Kokkos Array Performance-Portable Manycore Programming Model

H. Carter Edwards
hcedwar.sandia.gov

Daniel Sunderland
dsunder.sandia.gov

Sandia National Laboratories *
P.O. Box 5800
Albuquerque, NM 87109

ABSTRACT

Large, complex scientific and engineering application code have a significant investment in computational kernels which implement their mathematical models. Porting these computational kernels to multicore-CPU and manycore-accelerator (*e.g.*, NVIDIA® GPU) devices is a major challenge given the diverse programming models, application programming interfaces (APIs), and performance requirements. The *Kokkos Array* programming model provides library-based approach for implementing computational kernels that are performance-portable to multicore-CPU and manycore-accelerator devices. This programming model is based upon three fundamental concepts: (1) manycore *compute devices* each with its own memory space, (2) data parallel computational kernels, and (3) multidimensional arrays. Performance-portability is achieved by decoupling computational kernels from device-specific data access performance requirements (*e.g.*, NVIDIA coalesced memory access) through an intuitive multidimensional array API. The Kokkos Array API uses C++ template meta-programming to, at compile time, transparently insert device-optimal data access maps into computational kernels. With this programming model computational kernels can be written once and, without modification, performance-portably compiled to multicore-CPU and manycore-accelerator devices.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

performance

Keywords

multicore, manycore, GPU, multidimensional array, mini-application

*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This paper is cross-referenced at Sandia National Laboratories as SAND2011-9311C.

PMAM'12 February 26, 2012, New Orleans, LA, USA

1. INTRODUCTION

Multicore-CPU and manycore-accelerator devices promise improvements in both runtime performance and energy consumption. Porting large, complex scientific and engineering applications to these devices is a significant challenge in that (1) a multi-level parallel programming model is required to manage both distributed and on-device parallelism *and* (2) these devices have critical and device-specific performance considerations. For example, kernels executing on NVIDIA devices must use coalesced memory access and kernels executing on non-uniform memory access (NUMA) devices must manage thread/memory placement.

Many projects have successfully addressed these challenges by writing distinct versions of their codes that are specialized for particular compute devices (*e.g.*, see the "cudazone" [8]). However, this approach incurs the cost of developing, verifying, and maintaining a special version of the code for each class of compute device. For large complex applications this can be an unacceptable cost.

Programming Model. The Kokkos Array programming model uses a library-based (versus compiler-based) approach to support implementation of computational kernels that are performance-portable to multicore-CPU and manycore-accelerator devices. The programming model is similar to the Thrust library [10] in that it provides parallel_for and parallel_reduce operations, manages allocation and deallocation of data on the manycore device, and uses standard C++ as opposed to a new language or pragma extensions. Kokkos Array is unique in that it provides an intuitive multidimensional array API for applications to aggregate data, and it allows multiple devices to be used within a single application. In the Thrust programming model arrays are strictly one-dimensional and only a single, non-host device may be used within the application.

Performance Portability. Performance-portability includes source code portability of a kernel's code *and* performance that is commensurate with a device-specific implementation of that kernel. Memory access has become a dominant performance consideration for many kernels; *e.g.*,

coalesced memory access on NVIDIA and thread/memory placement on NUMA devices. The Kokkos Array API defines a separation of concerns between an application's need to aggregate data and a device's specific data access performance requirements. The Kokkos Array implementation uses C++ template meta-programming [1] to, at compile-time, transparently insert device-optimal data access maps into the application's multidimensional arrays. Thus computational kernels can be written once in the standard C++ language and, without modification, performance-portably compiled to multicore-CPU and manycore-accelerator devices.

Trilinos. The Kokkos Array library is available through Sandia National Laboratories' Trilinos[11] project. Available compute devices use CUDA v4 [8] and pthreads [6] with NUMA locality control via the hardware locality (HWLOC) library [9]. A collection of performance tests and mini-applications are used to investigate the performance and usability of the programming model, API, and implementations.

2. PROGRAMMING MODEL

The Kokkos Array programming model is based upon three fundamental concepts: (1) manycore *compute device* which has memory separate from the *host* main memory, (2) multidimensional arrays of data mapped into the memory of a compute device, and (3) data parallel computational kernels applied to these multidimensional arrays.

2.1 Multidimensional Array

Multidimensional arrays are historically intrinsic to scientific and engineering application codes, and intrinsic to languages commonly used by these codes. For example, the following two statements declare the same double precision multidimensional array `X` in the FORTRAN and C languages' syntax.

```
REAL*8   X(1000,24,8,3)       ; FORTRAN
double   X[3][8][24][1000];   // C
```

These two declarations specify the type of the array's data (double precision) and the array's dimensions. The semantics of the FORTRAN and C languages define the mapping of the array into memory; *i.e.*, the striding of the array indices. Neither the declarations nor the language semantics specify (1) into which memory space the array is mapped and (2) how the array would be accessed by parallel computations.

Definitions. A *multidimensional array* is a homogeneous collection of data members which are identified by multi-indices from a multi-index space and which reside in the memory of a computational device. A *multi-index* is simply an ordered list of integers denoted by $(i_0, i_1, i_2, \cdots)$. The *rank* of a multi-index is the number of indices; *e.g.*, $(1, 3, 5)$ and $(7, 5, 3, 1)$ are rank-3 and rank-4 multi-indices. A Kokkos Array *multi-index space* is a Cartesian project of integer ranges $[0..N_0) \times [0..N_1) \times [0..N_2) \cdots$, denoted by $(N_0, N_1, N_2, \cdots)$. The *cardinality* (or size) of a multi-index space is the product of its dimensions, $N_0 * N_1 * N_2 * \cdots$.

Multidimensional Array Map. A multidimensional array maps its multi-index space to its data members with a one-to-one mapping. Traditionally, these data members reside in a contiguous span of memory on a compute device. The map for such a multidimensional array X can be expressed by a base location in memory and a bijective function between the multi-index space and an offset. For example, FORTRAN and C multidimensional array index spaces and offset maps are as follows.

