Revised 2/4/2015 9:31 AM

N4355 : Shared Multidimensional Array with Polymorphic Layout

Authors:
Carter Edwards hcedwar@sandia.gov
Christian Trott crtrott@sandia.gov

Related papers:
N4177 Multidimensional bounds, index and array_view, revision 4
N4222 Minimal Additions to the Array View Library for Performance and Interoperability
N4184 SIMD Types: The Vector Type & Operations
N4300 Issues with Array View
N4356 Relaxed Array Type Declarator : enabling proposal to Core

1 Scope and Motivation

This paper integrates concepts of std::shared_ptr (20.8.2.2), std::array_view (N4177), and polymorphic array layout into a unified interface. This paper also addresses a inconsistency between std::array<T,N> (23.3.2) and std::array_view<T,N> (N4177) where std::array's N is the one dimensional array length and the proposed std::array_view's N is the rank of a multidimensional array. This paper integrates the separate std::strided_array_view (N4177) template class as a particular specialization in the space of polymorphic multidimensional array layouts. This paper enables a mix of runtime and compile-time multidimensional array dimensions to allow compile-time optimization of multidimensional offset computations.

Multidimensional arrays are a foundational data structure for science and engineering codes, as demonstrated by their existence in FORTRAN for five decades. High performance computing (HPC) is an essential consideration for these codes; as such their multidimensional arrays are often defined and redefined to accommodate evolving computational hardware. For example, arrays were laid out for vectorization on early Cray supercomputers and then re-laid out for performance on architectures with large multilevel caches. The current “array of structures” versus “structure of arrays” versus “structure of arrays of structures” struggle is simply a new chapter in an old data layout story.

Traditionally changing the layout of data and associated memory access patterns required extensive refactoring of code. If more than one layout and access pattern is required for hardware-specific HPC optimizations then multiple versions of functionally identical code had to be maintained. A sub-discipline of HPC kernel auto-tuning evolved to automatically generate code for such variants and measure their performance on target architectures.

The goal of this proposal is to provide a multidimensional array class that has a polymorphic layout to (1) provide a standard multidimensional array capability that is suitable for science and engineering HPC and (2) provide a polymorphic layout such that the layout may be changed without refactoring code or supporting multiple version of functionally identical code.

2 Shared and Weak Array

Two classes are proposed, **std::experimental::shared_array** and **std::experimental::weak_array**. The **weak_array** class fulfills the role of the proposed **array_view** class in N4177.

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 ;

The **shared_array** and **weak_array** classes conform to the shared ownership semantics of **std::shared_ptr** and **std::weak_ptr**. A **weak_array** object may be constructed from a **shared_array** object with conformal template arguments.

An alternative naming option discussed by the authors is '**shared'** and '**weak'**. Motivation for this alternative is that the proposed functionality and interface is compatible and applicable for a rank zero array; i.e., a non-array type.

2.1 Array Type Template Parameter

The ArrayType template parameter is expected to be a multidimensional array type declaration of the following syntax. This syntax requires relaxation of the multidimensional array type bounds constraint as proposed in N4356.

$$\texttt{T[N0}_{\texttt{opt}}\texttt{][N1}_{\texttt{opt}}\texttt{][N2}_{\texttt{opt}}\texttt{]...}$$

Type **T** is a possibly cv-qualified type of array members, the count of dimension **[N#$_{opt}$]** expressions is the multidimensional array rank, and each dimension **N#$_{opt}$** is an optional integral constant expression denoting explicitly declared array dimensions. Each omission of an **N#$_{opt}$** expression denotes an implicitly specified dimension that must be declared to construct a valid **shared_array** object.

The array's Cartesian product multi-index domain space is [0..N0)x[0..N1)x[0..N2)x... .

2.2 Array Layout Mapping

The optional ArrayLayout parameter specifies what mapping is used from the multi-index domain space to the array's memory range space. The extent of this range space is [ptr .. ptr+N_{extent}). The cardinality of the multi-index domain space may be less than the extent of the array if the layout mapping pads array dimensions or when indexing into a subarray. Dimension padding may be introduced to align a dimension for simd vector instruction units or cache lines. The extent of the array may be less than the cardinality of the domain space when, for example, the layout map implements a symmetric tensor.

The default ArrayLayout multi-index space layout map is the traditional bijective C and C++ array layout mapping. For example the default multi-index space layout map computation for a rank-5 array is the following.

offset(i0,i1,i2,i3,i4) = i4 + N4 * (i3 + N3 * (i2 + N2 * (i1 + N1 * (i0))))

Array layout polymorphism is elaborated in Section 3.

2.3 Shape and Usage Observers of both *shared_array* and *weak_array*

Both **shared_array** and **weak_array** classes have the following observers.

`typedef` *`ArrayLayout`* **`layout_type ;`**

Type of the ArrayLayout template argument.

`layout_type layout() const noexcept ;`

Returns array layout, if layout_type is non-void.

`constexpr static int rank() noexcept ;`

Returns the rank of the array which is equal to **`std::rank<ArrayType>::value`**.

