown above), and static extents can be mixed with dynamic extents:

```
void another_function(float* data) {
  auto another_matrix =
    mdspan<float, 20, dynamic_extent>(
      data, 40
    );
  /* ... */
}
```

This code snippet also treats `data` as a 20 by 40 matrix, but the first of these dimensions is "baked in" to the type at compile time—all instances of the type `mdspan<double, 20, dynamic_extent>` will have 20 rows.

The design is greatly simplified by delegating the ownership and lifetime management of the data to orthogonal constructs. Thus, `mdspan` merely interprets existing memory as a multi-dimensional entity, leaving management of the underlying memory to the user. This follows a trend of similar constructs recently introduced to C++, such as `string_view` [12], [13] and `span` [14], [15]. These constructs allow "API funnelling," which makes it easy for libraries to support users' own types instead of forcing users to use a specific type. Library interfaces can take `string_view`s or `mdspan`s, and library users can add interfaces to their own types that return a suitable `string_view` or `mdspan`. This design pattern enables easy adoption by existing codebases which have their own matrix types. Since `mdspan` is non-owning, users can always create an `mdspan` that refers to a matrix owned by another object. Older abstractions also take this approach. Iterators, which have been central to C++ algorithm design for decades, are also non-owning entities which delegate lifetime management as a separate concern. [16]

References to entries in these matrices are obtained by giving a multi-index (that is, an ordered set of indices) to the object's `operator()`, which has been overloaded for this purpose:

```
// add 3.14 to the value on the row with index 10
// and the column with index 5
some_matrix(10, 5) += 3.14;
// print the value of the entry in the row with
// index 0 and the column with index 38
printf("%f", some_matrix(0, 38));
```

The length of each dimension is accessed via the `extent` member function. It takes an index to indicate the dimension. A loop to multiply all entries of the matrix by a scalar could thus look like this:

```
for(int row = 0; row < my_mat.extent(0); ++row)
  for(int col = 0; col < my_mat.extent(1); ++col)
    my_mat(row, col) *= 2.0;
```

Arbitrary slices of an `mdspan` can be taken using the `subspan` function:

```
auto my_tens = mdspan<float, 3, 4, 5, 20>(data);
auto my_matrix = subspan(my_tens,
  2, all, pair{2, 4}, 0
);
```

The above snippet creates a 4 by 2 matrix sub-view of `my_tens` where the entries `i, j` correspond to index 2 in the first dimension of `my_tens`, index `i` in the second dimension, `j+2` in the third dimension, and `0` in the fourth dimension. This relatively verbose syntax for slicing was preferred over other approaches, because slicing needs can vary substantially across different domains. Domain-specific syntax can easily be built on top of `subspan`.

Just as `std::string` is actually a C++ alias for `std::basic_string` [12], [13], `std::mdspan` is an alias for `std::basic_mdspan`. Whereas `std::mdspan` only provides control over the scalar type and the extents, `std::basic_mdspan` exposes more customization points. It is templated on four parameters: the scalar type, the extents object, the layout, and the accessor. In the following

sections, we will describe these parameters and their value in improving performance or increasing portability.

Extents Class Template

In `basic_mdspan` the extents are provided via an `extents` class template. As with the `mdspan` alias template, the parameters are either static sizes or the `dynamic_extent` tag.

```
void some_function(float* data) {
  auto my_matrix =
    basic_mdspan<float, extents<20, dynamic_extent>>(
      data, 40
    );
  /* ... */
}
```

The ability to provide extents statically can help significantly with compiler optimizations. For example, a compiler may be able to unroll small inner loops completely if the extents are known at compile time. Knowing exact counts and sizes can also help with vectorization and the optimizer's cost model. A typical example of this in HPC is operations on a batch of small matrices or vectors, where the dimensions of each item is dictated by a physics property or the way the system was discretized, rather than by the problem size. When this sort of problem interacts with generic code, such information would be lost unless static extents can be part of the `mdspan` type itself. The `TinyMatrixSum` benchmark (below) provides a proxy for problems with this sort of behavior.

Layout abstraction

Modern C++ design requires library authors to orthogonalize certain aspects of the design into customization points, over which algorithms may be written generically. The most commonplace example of this is the `Allocator` abstraction [12], [17], which controls memory allocation for standard containers like `std::vector` [12], [18]. Most algorithms on containers do not change regardless of how the underlying data is allocated. The `Allocator` abstraction allows such algorithms to be generic over the form of memory allocation used by the container.

An example of one such aspect in the current context is the layout of the underlying data with respect to the multi-index domain. While a high-quality-of-implementation matrix multiply would definitely specialize for different data layouts, the simplest possible implementation would only need to know how to get and store data associated with a given multi-index into the underlying memory. This also describes the majority of use cases from the perspective of the caller of such algorithms, where only the semantics of a mathematical matrix multiply are needed regardless of data layout. The grouping of a single set of mathematical semantics under a common algorithm name (regardless of layout) serves as a conduit for performance portability, and additionally reduces the cognitive load for the writer and particularly the reader of the code.

The canonical example, again with reference to data layout, is the portability of access patterns in code that may run on a latency-optimizer processed (e.g., CPU) or on a bandwidth-optimized processor (e.g., GPU). GPUs need to coalesce accesses (that is, stride across execution agents) because of the vector nature of the underlying hardware, whereas CPUs want to maximize locality (that is, assign contiguous chunks to the same execution agent) in order to increase cache reuse.