FORTRAN multi-index space and offset map:
space: $[1 .. N_0] \times [1 .. N_1] \times [1 .. N_2] \times \cdots$
offset: $(i_0 - 1) + N_0 * ((i_1 - 1) + N_1 * ((i_2 - 1) + N_2 * \cdots))$

C multi-index space and offset map:
space: $[0 .. N_0) \times [0 .. N_1) \times [0 .. N_2) \times \cdots$
offset: $((((i_0) * N_1 + i_1) * N_2 + i_2) * \cdots)$

Key Concepts. (1) There are many valid multidimensional array maps. (2) Memory access patterns are defined by the map. (3) Different devices require different memory access patterns for optimal performance. For example, computations on an NVIDIA device must use a *coalesced* global memory access pattern. (4) Computational kernels and multidimensional array maps can be decoupled. (5) An optimal map for a given device can be inserted into a computational kernel at compile-time through C++ template meta-programming. For example, the map inserted for an NVIDIA device provides the performance-critical coalesced global memory access to multidimensional array data. Compile-time in-line insertion is critical for optimization of the heavily used multi-index mapping.

2.2 Data Parallel Operations

In data parallel computations multidimensional arrays are partitioned among the threads of a manycore device, and each thread applies one or more computational kernels to its designated subset of these arrays. Currently, Kokkos Array partitions a multidimensional array along exactly one dimension of the multi-index space. The left-most dimension was chosen for parallel partitioning by a consensus of computational kernel developers participating in a Kokkos Array software design review.

Parallel Partitioning. Parallel partitioning identifies N_P "atomic" units of parallel work for an array dimensioned $(N_P, N_1, N_2, \cdots)$. A thread calls a computational kernel to perform the i_P unit of work, where $i_P \in [0..N_P)$. To avoid thread-parallel race conditions and inter-thread locking a kernel must: *only* update array data members that are associated with that index $(i_P, *, *, \cdots)$ and *not* query array data members that are potentially updated by another thread applying the same kernel to a different unit of work.

Parallel Operations. Kokkos Array applies computational kernels through parallel_for and parallel_reduce operations. A parallel_for operation is trivially parallel in that the computational kernel's work is fully disjoint. In a parallel_reduce each application of the computational kernel generates data that must be reduced among all work items; *e.g.*, an inner product kernel generates N_P values which must be summed to a single value.

A **parallel_for kernel** is a function that inputs a collection of kernel-defined parameters and partitioned arrays, and updates a collection of partitioned arrays. A parallel_for function f is formally defined as follows.

$$f : (\{\alpha\}, \{X\}) \rightarrow \{Y\} \begin{cases} \{\alpha\} \equiv \text{input parameters} \\ \{X\} \equiv \text{input arrays} \\ \{Y\} \equiv \text{output arrays} \end{cases}$$

A **parallel_reduce kernel** is also a function that inputs a collection of parameters and partitioned arrays, and updates a collection of partitioned arrays. However, the func-

tion f also outputs an common (not partitioned) collection of parameters. Each application of the function to a unit of parallel work generates a contribution to these output parameters. Generated contributions are reduced by a *mathematically* commutative and associative reduction function f_Θ. An implementation f_Θ may be non-associative due to round-off in its floating point operations. A parallel_reduce function f and its associated reduction function f_Θ are formally defined as follows.

$$f : (\{\alpha\}, \{X\}) \rightarrow (\{\beta\}, \{Y\}) \begin{cases} \{\alpha\} \equiv \text{input parameters} \\ \{X\} \equiv \text{input arrays} \\ \{\beta\} \equiv \text{output parameters} \\ \{Y\} \equiv \text{output arrays} \end{cases}$$

$$f(\{\alpha\}, \{X(i_P, \cdots)\}) \rightarrow (\{\beta[i_P]\}, \{Y(i_P, \cdots)\}) \ \forall\, i_P$$
and then
$$f_\Theta(\{\beta[i_P] \ \forall\, i_P\}) \rightarrow \{\beta\}$$

2.3 Manycore Device

A manycore device owns memory which is separate from the host main memory and supports many concurrent threads of execution which share this memory. This conceptual model may reflect a physical separation of memory of the compute device (*e.g.*, NVIDIA GPU) or merely be a logical view of the same physical memory (*e.g.*, multicore CPU). Computations performed by the device only access and update data which are in the device's memory. As such data residing in host memory must be copied to the device before a computation can be performed by the device on that data.

A device implements parallel_for and parallel_reduce operations to call a kernel NP times from the device's concurrent threads. If the device has NP threads then all calls may be concurrent; otherwise a thread will call the kernel multiple times until the NP required calls are completed.

Heterogeneous Parallelism. An application may use both distributed memory and thread parallelism. Distributed memory parallelism, as implemented with the Message Passing Interface (MPI), defines multiple processes each with their own MPI rank and memory space. Kokkos Array assumes that each of these distributed memory parallel processes will use at most one device. This assumption is made to avoid introducing complexity associated with managing multiple devices within the same process. However, the abstraction for a single manycore device could aggregate multiple hardware devices into a single, logical device.

NUMA Thread Pool. The Kokkos Array manycore-CPU device is implemented using the *thread pool* strategy where a pool of threads (*e.g.*, pthreads [6]) is spawned once and work is dispatched to these threads. The HWLOC library is used to detect the number of NUMA nodes and the number of cores in each NUMA nodes. One thread is spawned per core, less the core already in use by the main thread, and the threads are explicitly pinned to NUMA nodes. Threads with adjacent ranks are assigned to work on adjacent partitions of an array as illustrated in Figure 1. These partitions are assigned to threads, and thus NUMA regions, through the NUMA "first touch" operation. By pinning adjacent rank threads to NUMA nodes the associated adjacent partitions of an array are also assigned to the same NUMA region, which reduces the number of inter-NUMA region boundaries in the array.

