Minimal Additions to the Array View Library for Performance and Interoperability

Document number:	N4222
Date:	2014-10-09
Project:	Programming Language C++, Library Evolution Working Group

Reply-to:	Rutger ter Borg, rutger@terborg.net, Jesse Perla, jesse.perla@ubc.ca

1 Introduction

This paper proposes a set of small additions and changes to the `array_view` library, initially proposed in N3851. The primary goal is to provide a minimal set of changes to fulfill the needs of the scientific computing community, and to future-proof the design to ensure further performance, semantic, and interoperability enhancements are possible. Summarizing the suggested changes:

- Support for column vs. row major style storage–through a template parameter, enabling future additions with more flexible storage control for contiguous data.
- Support for `operator()(...)` for access to enable easier porting of existing code, and provide future optimizations and slicing.
- Ensure that the semantics of `operator[]` slices and any iterators are independent of storage ordering. This feature could lead to the only major design consideration: merging the `array_view` and `strided_array_view` into a single `array_view` class.
- Access to internal pointers, such as the data in `strided_array_view` to enable creation of adapters and rough testing of aliasing.
- Support RandomAccessContainer and other range-supported concepts, especially for the 1-D specialization.

2 Motivation and Scope

The majority of computational libraries either implement their own classes for vectors, matrices, and multi-arrays, with varying degrees of efficiency and expressiveness, or give up and implement with raw C pointers. The creation of expression template libraries for in C++ has made the semantics close to mathematics and domain-specific languages like Matlab, but at the cost of even more fragmentation. The `array_view` library is a very good start on solving this mess by providing a standardized way to access and adapt these various formats relying on contiguous data.

In order to be accepted by the scientific community, `array_view` needs to (1) provide maximum performance–including the encapsulation of available static information, (2) enable easy adaptation of existing objects, and (3) provide a straightforward path to porting code and functions to use the objects in the library.

Rather than provide a list of "nice to haves", this paper is intended to suggest immediate changes that would be very hard to change later.

3 Impact on the Standard

All proposed changes would be entirely implemented within `array_view` and associated classes.

4 Design Decisions

For performance, most gains come from organizing data to ensure maximum cache locality, and ensuring that static information and constexpr operations are available wherever possible. This ties directly into how multi-dimensional data is stored vs. accessed, and is algorithm-dependent.

Keep in mind throughout this document the separation between the semantics of a "row", "column", etc. from how the data is actually stored in contiguous memory. A fundamental test of the library is that the user can write code for one storage ordering, and when they flip the storage ordering it generates the same results–albeit performance will be slower or faster depending on how well the ordering of data matches the ideal storage ordering for the algorithm to achieve cache locality.

4.1 Storage Ordering

The term row vs. column major, `http://en.wikipedia.org/wiki/Row-major_order`, is often used for multi-dimensional data even if it comes from matrices. For 2D data, only row or column is ordering is possible, while arbitrary storage ordering could be considered for more than 2 dimensions. While this document is written with the terms `row_major` vs. `column_major`, more general terminology such as `index_major` vs. `index_minor` or `big_index` vs. `little_index` are perfectly reasonable.

For interoperability and performance, a large number of scientific libraries are written with column-major data and nearly every major container used for numerical analysis has a storage option. This is especially true when calling high performance libraries written in Fortran, but also when interfacing with Matlab through MEX C/C++ extensions, Python (sometimes), OpenGL, and R. Several C++ libraries also use column-major directly, such as Trilinos.[1]

Independently from questions of interoperability, storage ordering is also a first-order concern for performance. Consider an algorithm that takes an inner product of an enormous matrix, A, with an enormous vector, x. Whether the direction of this inner product is $x \cdot A$ or $A \cdot x$ changes how you would want A stored in memory. With the 2nd variation, column-major ordering of the data ensures that SIMD parallelization and cache locality are possible, while pointer arithmetic with row-major ordering would jump all over the matrix and potentially perform orders of magnitude slower.

The primary design choices here are between dynamic vs. static ordering, and between row/ column major vs. arbitrary ordering. While some implementations such as `boost::multi_array` allows a runtime specification for the ordering, there do not seem to be essential use cases for this behavior that are common enough to warrant the complexity and lack of compile-time information. Beyond having no additional runtime storage, static ordering choices allow for a `static_assert` on the storage type to ensure that users don't pass the wrong data type (e.g. AMP doesn't need to support column-major). Moreover, tag dispatching enables optimized algorithms based on the particular ordering in a generic library.

While there are use cases for arbitrary storage ordering in physics and chemistry, these can be added later if the ordering is left fairly general as a template parameter.

Finally, it is essential that the storage ordering choice has no effect on the semantics of any of the operations. For example, if `myarray[1]` returns the first "row" of a 2 dimensional array, from the users perspective, they are given just given a view representing the "row". The storage order simply changes the internal strides and pointer-arithmetic. This would also apply to any iterators, and the `view.section(...)` function.