The abstraction for representing data layout generically is called the `LayoutMapping`. The primary task of the `LayoutMapping` is to represent the transformation of a multi-index into a single, scalar memory offset. A large number of algorithms on multi-dimensional arrays have semantics that depend only on the data as retrieved through the multi-index domain, indicating that this transformation is a prime aspect for orthogonalization into a customization point. (Note that many algorithms have *performance* characteristics that depend on this transformation, but the separation of semantic aspects of an algorithm from its performance characteristics is critical to modern programming model design. The fact that the `LayoutMapping` abstraction promotes this separation is further evidence of its utility as a customization point.)

A brief survey of existing practice (such as the BLAS Technical Standard [19], Eigen [20], and MAGMA [21]) reveals an initial set of layout mappings that such an abstraction must support. These include, at minimum,

- row-major or column-major layouts (represented by the `TRANS` parameters in BLAS), that generalize to describe layouts where the fast-running index is left-most or right-most;
- strided layouts (represented by the `LD` parameters in BLAS), that generalize to any in a class of layouts that can describe the distance in memory between two consecutive indices in a particular dimension with a constant (specific to that dimension); and
- symmetric layouts (e.g., from the `xSYMM` algorithms in BLAS), which also include generalizations like whether the upper or lower triangle is stored (the `UPLO` parameter in BLAS) and whether the diagonal is stored explicitly, implicitly, or in some separate, contiguous storage.

In addition to similarities, it also helps to look at what differences these layout mappings may introduce, over which some algorithms may not be generic. In general, as many previous researchers have noted, [22] the design of generic concepts for customization typically begins with the algorithms, not the data structures. Much of the design of `LayoutMapping` can be motivated with some very simple algorithms. Consider an algorithm, `scale`, that takes an `mdspan` and a scalar and multiplies each entry, in place, by the scalar. For brevity, we will only consider the two-

dimensional case here (though much of this motivation can be done even in the one-dimensional case). If such an algorithm is to be implemented in the simplest possible way—iterating over the rows and column indices and scaling each element—the implementation would fail to meet the semantic requirements of the algorithm for symmetric layouts, since non-diagonal entries reference the same memory. Thus, it is necessary for certain algorithms to know whether each multi-index in the domain maps to a unique offset in the codomain (the space of all offsets that could be valid results of the mapping). (An example of an algorithm for which this requirement is *not* needed is `dot_product`.) The `LayoutMapping` customization expresses this property through the requirement that it provide an `is_unique` method. Many algorithms are difficult or impossible to implement on general non-unique layouts. However, in the simple case of `scale`, the algorithm *could* be implemented for any layout that is simply *contiguous*, by viewing the codomain of the layout as a one-dimensional `mdspan` and scaling each item that way. Contiguousness is expressed through the requirement of an `is_contiguous` method, and the size of the codomain is expressed through the `required_span_size` required method. Similarly, as previously observed, many existing implementations (such as the BLAS) can specially handle any layout with regular strides. Layout mappings can express whether they are strided using the `is_strided` method. Finally, all of these aspects need to be expressible statically and dynamically, so for layout mappings where the uniqueness, stridedness, and continguousness are consistently `true` for all instances of the type, the `is_always_unique`, `is_always_strided`, and `is_always_contiguous` hooks are provided in the concept. These requirements allow, for instance, algorithms that cannot support layouts lacking certain properties to fail at compile time rather than run time. The requirements on the `LayoutMapping` concept are summarized in Table I.

TABLE I
REQUIREMENTS ON THE `LayoutMapping` CONCEPT

Expression	Meaning
`M`	A `LayoutMapping` type.
`m`	An instance of `M`.
`E`	A specialization of `std::extents`.
`e`	An instance of `E`.
`i...` and `j...`	Multidimensional indices in the multidimensional index space described by `e`.
`m.extents()`	The extents object `e` representing the multidimensional index domain of the mapping.
`m(i...)`	A nonnegative value representing the codomain offset corresponding to `i...`.
`m.required_span_size()`	The maximum value of `m(i...)` plus `1` if all extents are non-zero, or `0` otherwise.
`m.is_unique()`	`true` only if for every `i... != j...`, `m(i...) != m(j...)`.
`m.is_contiguous()`	`true` only if the set defined by all `m(i...)` equals the set $\{0, ..., $`m.required_span_size()` $- 1\}$.
`m.is_strided()`	`true` only if $\forall r \in [0,$ `e.rank()`$), \exists K_r$ such that $\forall$`i...`, `j...` $\in$ `e`, if all elements of `i...` and `j...` are equal except for the r^{th} element, with $j_r = i_r + 1$, then K_r equals `m(j...) - m(i...)`.
`m.stride(r)`	The integer K_r, as described above. Only required if `m.is_strided()` is `true`.
`M::is_always_unique()`	`true` only if `m.is_unique()` is `true` for all instances of `M`.
`M::is_always_contiguous()`	`true` only if `m.is_contiguous()` is `true` for all instances of `M`.
`M::is_always_strided()`	`true` only if `m.is_strided()` is `true` for all instances of `M`.