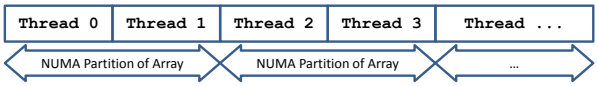


Figure 1: Explicit assignment of adjacent rank threads to adjacent partitions of an array, resulting in adjacent partitions being assigned to the same NUMA region.

3. KOKKOS ARRAY API

Kokkos Array defines C++ classes for general multidimensional arrays, multi-vectors, and values. A multi-vector is a restricted form of multidimensional array – a collection of one dimensional vectors all of the same length. A multi-vector is a simpler abstraction: it is at most rank-two (vector length and vector count) and has a single multi-index space mapping. A value is used to create and manage persistent parameters on the device which are typically shared by all calls to a computational kernel. For example, the $\{\beta\}$ parameters of a parallel_reduce kernel defined in Section 2.2 can be maintained in the device's memory as a value object. For brevity details of the simpler multi-vector and value APIs are not included here.

3.1 Index Space and Data Access

The first portion of the multidimensional array API given in Figure 2 identifies the data type of the members, device in which the data members are allocated, rank and dimensions of the multi-index space, cardinality of the array, and mapping from a multi-index to a data member. The data type (given by `value_type` in Figure 2) is restricted to be a simple intrinsic numerical type. This restriction is imposed so that data members can be simply and optimally mapped onto compute devices with performance-sensitive memory access patterns, such as NVIDIA GPU.

```
namespace Kokkos {
template < typename ValueType , class DeviceType >
class MDArray {
public:
  typedef ValueType  value_type ;
  typedef DeviceType device_type ;
  typedef   ...      size_type ;

  size_type rank() const ;
  size_type dimension( irank ) const ;
  size_type size() const ; // Cardinality

  // Map multi-index to associated data member
  value_type & operator()( iP , i1 , ... ) const ;
};
}
```

Figure 2: Multidimensional array API identifying the multi-index space and accessing data members via multi-index mapping.

The heavily use and performance-critical `operator()` maps multi-indices to data members. As such a device-optimal and in-lined implementation of this function is critical. A device-optimal implementation is selected through the `DeviceType` template parameter, which identifies a particular compute device. This device is specified at compile-time via a template parameter (as opposed to a runtime ob-

ject) so that the device-optimal multi-index mapping can be in-lined by the compiler. This API design avoids potential run-time overhead associated with C++ virtual functions or other run-time polymorphism strategies.

3.2 Shared Ownership Semantics

A given `MDArray` object is a *view* to array data, it does **not** exclusively own that data. All `MDArray` objects that view the same array data equally **share ownership** of that array data. The `MDArrayView` constructors, assignment operator, destructor, and allocation function given in Figure 3 implement shared ownership semantics.

```
namespace Kokkos {
template < typename ValueType , class DeviceType >
class MDArray {
public:
  MDArray();  // A NULL view.
  // New view of the same data viewed by RHS:
  MDArray( const MDArray & RHS );
  // Clear this view: if this is the last view
  // to an array then deallocate the array:
  ~MDArray();
  // Clear this view and then assign it to
  // view the same data viewed by RHS:
  MDArray & operator = ( const MDArray & RHS );
  // Query if 'this' is a non-NULL view:
  operator bool() const ;
  // Query if a view to the same data
  bool operator == ( const MDArray & RHS ) const ;
};

// Allocate data on the device and return a
// multidimensional array view to that data
template< class MDArrayType >
MDArrayType create_mdarray( nP , n1 , n2, ... );
}
```

Figure 3: Kokkos Array API for shared ownership and allocation semantics.

The `MDArray`copy constructor and assignment operator perform a *shallow copy* – they set the current `MDArray` object to be a view of same data viewed by the input object (`RHS` in Figure 3). A shallow copy only copies the minimal information required to view and access the array data, the data itself is **not** copied. In contrast the copy constructor and assignment operator of a *container* would perform a *deep copy* – they allocate their own array data as needed and then copy each data member from the input container.

Views are not Containers. Shared ownership semantics are fundamentally different from *container* semantics. A container has exclusive ownership of its data, versus a view which shares ownership of data with other views. The standard C++ container classes [5] implement container semantics where copy constructors and assignment operators perform a deep copy of all data owned by the container. The recent C++ shared pointer [2] class implements shared ownership semantics where multiple objects view the same data, and the last view to be destroyed is responsible for deallocating the viewed data.

Why View Semantics. In large complex applications arrays are allocated on the device by "driver" functions, passed among driver functions, passed from driver functions to computational kernels, passed from one computational kernel to another, and at some point should be deallocated to reclaim memory on the device. Managing the complexity of numerous references to many allocated arrays requires a high degree of software design and implementation discipline to prevent memory management errors of (1) deallocation of a still used array or (2) neglecting to deallocate an array no longer in use. Thus there is a significant risk that a team of application developers will lose track of when to, or not to, deallocate an array, and as a result will introduce one of the two memory management errors. This risk is mitigated by using view or *shared ownership* semantics for allocated Kokkos arrays. Under the shared ownership semantics multiple view to the same allocated data may exist and the last view to be *cleared* (see Figure 3) deallocates the allocated data.

Only Views. The Kokkos Array public API only provides views to array data – a container interface is intentionally omitted. This design decision simplifies the interface by providing a single, simple, and safe interface to allocated array data.