[1] For example, see a discussion with SerialDenseMatrix in `http://trilinos.sandia.gov/Trilinos10.12Tutorial.pdf`. The default Fortran-style ordering can be found in many libraries in the optimization and scientific computing community.

4.2 Operator ()

While C (and some C++ libraries) use `operator[]` for data access, the single argument isn't ideal for generic C++ code. And it seems unlikely that a breaking change to a variadic `operator[]` is going to happen in the near future.

Chaining together the template instantiations and lines of code for `a[i][j][k]` to eliminate any performance penalty in a release build is an impressive achievement, but is it necessary to just to avoid using the `a(i,j,k)` notation? One alternative to chaining is passing in a structure `a[index<3>(i,j,k)]`, and is occasionally useful though not very natural. On the other hand, if i, j, and/or k is a static index, then overloads of `a(i,j,k)` may provide significant performance increase using constexpr access, but this would be could be more difficult for the compiler (possible?) with the chained `operator[]` or by passing in an index. Finally, from experience with `boost::multi_array`, the template errors generated from chained `operator[]` or `[index<3>(i,j,k)]` style access are extremely difficult to decipher, and concepts-lite may not help very much as a failure could happen well down the instantiation chain. Consider the alternative with concepts-lite and the overload described in Section 5.2.

Because of these complexities, many other libraries have been built to provide access with the `operator()`, such as Eigen, boost::ublas, COIN-OR densematrix, and many others.[2] This is also important as one of the goals of `array_view` is to enable easy porting of code, and without `operator()`, a complete re-write of functions is required for anything using Eigen, ublas, etc., significantly increasing the cost of moving to this library. Of course, the same argument can be made for supporting `operator[]` to enable porting C-style code, which is why both should be supported.

With overloading, `operator()` also allows efficient creation of slicing without the useful but cumbersome `.section()` notation where you need to query the size of each dimension. For example, in something like matlab, one can write `myarray(2,:, 3:4)` meaning fix the 1st dimension to the 2nd index, take everything in the 2nd dimension, and only return the 3rd and 4th index of the 3rd dimension. This sort of behavior is done in `boost::multi_array` by passing in a complicated and error-prone `boost::indices` object to `operator[]`.[3] With variadic `operator()`, this code could be converted to something like `myarray(2, std::all, std::index_range(3,4))` complete with all sorts of concept checks. These advanced features of a variadic overload should not be implemented initially, but future-proofing is useful.

4.3 Other Features for Interoperability

In order to ensure that the `array_view` objects can be used by general libraries, a few small additions are necessary. One complete test is whether numeric bindings can be written using the structure of `boost::numeric_bindings`. This would allow immediate access to a variety of libraries such as LAPACK. See the example bindings in `https://svn.boost.org/svn/boost/sandbox/numeric_bindings/boost/numeric/bindings/` for other libraries such as Eigen and `boost::multi_array`.

In particular, some of the requirements become absolutely necessary for `strided_array` access with different storage orderings, and basic detection of aliasing.

In order to exploit existing standard library algorithms and code, support for `RandomAccessContainer` and other range-supported concepts should be added. These will be most useful for a single dimension as users will often take one-dimensional slices or directly create a 1D version.

[2] A notable exception is `boost::multi_array` which only uses the `[{i,j,k}]` or `[i][j][k]` access, but this is an older and experimental library made well before C++11 features were available.

[3] See `http://www.boost.org/doc/libs/1_55_0/libs/multi_array/doc/user.html#sec_views` for an example. As discussed above, the template errors here get ugly due to the monadic `operator[]`.

5 Technical Specifications

5.1 Storage Ordering

Add in a template parameter to the definitions for `array_view` and `strided_array_view`. As a change of the storage ordering would require a transposing of the physical data, the constructors and copy constructors for these would allow for those of the same storage ordering type. For example, writing out the template definition and one of the constructors,

```
struct column_major_order{}; //tag structs, see alternative names above
struct row_major_order{};

template <typename ValueType, int Rank = 1,
          typename StorageOrder = row_major_order>
class array_view{

template <typename ViewValueType, int ViewRank> //leaving off enable_if
array_view(const array_view<ViewValueType, ViewRank, StorageOrder>& rhs)
                  : Base{ static_cast<typename bounds_type::value_type>(rhs.size())
        {
        }

};

template <typename ValueType, int Rank = 1,
          typename StorageOrder = row_major_order>
class strided_array_view{...};
```

The slices from `std::array_view` would also need to generate an object with the same storage ordering as the parent, but with semantics independent of the ordering.[4]

The implementation of the ordering for pointer arithmetic follows the standard rules of access, where essentially the strides are reversed. This would require changes to the definition of the `.stride()` requirements for both `array_view` and `strided_array_view`.

Crucially, we believe that `operator[]` on an `array_view` may no longer return an `array_view` in general. If `myarray[2]` was called on row-major data, it can be represented as an `array_view`, though we think there is a strong reason to keep around the pointer to the original data and formalize an offset–as discussed in Section 5.3. But if the data is column-major, then `myarray[2]` on an `array_view` must return a `strided_array_view`. One possibility is to always return a `strided_array_view` since it nests an array view with standard strides for the row-major case.