Accessor abstraction

After several design iterations, [11] the authors came to the conclusion that many of the remaining customizations could be encapsulated in the answer to one question: how should the implementation turn an instance of some pointer type and an offset (obtained from the `LayoutMapping` abstraction) into an instance of some reference type? The `Accessor` customization point is designed to provide all of the necessary flexibility in the answer to this question. Our exploration in this space began with a couple of specific use cases: a non-aliasing `Accessor`, similar to the `restrict` keyword in C, [23] and an atomic `Accessor`, where operations on the resulting reference use atomic operations. The former needs to customize the pointer type to include implementation-specific annotations (usually some variant of the C-style `restrict` keyword) that indicate the pointer does not alias pointers derived from other sources within the same context (usually a function scope). The latter needs to customize the reference type produced by the dereference operation to have it return a `std::atomic_ref<T>`. (`std::atomic_ref<T>` was merged into the C++ Standard working draft during the C++20 cycle, and will likely be officially approved as part of the C++20 balloting process when that process completes sometime in 2020 [24]. These requirements immediately led us to include customizable `reference` and `pointer` type names as part of the `Accessor` concept. Marrying these two customizations could take several forms. One possibility is to have a function that simply takes a `pointer` and returns a `reference`. However, this requires the `pointer` type to be arbitrarily offsettable—e.g., using `operator+` or `std::advance`. A simpler approach that removes this requirement is to have a customization point that takes the `pointer` and an offset and returns the `reference` directly. We chose the latter in order to simplify the requirements on the `pointer` type, and named this required method `access`.

The issue of offsetting a `pointer` to create another `pointer`, while not necessarily separable from the creation of a `reference`, is nonetheless also a concern that `Accessor`

needs to address for the implementation of the `subspan` function. We named this customization with a required method `offset`. The type of the `pointer` retrieved when arbitrarily offsetting a `pointer` type may not necessarily match the input pointer type. For instance, in the case of an overaligned pointer type used for easy vectorization, a pointer derived from an arbitrary (runtime) offset to this pointer cannot guarantee the preservation of this alignment. Thus, the `Accessor` is allowed to provide a different `Accessor`, named with the required type name `offset_policy`, that differs in type from itself (and thus, for instance, may differ in its `pointer` type). Finally, given an arbitrary `pointer` type, the current design requires the ability to "decay" this type into an "ordinary" C++ pointer for compatibility with `std::span`, which does not support `pointer` type customization. The requirements on the `Accessor` concept are summarized in Table II.

TABLE II
REQUIREMENTS ON THE ACCESSOR CONCEPT

Expression	Meaning
`A`	An `Accessor` type.
`a`	An instance of `A`.
`A::element_type`	The type of each element in the set of elements described by the associated `mdspan`.
`A::pointer`	The pointer type through which a range of elements are accessed.
`p`	An instance of `A::pointer`.
`i`	Non-negative value of type `ptrdiff_t`.
`A::reference`	The type through which an element is accessed. Must be convertible to `A::element_type`.
`A::offset_policy`	(Optional) An `Accessor` type convertible from `A`. Defaults to `A`.
`a.access(p, i)`	Returns an object that provides access to the `i`-th element in the range of elements that starts at `p`.
`a.offset(p, i)`	An instance of `A::offset_policy::pointer` for which `A::offset_policy(a).access(a.offset(p, i), 0)` references the same element as `a.access(p, i)`.

Accessor Use Case: Non-Aliasing Semantics: As a concrete example, the (trivial) `Accessor` required to express non-aliasing semantics (similar to the `restrict` keyword and supported in many C++ compilers as `__restrict`) is shown in Figure 1. This differs from the default accessor (`std::accessor_basic<T>`) only in the definition of the nested type `pointer`. Interestingly, because the design of `mdspan` requires the `pointer` to be used as a parameter (in `access`) before it is ever turned into a reference, `mdspan` is able to skirt the well-known issues surrounding the meaning of the `restrict` qualifier on a data member of a struct. [25]–[27]

Accessor Use Case: Atomic Access: Frequently in HPC applications, it is necessary to access a region of memory atomically for only a small portion of its lifetime. Constructing the entity to be atomic for the entire lifetime of the underlying memory, as is done with `std::atomic`, may have unacceptable overhead for many HPC use cases. As an entity that references a region of memory for a subset of that memory's lifetime, `mdspan` is ideally suited to be paired with a fancy reference type that expresses atomic semantics (that is, all operations on the underlying memory are to be performed atomically by the abstract machine). With the introduction of `std::atomic_ref` in C++20, all that is needed is an accessor that customizes the reference type and provides an `access` method that constructs such a reference. An implementation of such an `Accessor` is shown in Figure 2.

```
template <class T>
struct RestrictAccessor {
  using element_type = T;
  using pointer = T* __restrict;
  using reference = T&;
  reference access(pointer p, ptrdiff_t i)
    const noexcept
  { return p[i]; }
  pointer offset(pointer p, ptrdiff_t i)
    const noexcept
  { return p + i; }
};
```

Fig. 1. An Accessor that provides an expression of non-aliasing semantics for mdspan.

```
template <class T>
struct AtomicAccessor {
  using element_type = T;
  using pointer = T*;
  using reference = atomic_ref<T>;
  reference access(pointer p, ptrdiff_t i)
    const noexcept
  { return atomic_ref{ p[i] }; }
  pointer offset(pointer p, ptrdiff_t i)
    const noexcept
  { return p + i; }
};
```

Fig. 2. An Accessor that provides an expression of atomic reference semantics for mdspan.