3.3 Copying Data

The data viewed by an `MDArray` object resides in the memory space of the device specified by the `DeviceType` template argument and has a device-specific mapping from the index space into that memory. This memory space may be separate from the host's memory space (*e.g.*, an NVIDIA device) or may be the same memory space (*e.g.*, a pthread device). Thus array data may, or may not, be directly accessible to code executing on the host process, and a *deep copy* of data between memory spaces may be required for the host to access data. The mirroring and deep-copying array data is given in Figure 4.

```
namespace Kokkos {
template < typename ValueType , class DeviceType >
class MDArray {
public:
  // Compatible MDArray in the host memory
  typedef ... HostMirror ;
};

// Create a compatible array in the host memory
template< MDArrayType >
typename MDArrayType::HostMirror
create_mirror( const MDArrayType & );

// Deep copy data between arrays
// with compatible type and dimension
template< typename ValueType , class DeviceDest ,
                               class DeviceSource >
void deep_copy(
  const MDArray<ValueType,DeviceDest>   & dest ,
  const MDArray<ValueType,DeviceSource> & source);
}
```

Figure 4: Kokkos Array API for deep-copying and mirroring data between host and device memory.

The `MDArray::HostMirror` type defines a multidimensional array type that (1) has data in the host memory space and (2) maps array data according to the original `MDArray` index space to data mapping. Thus the type defines an exact "mirror" of the device-resident multidimensional array data. Such a mirror allows direct, memory-to-memory, deep copy of data without requiring a potentially time consuming remapping of data between the device's optimal multi-index map and the host's optimal multi-index

map. The create_mirror function simply creates a multidimensional array of the HostMirror type with the same dimensions as the input array.

The deep_copy function copies data between two compatible arrays. For the deep copy function "compatible" is loosely defined as either (1) having the same type (same device and same map), (2) having different devices and the same map, or (3) using the "host" device with different maps.

An example of using the mirror and deep copy capabilities are given in Figure 5.

```
typedef MDArray< double, Device > array_type ;

array_type x = create_mdarray< array_type >( nP , nX );
array_type y = create_mdarray< array_type >( nP , nY );

array_type::HostMirror xh = create_mirror( x );
array_type::HostMirror yh = create_mirror( y );

// read data into 'xh' on the host process
deep_copy( x , xh );

// perform computations on the device
// inputting 'x' and outputting 'y'
parallel_for( nP , SomeFunction( x , y ) );

deep_copy( yh , y );
// write data from 'yh' on the host process
```

Figure 5: Example of creating mirrors and deep copying array data between host and device memory.

If MDArray::HostMirror and MDArray are the same type it is not necessary to create a mirror and deep copy to and from that mirror. In this special case, the create_mirror function can be instructed with a compile-time option to simply return a new view of the input array as opposed to allocating a compatible array. When the deep_copy function is given two views to the same data a deep copy of that data is unnecessary, and the function immediately returns. This allows, via a compile-time option, unnecessary data allocations and deep copy operations to be omitted from the code. In Figure 5 "xh" and "yh" would become views of the original "x" and "y", and the deep copy functions would discover that the arguments are views of the same data and return immediately.

4. COMPUTATIONAL KERNEL

A computational kernel is implemented as a **functor** for execution by a parallel_for or parallel_reduce operation. A functor is a C++ class that composes the computation, its parameters, and views to data to which the computation is applied (recall Section 2.2). Functor semantics are common to several programming models; *e.g.*, the C++ Standard Template Library (STL) algorithms [5], Intel Threading Building Blocks [7], and Thrust [10]. A functor is created on the host process, copied to the compute device, and then run thread-parallel on the compute device.

4.1 Parallel For Functor Interface

Interface requirements for a parallel_for functor are summarized in Figure 6. For performance-portability to different manycore devices a functor must

```
template< class Device /* REQUIRED template parameter */>
class MyParallelForFunctor {
public:
  typedef Device device_type ;      // REQUIRED typedef

  KOKKOS_MACRO_DEVICE_FUNCTION      // REQUIRED qualifier
  void operator()( int iP ) const; // REQUIRED operator

  // Functor members include input parameters,
  // views to input arrays, and views to output arrays.

  // Constructor typically copies input parameters
  // and shallow-copies input/output views from
  // constructor arguments into members of this class.
  MyParalleForFunctor( ... );
};
// Construct, copy to device memory, and
// call this functor nP times on the device
parallel_for( nP , MyParallelForFunctor(...) );
```

Figure 6: Interface requirements for a parallel_for functor.

- have the device as a template parameter and declare the devices via typedef device_type,
- declare all array view class-members using that device parameter,
- access array data through the array API, and
- implement the operator() with a simple subset of C++ (*i.e.*, the CUDA v4 subset).

A functor's operator()(iP) function is called nP times, where nP is the value passed to the parallel_for operator. Each call to the functor is passed a unique index iP in the range [0..nP) which the functor must use to access array data as per Section 2.2.

4.2 Fundamental Performance Considerations

Fundamental performance tests were applied in the early stages of Kokkos Array development [3], including comparison to hand-coded CUDA kernels. Results from these tests led to the following fundamental performance considerations.

Minimize global memory reads and writes. A global memory read or write of array data must (1) map a multi-index to a global memory location and (2) fetch data from or push data to that location. As such data that is used more than once in call to the functor's operator()(iP) should be read from global memory into a local variable which is typically cache-resident. Similarly computations should update local variables and then write these variables once to global memory. These techniques reduce use of, and multi-thread contention for, off-chip global memory access bandwidth.

Compile-time knowledge of dimensions. The mapping of a multi-index to a global memory location can be performed, in part, at compile-time **if** the dimensions of the index space are known at compile-time. However, the current MDArray API has runtime, not compile-time, knowledge of the index space. As such an enhancement of the MDArray API is planned to allow compile-time declaration of dimensions as follows.

```
template < typename ValueType , class Device ,
           unsigned N1 , unsigned N2 , unsigned N3 , ... >
class MDArray ;
```

In this proposed API only the parallel work dimension is declared at runtime, and all other dimensions declared at compile-time. Such an enhancement will allow a compiler to pre-compute the multi-index mapping from the template-specified dimensions.