`std::array<SizeType,Rank> size() const noexcept ;`

Returns a std::array containing the explicitly and implicitly specified dimensions of the array.

`SizeType extent() const noexcept ;`

Returns the extent of memory in the range space.

`explicit operator bool() const noexcept ;`

Returns whether the referenced array is not empty.

`long use_count() const noexcept ;`

Returns a count of the number of **shared_array** objects that share ownership of the allocated array when the **use_count** function is called.

2.4 Additional Usage Observers of *weak_array*

`bool expired() const noexcept ;`

Returns use_count() == 0.

`shared_array< ArrayType, ArrayLayout, SizeType> lock() const noexcept;`

Returns a **shared_array** object for the multidimensional array **if** the array originated from a shared object and the use_count() != 0.

2.5 Member Access

Both **shared_array** and **weak_array** have the following member access operator.

```
template < typename... IntegralTypes >
bool contains( IntegralTypes... indices ) const ;
```

The parameter pack is a sequence of integral values with count equal to the rank of the array representing a multi-index within the array's Cartesian index space. The contains function determines whether the multi-index is contained within the Cartesian index space.

```
template < typename... IntegralTypes >
reference operator()( IntegralTypes... indices ) const ;
```

The parameter pack is a sequence of integral values with count equal to the rank of the array representing a multi-index within the array's Cartesian index space. The operator is a function that evaluates the array layout mapping and returns a reference to the associated member of the array. If the multi-index is not within the array's Cartesian index space the function has undefined behavior. The operator is const as it does not alter the **shared_array** or **weak_array** object.

```
template < typename IntegralType >
reference operator[]( IntegralType ) const ;
```

The operator[] is defined for an array of rank one.

2.6 Shared Array Allocation and Construction

Syntax for construction and allocation is challenging due to the need for ArrayType and ArrayLayout parameters as well as implicit dimensions. Members of the array are constructed with the Allocator::construct function.

```
template< class Allocator , typename... IntegralTypes >
explicit shared_array( const Allocator &
                     , IntegralTypes... implicit_dimensions );

template< class Allocator , typename... IntegralTypes >
explicit shared_array( const Allocator &
                     , const ArrayLayout &
                     , IntegralTypes... implicit_dimensions );
```

Array memory is allocated from the specified allocator. Implicit dimensions are input through the **implicit_dimensions** argument and merged with the explicitly defined dimensions to specify the complete Cartesian product index space. The ArrayLayout argument provides additional details that may be required for the array layout mapping. For example, a strided layout requires either a stride-ordering of the dimensions for a dense layout mapping or specific strides for potentially non-dense layout mapping.

```
~shared_array();
```

The destructor decrements the use_count. When use_count is zero members of the array are destructed with the Allocator::destroy function and allocated memory is released.

2.7 Construction / Referencing User Supplied Memory

A **weak_array** object may be constructed from user supplied memory. It is the user's responsibility to insure that the supplied memory meets or exceeds the required extent of allocated memory. The constructed **weak_array** object is non-empty; however, the **weak_array** object **use_count()** is zero as the memory did not originate with a **shared_array** object.

```
template< typename... IntegralTypes >
explicit weak_array( T * ptr , IntegralTypes... implicit_dimensions );

template< typename... IntegralTypes >
explicit weak_array( T * ptr
                   , const ArrayLayout &
                   , IntegralTypes... implicit_dimensions );
```

The extent of user memory **[ptr .. ptr+N$_{extent}$)** must satisfy the following condition.

assert(N$_{extent}$ <= weak_array<ArrayType,ArrayLayout>(ptr , ...).extent())

A **weak_array** object constructed from user supplied memory fulfills the role of the proposed **array_view** in paper N4177.

3 Array Layout Polymorphism

The **shared_array** and **weak_array** interfaces do not imply nor guarantee a particular multi-index space layout map. Thus the interface specification may be satisfied with a layout other than the traditional (default) right-to-left contiguous mapping of the Cartesian multi-index space.

3.1 Initial Defined Layout

Three initial defined layouts are proposed in addition to the default: **layout_right**, **layout_left**, and **layout_stride**. Polymorphic array layout is an extension point for this capability. For example, many dense linear algebra and finite difference computations use tiled array layouts to improve performance.

The **layout_right** multi-index space mapping specifies that the right-most index is stride-one and strides increase from right-to-left, as with the traditional C and C++ multidimensional array layout. The layout right departs from the traditional layout in that the strides may be padded to allow subarrays of members to be aligned in memory. Construction of a **shared_array** or **weak_array** with **layout_right** does not require an ArrayLayout argument, or may accept a default constructed **layout_right** argument.

The **layout_left** multi-index space mapping specifies that the left-most index is stride-one and strides increase from left-to-right, as with the traditional FORTRAN multidimensional array layout. The layout left departs from the traditional layout in that the strides may be padded to allow subarrays of members to be aligned in memory. For example, dense linear algebra libraries often have a "leading dimension of matrix A" that is equal to or less than the actual computational dimension of the left-most index. Construction of a **shared_array** or **weak_array** with **layout_left** does not require an ArrayLayout argument, or may accept a default constructed **layout_left** argument.