Accessor Use Case: Bit-Packing: Similar to the infamous `std::vector<bool>`, the accessor abstraction can be used to return a fancy reference type that references individual bits packed into the bytes of underlying memory. (Unlike `std::vector<bool>`, though, `std::accessor_basic<bool>` does not do this by default).

Accessor Use Case: Strong Pointer Types for Heterogeneous Memory Spaces: Heterogeneity often requires a program to access multiple, potentially disjoint memory spaces. Thus far, vendor-provided APIs for heterogeneity have tended to represent this memory with plain-old raw pointers. An important emerging paradigm in modern programming model design is so-called "strong types" (also called "opaque typedefs" or "phantom types"), [28], [29] wherein meaning is opaquely attached to the form of the type. For instance, `distance<double>` and `temperature<double>` might be

different concrete types that the compiler forbids mixing, even though they both would use `double` for storage and arithmetic. Applied to heterogeneity, the paradigm would suggest replacing raw pointers with opaque typedefs indicating things like their compatibility or accessibility. This not only introduces safety with respect to memory access by an execution resource, but also allows generic software design strategies where execution mechanisms can be deduced from the type of the data structure. In `mdspan`, such strong typing can be injected via the customization of the associated pointer type in the `Accessor`. Initially, of course, such extensions will be outside of the C++ Standard (e.g., OpenMP, HIP, SYCL, and older versions of CUDA), but this design provides a means of forward compatibility if and when the Standard addresses the concept of heterogeneous memory resources natively in C++.

Implementation

This work is accompanied by a production-oriented implementation of the proposed `std::mdspan`, available at github.com/kokkos/mdspan, the details of which are discussed in this section. While the proposal it implements targets C++23, the implementation includes compatibility modes for C++17, C++14, and C++11. The implementation also includes a couple of macros, `MDSPAN_INLINE_FUNCTION` and `MDSPAN_FORCE_INLINE_FUNCTION`, that can be used to add the appropriate markings to functions and function templates, such as `__device__` for CUDA compatibility. It has been tested on various versions of numerous C++ compilers, including GCC, Clang, Intel's ICC, Microsoft's MSVC, and NVIDIA's NVCC. The implementation differs modestly from the proposal in several places (mostly with respect to typos in the latter), all of which are documented in the implementation repository (link given above).

Benchmarks

A common complaint about C++ abstractions in HPC is that they hinder compiler optimizations. While that was largely true in the past, a number of developments have improved the situation. More recent C++ standards introduce capabilities and constraints which help the compiler optimize code. Furthermore, with the widespread adoption of C++ abstraction layers in industry, significant work has gone into optimizing commonly used compilers. To demonstrate that `mdspan` does not introduce overheads compared to using raw pointers with manual indexing, we will show benchmark results both for the version using `mdspan` and an equivalent implementation using raw pointers. Since the difference in most benchmarks is very small, most figures in this section show overhead of the `mdspan` version over the raw pointer variant. Negative overhead indicates cases where the `mdspan` version was faster.

Figure 3 shows a normalized comparison of `mdspan` versions of several selected benchmarks with the same benchmark expressed with raw pointers. A more thorough elaboration follows. Most of the benchmarks showed overheads within the measurement noise, and no benchmarks showed overhead greater than 10%. Examination of generated assembly (and, at least in the case of the Intel compiler, optimization reports) indicates similar—usually identical—vectorization of the `mdspan` and raw pointer versions of our benchmarks.

Methodology

All benchmarks were prepared and executed using the Google Benchmark microbenchmarking support library. [30] Table III lists the test systems and compilers used for benchmarking. Unless otherwise stated, CPU benchmarks were run on Mutrino, and GPU benchmarks were run on Apollo. CPU benchmarks are serial unless labeled "OpenMP", in which case they were parallelized with the OpenMP `parallel for` directive on the outermost loop (with the intent of measuring typical basic usage of OpenMP). CPU benchmarks were compiled with GCC 8.2.0, Intel ICC 18.0.5, and the latest Clang development branch (GitHub hash `1fcdcd0`, which is LLVM SVN revision 370135; labeled as "Clang 9 (develop)" herein). GPU benchmarks were compiled with NVIDIA's NVCC version 10.1, using GCC 5.3.0 as the host compiler. The source code of all benchmarks is available on the mdspan implementation repository that accompanies this paper (see Implementation section above). A brief description of each benchmark is also included here for completeness. These benchmarks tend to focus on the three-dimensional use case (which we view as the smallest "relatively non-trivial" number of dimensions), but spot checks with larger numbers of dimensions—up to 10—yielded similar results and led to similar conclusions.

TABLE III
Test Systems and Software

Machine	Hardware	Compilers
Apollo	NVIDIA TitanV	CUDA 10.1 (nvcc), GCC 5.3.0
Astra	Cavium Thunder-X2 CN9975-2000	GCC 8.2 (ARM)
Blake	Intel(R) Xeon(R) Platinum 8160	Intel 19.3.199, Intel 18.0.5
Mutrino	Intel(R) Xeon(R) CPU E5-2698 v3	Intel 18.0.5, GCC 8.2.0, Clang 9 (develop)

`Sum3D` Benchmark: Intended as a "simplest possible" benchmark, this benchmark simply sums over all of the entries in a 3D `mdspan`. The raw pointer version (as with all of the benchmarks) does the same thing, but uses hard-coded index arithmetic. Both right-most fast-running and left-most fast-running loop structures and layouts were tested (and yielded similar results), and only the right layout, right loop structure results are discussed in this paper, for