Overlap global memory access and computations. Contention for access to global memory can be further reduced by overlapping accesses to global memory and computations among concurrent threads of execution. A computational kernel may be able to facilitate this overlap *if* each unit of work (1) accesses a relatively large amount of global memory and (2) has a relatively large computational intensity (ratio of operations to global memory accesses). This concept is illustrated by the left and right execution profiles in Figure 7. In the left execution profile every thread performs all of its global memory reads "up front." During this read-phase threads which share access to global memory are in contention and their global memory accesses can become serialized. In the right execution profile global memory reads are dispersed throughout the computation. This reduces contention and allows improved overlapping of global memory access and computations among threads.

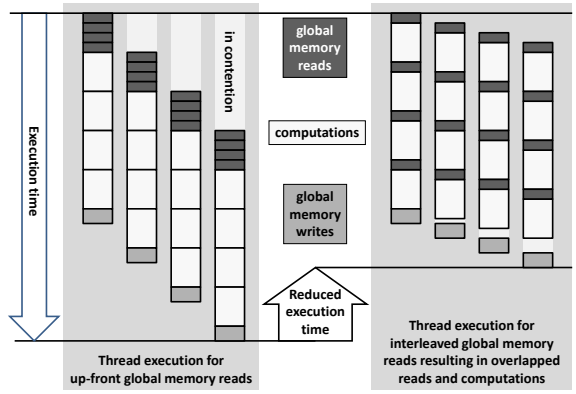


Figure 7: Conceptualization of overlapping global memory access and computations for threads sharing access to global memory.

NUMA Thread / Memory Management. For multisocket and multicore NUMA devices a core and the thread executing on that core will be associated with a particular memory controller and corresponding partition of global memory. That thread will have more direct, and thus faster, access to global memory within its corresponding partition as compared to memory outside of its partition. Furthermore, when threads access global memory outside of their designated partition they utilize both their own memory controller as well as the memory controller corresponding to the accessed memory. Such "out of partition" memory accesses increase multi-threaded contention for memory bandwidth and can further impact performance.

A Kokkos Array manycore-CPU device and multi-index mapping has been implemented to manage correlated placement of threads and array data to cores and global memory. During parallel execution each thread is assigned units of work with array data residing in that thread's corresponding partition of global memory. This correlated placement is intended to reduce cross-NUMA region memory traffic and associated demands on the memory subsystem.

4.3 Parallel Reduce Functor Interface

A parallel_reduce functor implements both the parallel kernel f *and* reduction function f_Θ defined in Section 2.2. Interface requirements for a parallel_reduce functor are summarized in Figure 8.

```
template< class Device /* REQUIRED template parameter */>
class MyReduceFunctor {
public:
  typedef Device device_type ; // REQUIRED typedef
  typedef  ...   value_type ;  // REQUIRED typedef
  // REQUIRED qualified operator and functions:
  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int iP , value_type & update ) const ;
  KOKKOS_MACRO_DEVICE_FUNCTION
  static void join( volatile       value_type & update ,
                    volatile const value_type & input );
    // example summation 'join': { update += input ; }
  KOKKOS_MACRO_DEVICE_FUNCTION
  static void init( value_type & output );
    // example summation 'init': { output = 0 ; }
  MyReduceFunctor( ... );
};
// Call this functor nP times on the device.
// Return the reduction result.
template< class ReductionFunctor >
typename ReductionFunctor::value_type
parallel_reduce( size_t NP ,
   const ReductionFunctor & functor );
// Output the reduction result.
template< class ReductionFunctor >
void parallel_reduce( size_t NP ,
  const ReductionFunctor & functor ,
  typename ReductionFunctor::value_type & result );
```

Figure 8: Interface requirements for a parallel_reduce functor.

The reduction parameters are defined by the `value_type` typedef. This type can be any "plain old data" type (*e.g.*, no pointers, byte-wise copyable) so that the parallel_reduce operation can safely and efficiently create and use temporary values of this type. The `value_type` can be a simple aggregate (*e.g.*, a C++ struct) so a collection of scalar parameters can be reduced by a single reduction function. The *init* member function is used to initialize temporary values to the identity value of the reduction function; *i.e.*, the identity value of a summation is zero.

The `join` member function reduces the `input` value from one thread into the `update` value from a different thread. The arguments are declared `volatile` to prevent aggressive compiler optimization from assuming the input value is unchanged and "optimizing away" the reduction implementation. Each call to the `operator()` member function contributes to the reduction through the `update` argument. The `join` and `operator()` must be compatible such that if `operator()` output its contribution then application of the `join` function to the `update` and contribution values would produce the same result. The `operator()` contributes via update, as opposed to an output followed by a join, to avoid copying potentially large `value_type` parameters.

Serial Finalization. The reduced value may be serially post-processed to complete the computational kernel. For example the 2-norm computation, $\sqrt{\sum X_i * Y_i}$, performs a parallel summation followed by a serial square root. The result of a parallel_reduce operation can be output to the host for post-processing, or can be serially post-processed on the device. Serial post-processing on the device has the

performance benefit of avoiding the necessity of copying the reduction result from the device to the host, post-processing on the host, and then copying the post-processed values back to the device.

Three versions of the parallel_reduce are provided. The first two versions (Figure 8) apply the functor and output the reduction value to the host. The third version given in Figure 9 does not output the reduction value – instead it calls an application provided *reduction finalization* functor to perform a final, serial post-processing operation on the reduction value.

```
template< class ReductionFunctor ,
          class FinalizeFunctor >
void parallel_reduce( size_t NP ,
  const ReductionFunctor & functor ,
  const FinalizeFunctor  & finalize );

template< class Device >
class MyFinalizeSqrt {
public:
  typedef Device device_type ;
  typedef double value_type ;
  Value< value_type , Device > result ;
  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( const value_type & input ) const
    { *result = sqrt( input ); }
  MyFinalizeSqrt( const Value<double,Device> & arg )
    : result( arg ) {}
};
```

Figure 9: Parallel_reduce operation using finalization functor, and example finalization functor which stores the square root of a parallel reduction on the device.