The **layout_stride** multi-index space mapping specifies that indices have arbitrary stride, while still guaranteeing that the mapping is injective to the extent of the array. A **weak_array** object instantiated with **layout_stride** fulfills the role of the proposed **strided_array_view** in paper N4177. Construction of a **shared_array** or **weak_array** with **layout_stride** without an ArrayLayout argument, or default constructed **layout_stride** argument, defaults to an unpadded right-to-left ordering of dimensions.

3.2 Layout Stride and Array Layout Extensibility

The array layout mapping type is an extension point to enable library or application centric layout specifications. The strided array layout mapping is proposed due to its pervasive need across domains. The strided array layout mapping also serves to define the functionality and interface requirements for user defined array layout mappings. For example, a common array layout mapping in high performance computing is tiling.

```
struct layout_stride {

  template< unsigned Rank , typename SizeType >
  struct mapping {

    constexpr static int rank() noexcept { return Rank ; }

    template< typename... IntegralTypes >
    mapping( const layout_stride & , IntegralTypes... dimensions );

    SizeType extent() const noexcept ;

    std::array<SizeType,Rank> size() const noexcept ;

    template< typename... IntegralTypes >
    bool contains( IntegralTypes... indices ) const noexcept ;

    template< typename... IntegralTypes >
    SizeType operator()( IntegralTypes... indices ) const ;
  };

  layout_stride();

  // Explicitly defined strides for each dimension
  template< typename... IntegralTypes >
  explicit layout_stride( IntegralTypes... strides );
};
```

Implementations of **shared_array** and **weak_array** are required to function *as if* they were implemented with the corresponding array layout mapping. As such a specialized **shared_array** or **weak_array** implementation may have better performance than explicitly using an array layout mapping.

ArrayLayout::mapping<Rank,SizeType>

The **mapping** nested template class implements *ArrayLayout*'s mapping from the Cartesian index domain space to integral extent range space. The functionality and interface for the nested mapping template class conforms to an array layout mapping concept. However, an *ArrayLayout* is likely to have a layout specific interface for non-trivial layouts.

```
template< typename... IntegralTypes >
ArrayLayout::mapping<Rank,SizeType>::
  mapping( const ArrayLayout & , IntegralTypes... dimensions );
```

Construct a mapping for the given rank, integral size type, array layout, and dimensions.

```
SizeType ArrayLayout::mapping<Rank,SizeType>::extent() const ;
```

Return the extent of the range space of the mapping. For the strided layout this will be the product of the largest stride and its corresponding dimension.

```
std::array<SizeType,Rank> ArrayLayout::mapping<Rank,SizeType>::size() const ;
```

Return dimensions of the Cartesian index domain space.

```
template< typename... IntegralTypes >
bool ArrayLayout::mapping<Rank,SizeType>::
  contains( IntegralTypes... indices ) const ;
```

Return whether the Cartesian index domain space contains the given multi-index.

```
template< typename... IntegralTypes >
SizeType ArrayLayout::mapping<Rank,SizeType>::
  operator()( IntegralTypes... indices ) const ;
```

Return the mapping of a Cartesian index domain multi-index to an integral value in the range space. If the multi-index is not contained in the domain space the result is undefined.

```
layout_stride();
```

An array layout mapping constructed from a default constructed strided layout is the standard C or C++ right-to-left array layout maping.

```
template< typename... IntegralTypes >
explicit layout_stride( IntegralTypes... strides );
```

Specify strides for each dimension.

4 Future Extension: Subarray

An important capability in scientific and engineering codes with multidimensional arrays is to view subarrays of an array. This capability is introduced here for completeness of the multidimensional array functionality vision and a specific proposal will appear in a subsequent paper. A draft of this capability is summarized by the following subarray function.

```
template< class ArrayType
        , class ArrayLayout
        , typename... Ranges_And_Indices >
shared_array< SubarrayType , SubarrayLayout >
subarray( const shared_array<ArrayType,ArrayLayout> & source
        , Ranges_And_Indices... );
```

The Ranges_And_Indices argument pack contains a sequence of integral type [begin,end) ranges and indices with a count equal to the rank of the source array. The rank of the returned subarray is equal to the count of ranges in the argument pack. The source array and returned subarray have a common use_count(); i.e., the memory associated with the array will not be released until the last shared array and shared subarray that references into that array are destroyed.

The *SubarrayType* and *SubarrayLayout* parameters are derived from the ArrayType, ArrayLayout, and Ranges_And_Indices argument pack. In practice the return value from **subarray** would be captured by an **auto** declared variable. A meta-function will be used to determine these derived types and thus the actual return type.

```
template< class ArrayType
        , class ArrayLayout
        , typename... Ranges_And_Indices >
struct subarray_traits {
  typedef /*...*/ type ;   // SubarrayType
  typedef /*...*/ layout ; // SubarrayLayout
};
```

This planned extension is introduced for completeness of the vision, and to influence candidate implementations to be designed in anticipation of this capability.