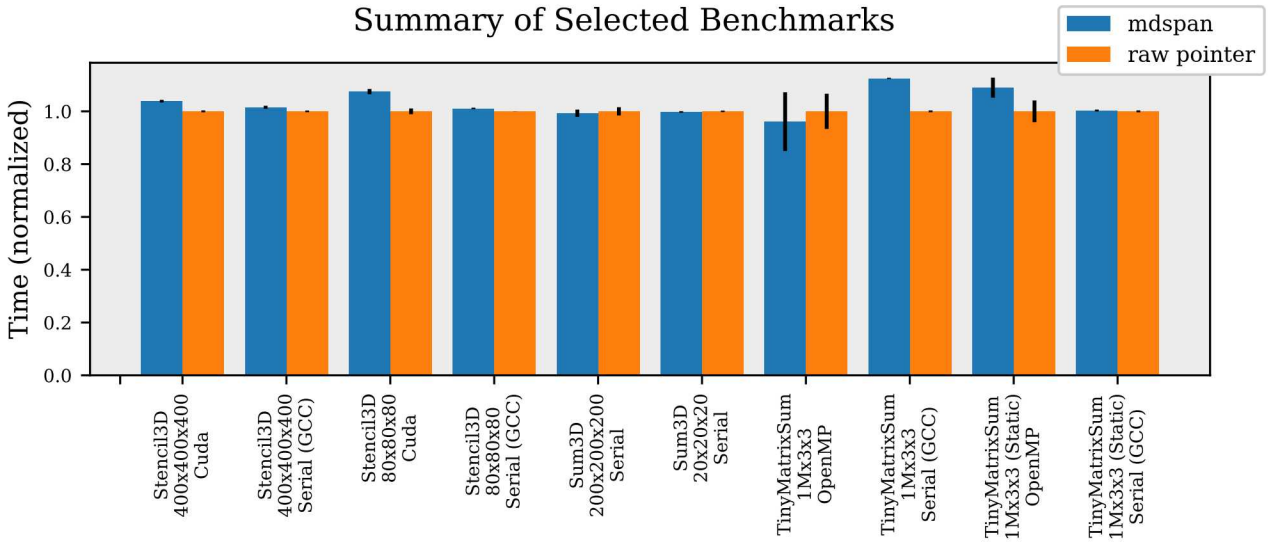

Fig. 3. An overview of selected benchmark comparisons of mdspan and raw pointer performance. Each benchmark is normalized to the average execution time of the raw pointer case. Details of each of these benchmarks are described in the text.

brevity. The relevant portion of the source code for this benchmark, for an input `mdspan` named `s` and an output named `sum`, looks like:

```
for(ptrdiff_t i = 0; i < s.extent(0); ++i) {
  for (ptrdiff_t j = 0; j < s.extent(1); ++j) {
    for (ptrdiff_t k = 0; k < s.extent(2); ++k) {
      sum += s(i, j, k);
    }
  }
}
```

`Stencil3D` Benchmark: This benchmark takes the sum of all of the neighboring points in three-dimensional space from an input `mdspan` and stores it in the corresponding entry of the output `mdspan`. In terms of structured grid computations, it has a "stencil size" of one. The deduced index type is `ptrdiff_t`. The relevant portion of the source code for this benchmark, for an input `mdspan` named `s` and an output `mdspan` named `o`, looks like this:

```
for(auto i = d; i < s.extent(0)-d; i ++) {
  for(auto j = d; j < s.extent(1)-d; j ++) {
    for(auto k = d; k < s.extent(2)-d; k ++) {
      value_type sum_local = 0;
      for(auto di = i-d; di < i+d+1; di++) {
        for(auto dj = j-d; dj < j+d+1; dj++) {
          for(auto dk = k-d; dk < k+d+1; dk++) {
            sum_local += s(di, dj, dk);
          }
        }
      }
      o(i,j,k) = sum_local;
    }
  }
}
```

`TinyMatrixSum` benchmark: This benchmark applies a batched sum operation to large number of small (in this paper, 3x3) matrices, accumulating from the input Nx3x3 `mdspan` into an Nx3x3 `mdspan`. The relevant portion of the source code for this benchmark, for an input `mdspan` named `s` and an output `mdspan` named `o`, looks like this:

```
for(ptrdiff_t i = 0; i < s.extent(0); i ++) {
  for(ptrdiff_t j = 0; j < s.extent(1); j ++) {
    for(ptrdiff_t k = 0; k < s.extent(2); k ++) {
      o(i,j,k) += s(i,j,k);
    }
  }
}
```

`Subspan3D` benchmark: This benchmark performs the same operations as the `Sum3D` benchmark, but uses three calls to `subspan`, instead of accessing the entries of the `mdspan` in the "normal" way (`operator()` with three integer indices). It is intended to stress the abstraction overhead (or lack thereof) in the implementation, since `subspan` is the most complex part of the `mdspan` implementation from a C++ perspective. Note that this is not the intended use case of the `subspan` function, though it serves as a reasonable worst-case proxy. The relevant portion of the source code for this benchmark, for an input `mdspan` named `s` and an output named `sum`, looks like this:

```
for(ptrdiff_t i = 0; i < s.extent(0); ++i) {
  auto sub_i = subspan(s, i, all, all);
  for (ptrdiff_t j = 0; j < s.extent(1); ++j) {
    auto sub_i_j = subspan(sub_i, j, all);
    for (ptrdiff_t k = 0; k < s.extent(2); ++k) {
      sum += sub_i_j(k);
    }
  }
}
```