5. MINI-APPLICATIONS

Performance-portability and usability of Kokkos Array are evaluated with two simple finite element mini-applications: an implicit thermal conduction mini-application and an explicit dynamics mini-application. The implicit thermal conduction mini-application computes and solves a sparse linear system on the device. The explicit dynamics mini-applications computes forces and integrates mesh motion. The mini-application's flow of computations and data between the host and manycore device are summarized in Figure 10.

Data Structures. Multidimensional arrays are used to efficiently represent the simple unstructured finite element mesh consisting of nodes (*i.e.*, vertices), elements (*i.e.*, volumes), traditional element → node connectivity, and the converse node → element connectivity. The node → element connectivity arrays are used to (1) efficiently generate the sparse linear system graph, (2) map element contributions to the linear system, and (3) perform thread-safe, lock-free parallel assembly of element contributions to the linear system. Multi-vectors are used to efficiently represent a compress row storage (CRS) sparse linear system

Element Computations. Both mini-applications perform parallel element computations via parallel_for. The implicit thermal element computations output contributions for the sparse linear system into multidimensional arrays with a per-element parallel dimension. The explicit dynamics element computations output contributions to nodal

Explicit Dynamics Mini-Application		
Serial on Host	copy	**Parallel on Device**
Generate finite element mesh	⇒	
Define boundary conditions.	⇒	
		Compute element mass Compute nodal mass
Drive time step loop:		
		Compute element forces
		Reduce stable time step
		Assemble element forces and integrate mesh motion
	⇐	Output every N-th step

Implicit Thermal Conduction Mini-Application		
Serial on Host	copy	**Parallel on Device**
Generate finite element mesh	⇒	
Define boundary conditions	⇒	
Generate linear system graph	⇒	
Map graph → element	⇒	
		Compute element contributions
		Assemble element contributions
		Apply boundary conditions
	⇐	Solve linear system

Figure 10: Finite element mini-applications' computations and data movement between the Host and Device

forces into a similar multidimensional array with a per-element parallel dimension.

Implicit thermal element computation output arrays:
  ElementMatrix(#E , #NPE , #NPE)
  ElementVector(#E , #NPE)
Explicit dynamics element computation output arrays:
  ElementVector(#E , #NPE , #D)
where:
  #E = number of elements
  #NPE = number of nodes connected to an element
  #D = spatial dimension

Assembly. The output per-element array data are assembled into the sparse linear system or nodal arrays. These assembly operations are the recommended parallel *gather assemble* algorithm for GPUs [4]. In these functions each sparse linear system row or node is defined to be an atomic unit of work with exclusive "ownership" of that row or node. A call to the assemble functor gathers all element data for that row or node and assembles that data into the sparse linear system or nodal arrays. These gather assemble operations use pre-generated graph → element and node → element maps to improve performance of this gather-assemble operation.

Boundary Conditions. Boundary conditions are applied through similar parallel operations over the rows or nodes. In the implicit thermal mini-application boundary conditions are applied by directly modifying sparse linear system data. In the explicit dynamic mini-application boundary conditions are applied by replacing a computed nodal acceleration with a value which enforces the specified boundary condition.

Solve Sparse Linear System. There is an ample body of previous and ongoing research and development (R&D) for manycore device accelerated solution strategies and algorithms for sparse linear systems. As such this R&D is not addressed within the scope of Kokkos Array programming model project.

6. PERFORMANCE

Finite element mini-applications' performance is evaluated on Intel Westmere, AMD Magny-Cours, and NVIDIA C2070 devices. Element computations and gather assemble (sparse linear system fill and nodal displacement update) operations are timed over a range of problem sizes. Performance results measure *element throughput*: the total time required for the given operation divided by the number of elements in the problem. This measure reflects the scalability of computation, Kokkos Array API and implementation, and manycore accelerator. The unmodified computational kernels are compiled for, and run on, the three devices.

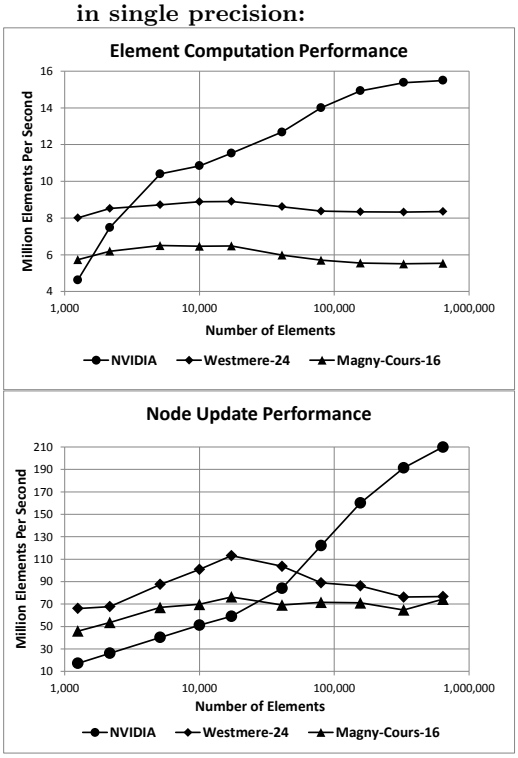
Intel Westmere: Intel Xeon X5670 at 2.93 GHz Linux Kernel v2.6.18-194.el5 24 pthreads on 2 cpus × 12 cores × 2 hyperthreads compiled with Intel v11 using -O3 optimization
AMD Magny-Cours: AMD Opteron 6136 at 2.4 GHz Linux Kernel v2.6.18-194.el5 16 pthreads on 2 cpus × 8 cores compiled with Intel v11 using -O3 optimization
NVIDIA C2070: NVIDIA C2070 at 1.2 GHz ; 448 cores compiled with CUDA v4 using -O3 -arch=sm_20

Results. Performance results from single precision and double precision instantiations of the explicit dynamics mini-application (Figure 11) and implicit thermal conduction mini-application (Figure 12) are compared for the three devices. These results demonstrate performance-portability of the mini-applications: that the same application code can be compiled and run on multicore-CPU and manycore-accelerator devices and achieve performance which is commensurate with the capabilities of the device.