This leads to the only major design change to consider: whether it makes sense to have `array_view` and `strided_array_view` as separate classes at all. If both of them need to have a `.data()` and a `.offset()` as described Section 5.3, then can't `array_view` just be when strides the same as the extents (or reversed, depending on storage order). Simple constructors could be made to simplify that. The

[4] In future versions of this library, arbitrary storage ordering would just require a static list of orders. For example, `struct storage_order:std::integer_list<Order...> {};`. This could be used like `array_view<double, 3, std::storage_order<1,3,2>>`.

When a fixed extent version of the view is added to the library, consider putting the storage order at the end of the argument list. i.e.,

```
template <typename ValueType, int Extents..., typename StorageOrder = row_major_order>
class fixed_array_view{...};
```

concern for implementors that this adds in additional storage of the list of strides of the class could be resolved by a partial specialization with some simple templates (e.g.

```
struct general_strides{};
struct contiguous_strides{};
template <typename ValueType, int Rank = 1,
          typename StorageOrder = row_major_order,
          typename StrideType = contiguous_strides>
class array_view {...
//no storage of strides if the StrideType = contiguous_strides specialization
};
%
```

We should point out that merging the two views is exactly how `boost::multi_array` did it, though they don't seem to take the next step to a specialization without storing strides, which is an implementation decision.

From a teaching perspective, none of the user interfaces for creating views or the constructors implemented in your documentation change with the merged class, and only a single class would need to be taught (getting around the confusion for people that a `strided_array` is mostly a superset of an `array_view`).

5.2 Operator ()

Add in a variadic operator, which is required to be of the same dimensionality as the `array_view`.

```
template <typename ValueType, int Rank = 1,
          typename StorageOrder = row_major_order>
class array_view{
...

template<typename... Indices>
constexpr operator()(Indices... indices) //Independent of storage order
//requires Rank = length(Indices)
//requires all indices convertible to the array_view::index_type
{ //Tag dispatch to an implementation based on the storage order
};
```

As discussed, the `operator()` notation can lead to powerful overloads with clear error messages due to concepts-light. Recall that concepts only apply to a single instantiation, rather than tying down a complicated chain of nested `operator[]` calls or a nested object passed into `operator[]`. None of these need to be implemented in the first version.

5.3 Other Features for Interoperability

In general, we believe that exposing all underlying data pointers, offsets, etc. is important for interoperability. This is not an inherently safe library, so always err on the side of less encapsulation and ensure maximum interoperability. Much better safety and encapsulation could be attained in the inevitable container for multi-arrays itself.

The most important feature is to ensure that the underlying pointers for `strided_array_view` can be accessed. i.e., an equivalent to the `.data()` of `array_view`. The `boost::multi_array` kept around both the offset and the origin for its version of `strided_array_view`, and believe it is necessary. When

alternative storage ordering is implemented in `strided_array_view`, we believe it will be necessary to have the underlying pointer to the data.

Note that the `.data()` of a `strided_array_view` is **not** the address of the first element you are accessing, especially with different storage orderings, so you cannot just dereference it as a trick. You see this with the `.section(...)` function as well, which has an offset. Without a `.data()`, it is also impossible to do a rough aliasing check in a library. Having a `.offset(...)` may also apply to the `array_view` itself, especially if an `array_view` is returned from certain operations.

For `RandomAccessContainer` support, an iterator type would need to be created to go through the views along the first semantic dimension (i.e., the equivalent of taking slices [1], [2], ...). As always, this means that whether the iterator goes through memory contiguously or strided depends on the storage ordering. Coupled with `begin()`, `end()`, and the existing slice notation, this would enable access to a large number of existing functions as well as the range based for. While many algorithms would recursively go through these iterators (i.e., multi-dimensional splines), it is reasonable to start with this only available for the one-dimensional case.

6 Acknowledgments

Thank you to all those providing comments on iso future proposals forum, and to Łukasz Mendakiewicz.

7 References

- For the referenced `array_view` proposal,
  - The original was `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3851.pdf`, with `https://isocpp.org/files/papers/N3976.html` and the latest changes in `http://isocpp.org/files/papers/N4087.html`
  - Further presentation on this material can be found on `https://github.com/CppCon/CppCon2014/tree/master/Presentations`
  - An experimental implementation of that specification is implemented in `http://parallelstl.codeplex.com/SourceControl/latest#include/experimental/impl/array_view.h` with a test in `http://parallelstl.codeplex.com/SourceControl/latest#Test/array_view.cpp`
- For the boost `numeric_bindings` as an example of an external interface, see `https://svn.boost.org/svn/boost/sandbox/numeric_bindings/boost/numeric/bindings/` and in particular, `https://svn.boost.org/svn/boost/sandbox/numeric_bindings/boost/numeric/bindings/boost/multi_array.hpp` and `https://svn.boost.org/svn/boost/sandbox/numeric_bindings/boost/numeric/bindings/eigen/matrix.hpp`