`MatVec` benchmark: The `MatVec` benchmark performs a simple dense matrix-vector multiply operation. It is aimed at demonstrating the impact of layout choice on performance, particularly in the context of performance portability of parallelization across diverse hardware platforms. Consider this serial implementation:

```
for(ptrdiff_t i = 0; i < A.extent(0); ++i) {
  for(ptrdiff_t j = 0; j < A.extent(1); ++j) {
    y(i) += A(i,j) * x(j);
  }
}
```

When parallelizing the outer loop via OpenMP, C++17 standard parallel algorithms, or CUDA, the optimal layout depends on the hardware. On CPUs, the compiler will vectorize the inner loop over `j`; thus, unit-stride access on the second dimension of `A` is optimal. On GPUs, no implicit auto-parallelization happens, so unit-stride access on the first dimension is optimal. Being able to make this layout change in the type of `A`, without actually changing the algorithm, means that the algorithm can be generic over different architectures.

Results: Compiler Comparison

Figure 4 shows a comparison of `mdspan` overheads relative to the raw pointer analog for serial versions of several benchmarks. With the exception of the `TinyMatrixSum` benchmark using dynamic extents, overheads on all of the benchmarks were either completely or very nearly within the experimental noise. The outlier in this regard, `TinyMatrixSum` with dynamic extents, is an interesting case study in the brittleness of modern loop optimizers, whether or not C++ abstraction is involved. To a first approximation, the authors believe the explanation for this is as follows: if the compiler heuristic guesses that the inner loop sizes are too large, the resulting optimization decisions (such as the amount of unrolling) are inefficient for a 3x3 matrix. How the use of `mdspan` interacts with the compiler's heuristic for generating this guess varies from compiler to compiler. For instance, with the latest version of Clang, the optimizer actually happens to make a *better* guess, leading to a "negative overhead."

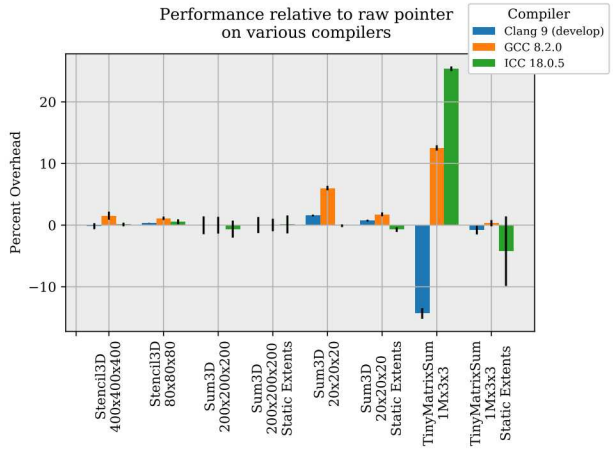

Fig. 4. Comparison of overheads, relative to raw pointer implementations, of the serial versions of various benchmarks across different compilers.

In many ways, the optimizer brittleness in this single outlier presents a strong argument for the sort of genericity that `mdspan` provides. As C++ continues to evolve, more compiler-specific extensions that let programmers give hints to guide compiler optimization are likely to trickle in. Maintaining such hints inside the logic of application code is often impractical or impossible, but incorporating that information into the `mdspan` accessor (particularly if such accessors can be vendor-provided), over which most algorithms can be generic, is a completely reasonable proposition in many cases.

Results: Effect of Static Extents

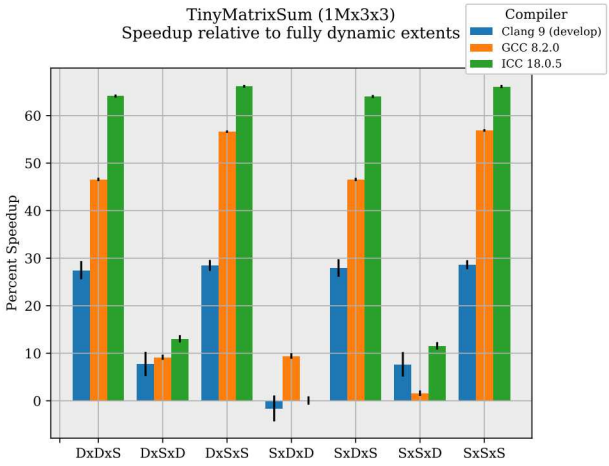

Fig. 5. Comparison of speedups, relative to the fully dynamic version, of the TinyMatrixSum benchmark. "D" indicates a dynamic expression of the particular extent, while "S" indicates a static expression (for instance, "DxDxS" indicates that the first extent, 1 million, was expressed dynamically, the second extent, 3, was expressed dynamically, and the third extent, 3, was expressed statically).

Figure 5 shows the speedup achieved when using static extents for the two inner dimensions as opposed to dynamic extents. When programmers provide them as static extents, the compiler is able to unroll the inner loops fully, resulting in nearly two times better performance on the test system Mutrino. The effect of static extents on the compiler's ability to optimize can vary significantly from compiler to compiler based on design decisions internal to the compiler's implementation.