The explicit dynamics mini-application has a high computational intensity (ratio of computations to global memory accesses). As such performance is dominated by the computational capability of the device. Given a sufficiently large problem, the single precision instantiation yields notably better performance with NVIDIA C2070 device versus the 24-thread Westmere and 16-thread Magny-Cours due to its higher single precision computational capability.

The sparse linear system fill operation has notably better performance for small problems where the gathered element contribution array can be cache resident; *e.g.*, for the 10,000 element problem the entire single precision element-matrix array requires only 2.56 Mbytes of storage. As this array is randomly queried by the gather-assemble operation it can remain cache-resident resulting in improved memory access performance. Note that the gather-assemble operation's queries are not truly random due to the elements and nodes having a similar ordering – yielding some *temporal locality* for element and node loop iterations. Historically, similar cache utilization improvements have been obtained by intentionally ordering finite element and node data for temporal locality.

Explicit Dynamics Mini-Application in single precision:



Explicit Dynamics Mini-Application in double precision:

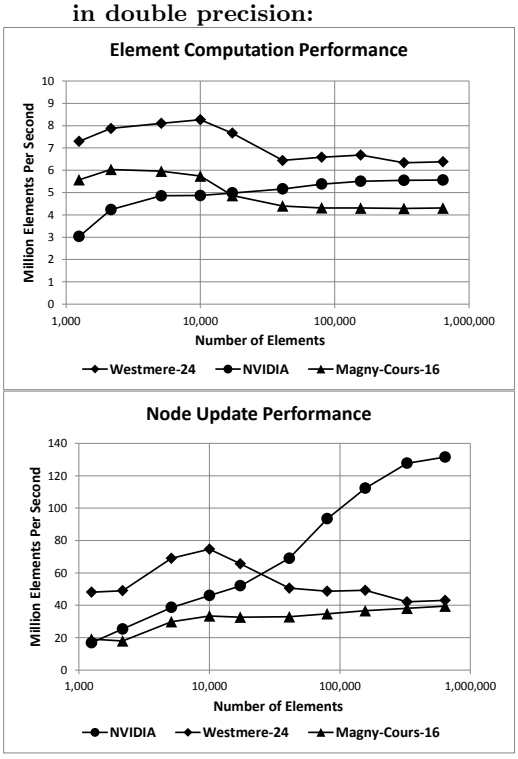
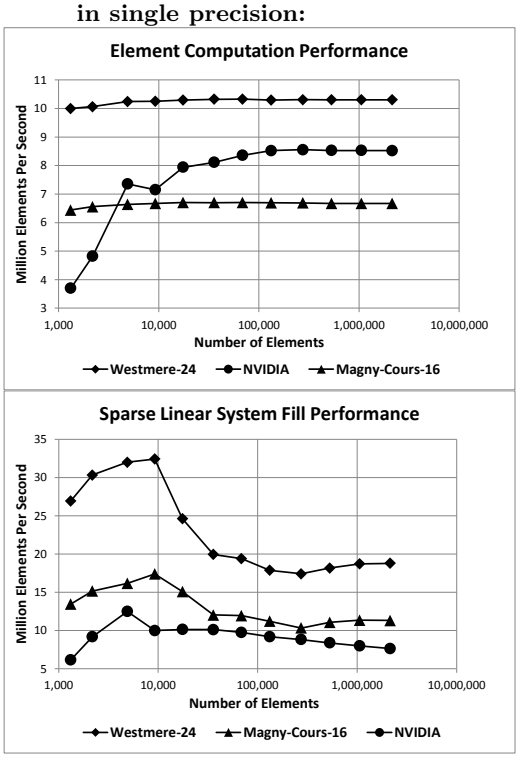


Figure 11: Explicit dynamics mini-application performance results for element and node update operations over a range of problem sizes.

Implicit Thermal Conduction Mini-Application in single precision:



Implicit Thermal Conduction Mini-Application in double precision:

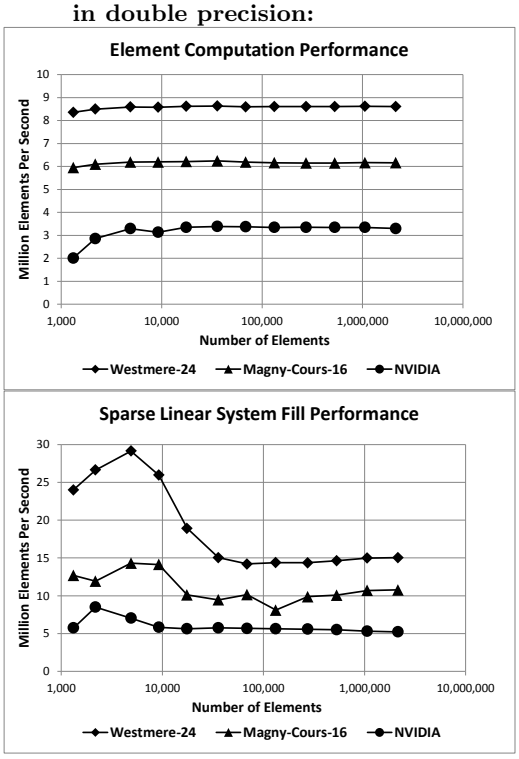


Figure 12: Implicit thermal conduction mini-application performance results for element and sparse linear system fill operations over a range of problem sizes.

NUMA Thread Pinning. The mini-applications are run on an Intel Westmere device with and without pthread placement and pinning via the HWLOC library. In both scenarios all array memory is NUMA first touched by threads associated with the array's partitions. Performance results presented Figure 13 show that for both mini-applications element throughput is dramatically improved by pinning threads to NUMA regions. In contrast NUMA thread pinning had no effect on performance results from the Magny-Cours device. More detailed performance testing and analysis is pending to identify root-cause(s) of these NUMA results.

7. CONCLUSION

The Kokkos Array performance-portable library provides a classical multidimensional array abstraction and API for computational kernels to organize and access computational data. Performance-portability is achieved using C++ template meta-programming to insert at compile-time the multi-index space mapping that is optimal for the specified many-core device.

A non-traditional shared-ownership *view* memory management abstraction is used, as opposed to the traditional exclusive-ownership container abstraction. A view abstraction is used to mitigate risks of memory management errors in large complex libraries and applications. A view abstraction is exclusively used in the programming model, as opposed to mixing container and view abstraction, to simplify the programming model and API.

Kokkos Array has been implemented on several devices and its performance is evaluated with finite element mini-applications. Performance-portability is demonstrated on Intel Westmere, AMD Magny-Cours, and NVIDIA C2070 devices: the exact same mini-application code is compiled and run on these devices and achieves performance commensurate with these devices' capabilities. A significant performance gain was demonstrated on the Westmere by pinning threads NUMA nodes and performing a correlated NUMA first touch on these arrays to place memory in associated NUMA regions. In contrast Magny-Cours performance was not effected by thread pinning.

Evaluation of the usability of the programming model and API is pending use and feedback from an "alpha user" community. Kokkos Array is available in the public domain for such an evaluation at `http://trilinos.sandia.gov`.

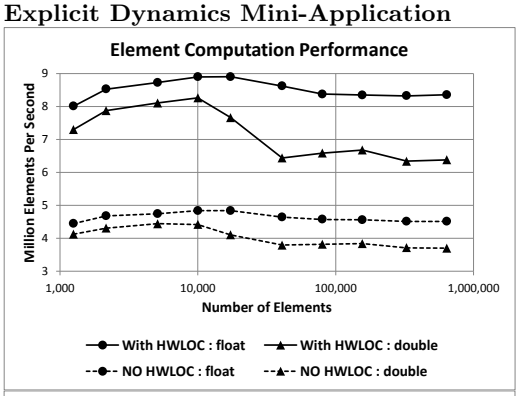
8. REFERENCES

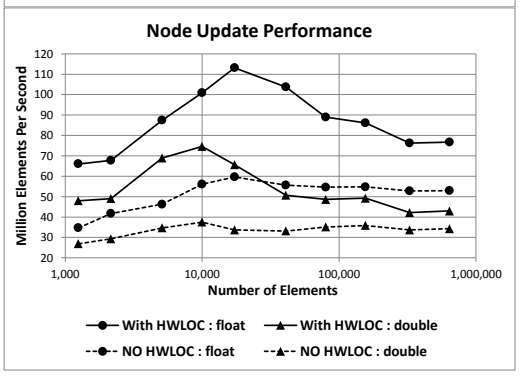
[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, first edition, 2005.

[2] Draft Technical Report on C++ Library Extensions. http://www.openstd.org /jtc1/sc22/wg21/docs/papers/2005/n1836.pdf, June 2005.

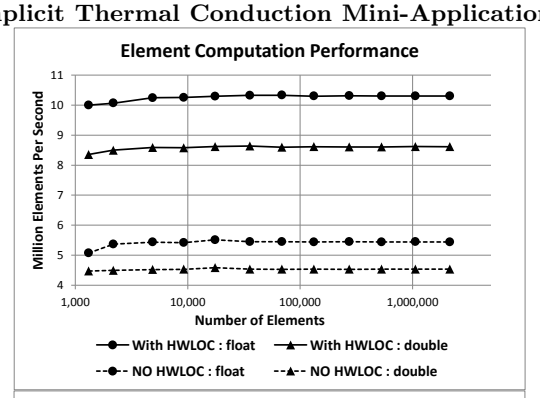
[3] H. C. Edwards, D. Sunderland, C. Amsler, and S. Mish. Multicore/gpgpu portable computational kernels via multidimensional arrays. In *Cluster Computing, 2011 IEEE Conference on Cluster Computing*, pages 363–370. IEEE Computer Society, Sept. 2011.

Explicit Dynamics Mini-Application





Implicit Thermal Conduction Mini-Application



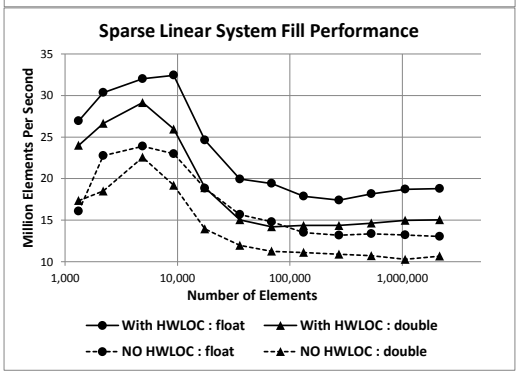


Figure 13: Intel Westmere performance results with and without thread pinning via HWLOC for mini-applications over a range of problem sizes.

[4] W.-M. W. Hwu, editor. *GPU Computing Gems Jade Edition*. Elsevier, 225 Wynn Street, Waltham, MA 02451, USA, first edition, 2012.

[5] Information Technology Industry Council. *Programming Languages — C++, International Standard ISO/IEC 14882*. American National Standards Institute, 11 West 42nd Street, New York, New York 10036, first edition, 1998.

[6] IEEE Std 1003.1, 2004 Edition, <pthread.h>, 2004.

[7] J. Reinders. *Intel Threading Building Blocks*. O'Reilly, July 2007.

[8] NVIDIA CUDA home page. http://www.nvidia.com/object/cuda_home.html, Feb. 2011.

[9] Hardware Locality library home page. http://www.open-mpi.org/projects/hwloc/, Dec. 2011.

[10] Thrust home page. http://code.google.com/p/thrust/, May 2011.

[11] Trilinos website. http://trilinos.sandia.gov/, Aug. 2011.