Results: Effect of Layout Abstraction

The benchmark in Figure 6 was run on the ARM ThunderX2 (test system Astra), Intel SkyLake (test system Blake), and NVIDIA TitanV (test system Apollo) platforms using OpenMP parallelization for the CPUs and CUDA for the GPU. On the CPU systems the use of `layout_right` (for the matrix) provides the better performance, with `layout_left` being 3x-7x slower. On the GPU, however, the `layout_left` version achieves a 10x higher throughput. The results shown represent performance measured in terms of algorithmic memory throughput (that is, the count of memory accesses in the algorithm, divided by run time.)

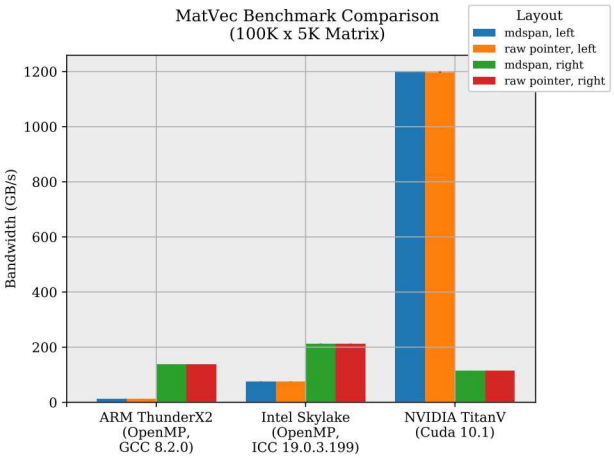

Fig. 6. Comparison of absolute memory bandwidths for the MatVec benchmark with different memory layouts.

Results: Overhead of `subspan`

For recent versions of GCC and Clang, the results are essentially identical to the raw pointer implementation of `Sum3D`, as shown in Figure 7. (There is no raw pointer implementation of `Subspan3D`, since the whole point is that it would be identical to `Sum3D`.) For ICC 18.0.5, the results showed significant overhead, rendering the GCC and Clang results invisible—as much as 400%. (The absolute magnitudes of the raw pointer timings were similar across all three compilers, so this is a genuine measurement of overhead introduced by the ICC frontend). Using the more recent ICC 19.0.3.199, we were able to obtain much more reasonable results in C++17 mode. Interestingly, though, the C++14 results *with the same compiler version* were much more similar to the ICC 18.0.5 results, indicating that the difference arises, at least in part, from more modern C++ abstractions being easier for modern compilers to understand. These results are shown in Figure 8.

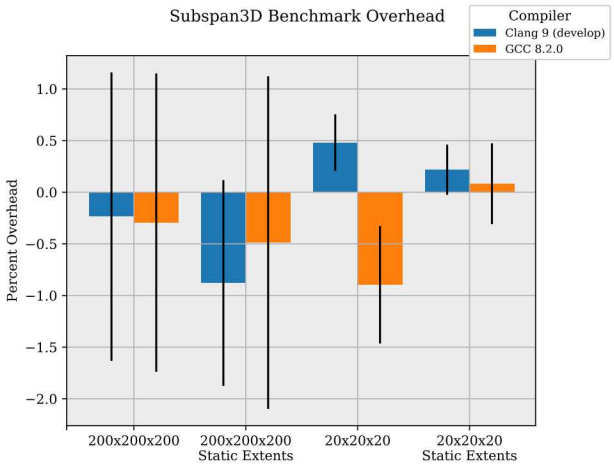

Fig. 7. Comparison of overheads, relative to raw pointer implementations, of the Subspan3D benchmark for GCC and Clang.

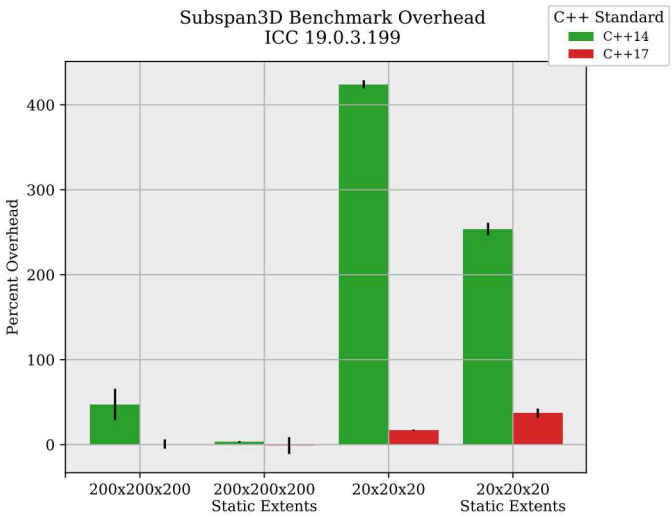

Fig. 8. Comparison of overheads, relative to raw pointer implementations, of the Subspan3D benchmark for ICC 19.0.3.199. Note that this compiler was not available on our primary testing machine, so the test system Blake was used for this benchmark.

Conclusions

We have presented both the ISO C++ design and a production-oriented implementation of `mdspan`. The `mdspan` data structure is based on the `View` class in the Kokkos C++ Performance Portable Programming Model. `mdspan` introduces a multi-dimensional array view abstraction into the C++ Standard. The class' layout and accessor abstractions address performance portability concerns. Besides controlling memory access patterns and data access semantics, the abstractions also open the door for incorporating heterogeneous memory paradigms via strong typing. Using a number of microbenchmarks, we have demonstrated that our implementation of `mdspan` has (in most cases) negligible overhead, compared to using raw pointers to represent multi-dimensional arrays. The implementation can be used with a C++11 standard-compliant compiler, and thus can be used with currently available toolchains on typical supercomputing systems. The standardization of `mdspan` lays the foundation for further efforts, such as standardized linear algebra, [9] which can help to address future performance portability needs of HPC and heterogeneous computing use cases.

Acknowledgments

This work was carried out in part at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidary of Honeywell International Inc., for the U. S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] J. R. Neely, "Doe centers of excellence performance portability meeting."

[2] Boyle, Peter A., Clark, M.A., DeTar, Carleton, Lin, Meifeng, Rana, Verinder, and Vaquero Avilés-Casco, Alejandro, "Performance portability strategies for grid c++ expression templates," *EPJ Web Conf.*, vol. 175, p. 09006, 2018. [Online]. Available: https://doi.org/10.1051/epjconf/201817509006

[3] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, and M. Bussmann, "Alpaka – an abstraction library for parallel kernel acceleration," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 631–640.

[4] D. S. Medina, A. St-Cyr, and T. Warburton, "Occa: A unified approach to multi-threading languages," *arXiv preprint arXiv:1403.0968*, 2014.

[5] R. Hornung, H. Jones, J. Keasler, R. Neely, O. Pearce, S. Hammond, C. Trott, P. Lin, C. Vaughan, J. Cook, R. Hoekstra, B. Bergen, J. Payne, and G. Womeldorff, "Asc tri-lab co-design level 2 milestone report 2015."

[6] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[7] "Kokkos: C++ Performance Portability Programming Model," 2019, version 2.9.00. [Online]. Available: https://github.com/kokkos/kokkos

[8] "Apps using kokkos," https://github.com/kokkos/kokkos/issues/1950, 2019.

[9] Mark Hoemmen and David Hollman and Christian Trott and Daniel Sunderland and Nevin Liber and Siva Rajamanickam and Li-Ta Lo and Graham Lopez and Peter Caday and Sarah Knepper and Piotr Luszczek and Timothy Costa, "P1673: A Free Function Linear Algebra Interface Based on the BLAS," ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee, Tech. Rep., 2019, https://wg21.link/P1673.

[10] Guy Somberg and Guy Davidson and Timur Doumler, "P1386: A Standard Audio API for C++: Motivation, Scope, and Basic Design," ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee, Tech. Rep., 2019, https://wg21.link/P1386.

[11] H. Carter Edwards and Bryce Adelstein Lelbach and Daniel Sunderland and David Hollman and Christian Trott and Mauro Bianco and Ben Sander and Athanasios Iliopoulos and John Michopoulos and Mark Hoemmen, "P0009: `mdspan`: A Non-Owning Multidimensional Array Reference," ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee, Tech. Rep., 2019, https://wg21.link/P0009.

[12] ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee, Tech. Rep.

[13] cppreference.com documentation for `std::string_view`. https://en.cppreference.com/w/cpp/string/basic_string_view.

[14] ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee, Tech. Rep.

[15] cppreference.com documentation for `std::span`. https://en.cppreference.com/w/cpp/container/span.

[16] A. A. Stepanov and P. McJones, *Elements of Programming*. Addison-Wesley Professional, 2009.

[17] cppreference.com documentation for `Allocator`. https://en.cppreference.com/w/cpp/named_req/Allocator.

[18] cppreference.com documentation for `std::vector`. https://en.cppreference.com/w/cpp/container/vector.

[19] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 239–267, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/567806.567810

[20] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[21] W. Bosma, J. Cannon, and C. Playoust, "The Magma algebra system. I. The user language," *J. Symbolic Comput.*, vol. 24, no. 3-4, pp. 235–265, 1997, computational algebra and number theory (London, 1993). [Online]. Available: http://dx.doi.org/10.1006/jsco.1996.0125

[22] A. Sutton and B. Stroustrup, "Design of concept libraries for c++," in *International Conference on Software Language Engineering*. Springer, 2011, pp. 97–118.

[23] ISO/IEC JTC1/SC22/WG14, The C Standards Committee, Tech. Rep.

[24] Daniel Sunderland and H. Carter Edwards and Hans Boehm and Olivier Giroux and Mark Hoemmen and David Hollman and Bryce Adelstein Lelbach and Jens Maurer, "P0019: Atomic Ref," ISO/IEC JTC1/SC22/WG21, The C++ Standards Committee, Tech. Rep., 2018, https://wg21.link/P0019.

[25] Using the GNU Compiler Collection (GCC): Restrict Pointer Aliasing. https://gcc.gnu.org/onlinedocs/gcc/Restricted-Pointers.html#Restricted-Pointers.

[26] C++ in Visual Studio: ___restrict. https://docs.microsoft.com/en-us/cpp/cpp/extension-restrict?view=vs-2019.

[27] Finkel, Hal and Carruth, Chandler and Nelson, Clark and Vandevoode, Daveed, Tech. Rep.

[28] J. Boccara. (2016) Strong types for strong interfaces. https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/.

[29] Leijen, Daan and Meijer, Erik, "Domain Specific Embedded Compilers," https://www.usenix.org/legacy/events/dsl99/full_papers/leijen/leijen.pdf, Tech. Rep., 1999.

[30] "benchmark: A microbenchmark support library," 2019, github hash `7ee7286`. [Online]. Available: https://github.com/google/benchmark