Scientific Programming 20 (2012) 89–114
DOI 10.3233/SPR-2012-0343
IOS Press

Manycore performance-portability: Kokkos multidimensional array library

H. Carter Edwards [a,*], Daniel Sunderland [b], Vicki Porter [b], Chris Amsler [c] and Sam Mish [d]
[a] *Computing Research Center, Sandia National Laboratories, Livermore, CA, USA*
[b] *Engineering Sciences Center, Sandia National Laboratories, Albuquerque, NM, USA*
[c] *Department of Electrical and Computer Engineering, Kansas State University, Manhattan, KS, USA*
[d] *Department of Mathematics, California State University, Los Angeles, CA, USA*

Abstract. Large, complex scientific and engineering application code have a significant investment in computational kernels to implement their mathematical models. Porting these computational kernels to the collection of modern manycore accelerator devices is a major challenge in that these devices have diverse programming models, application programming interfaces (APIs), and performance requirements. The *Kokkos Array* programming model provides library-based approach to implement computational kernels that are performance-portable to CPU-multicore and GPGPU accelerator devices. This programming model is based upon three fundamental concepts: (1) manycore *compute devices* each with its own memory space, (2) data parallel kernels and (3) multidimensional arrays. Kernel execution performance is, especially for NVIDIA® devices, extremely dependent on data access patterns. Optimal data access pattern can be different for different manycore devices – potentially leading to different implementations of computational kernels specialized for different devices. The Kokkos Array programming model supports performance-portable kernels by (1) separating data access patterns from computational kernels through a multidimensional array API and (2) introduce device-specific data access mappings when a kernel is compiled. An implementation of Kokkos Array is available through Trilinos [Trilinos website, http://trilinos.sandia.gov/, August 2011].

Keywords: Multicore, manycore, GPGPU, data-parallel, thread-parallel

1. Introduction

Manycore compute devices provide a potential gain in computational performance with respect to both runtime and energy consumption. Porting large, complex scientific and engineering high-performance computing (HPC) applications to these devices is challenging in that it introduces an additional layer (or two) of parallelism, it can be difficult to obtain good performance on these devices, and the diversity of device-specific programming models. Many projects have successfully addressed these challenges by writing distinct versions of their codes that are specialized for particular compute devices (e.g., CUDA™ [10]). However, this approach incurs the cost of developing, verifying, and maintaining a special version for each class of compute device. For large, complex scientific and engineering applications this may be an unacceptable cost.

*Corresponding author: H. Carter Edwards, Computing Research Center, Sandia National Laboratories, Livermore, CA, USA. E-mail: hcedwar@sandia.gov.

1.1. Programming model

The Kokkos Array programming model provides library-based approach to implement computational kernels that are performance-portable to manycore and GPGPU accelerator devices. This approach uses C++ template meta-programming [1], as opposed to a new or extended language, for compile-time specialization of kernels to the target manycore device. The data parallel programming model and API focuses on two responsibilities: (1) managing multidimensional array data on the manycore device and (2) executing parallel_for and parallel_reduce operations on that data.

This programming model is similar to the Thrust library [13] in that it uses C++ template meta-programming for manycore device portability, it provides parallel_for and parallel_reduce operations, and it manage data on the manycore device. The Kokkos Array programming model is unique in that it provides multidimensional array semantics with parameterized data structures such that computational kernels may be performance-portable compiled to execute on multiple

devices within the same application. Aggregation of data via multidimensional arrays has a solid history in engineering and scientific computations; as is evident by decades of Fortran use in this computational domain. In contrast, when using the Thrust library computations must use "`zip_iterator`" objects to bind individually declared one dimensional vectors into an aggregated data structure.

Kokkos Array is similar to Blitz++ arrays [17], NumPy arrays [15], PyCuda arrays [9] and Haskell Repa [12] in that multidimensional arrays have a flexible storage order (i.e., multi-index space map in Section 2.2). Blitz++ has significantly greater functionality with array subsets and slices, and a rich expression template library for efficient memory traversal during common array operations [16]. However, Blitz++ is not designed for parallel computing [18] and its flexibility and rich functionality would be very difficult to port to coalesced memory access patterns requires by NVIDIA accelerators. NumPy is also not designed for parallel computing but can be used in conjunction with python multicore-CPU capabilities. PyCuda includes the GPUArray NumPy "work-alike" multidimensional arrays on NVIDIA accelerators; however, this implementation is CUDA-specific. Haskell Repa is parallel; however, it is embedded within the Haskell programming language. In contrast the Kokkos Array programming model is a minimalistic, standard C++ implementation focused on application kernel performance-portability to multicore-CPU and manycore-accelerator devices.

1.2. 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 is the dominant constraint on performance; and memory access patterns dominate memory access performance on NVIDIA devices. As such the Kokkos multidimensional array programming model uses compile-time polymorphism (i.e., C++ template meta-programming) to insert device-optimal memory access patterns into computational kernels without requiring modification of the kernel's source code.

1.3. Implementation in Trilinos [14]

The Kokkos Array library is implemented for multicore/manycore devices using:

- a managed pool of pthreads [6],
- CUDA Version 4 [10], and
- a prototype Intel Knights Ferry [8] compute device.

A collection of performance tests and mini-applications were used to investigate performance and usability of the programming model, API and implementations.

2. Programming model

The Kokkos Array programming model is *data parallel* in that computational kernels are applied in parallel to members of a partitioned data set. This programming model is based upon the following fundamental concepts formally defined here:

- Manycore *compute device* with memory separate from the *host* main memory.
- Mapping of a multidimensional arrays onto the memory of a compute device.
- Application of data parallel computational kernels to these multidimensional arrays.

2.1. Multidimensional arrays

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 similar double precision multidimensional array `X` in the Fortran and C languages' syntax.

```
Real*8  X(1000,24,8,3);
  Fortran declaration
double  X[3][8][24][1000];
  // C declaration
```

These two declarations specify the type of the array's data (double precision) and the array's dimensions. Note that these example declarations assume and do not specify (1) the memory space in which the array resides and (2) how the array will be accessed by parallel computations. Formally, a multidimensional array is a homogeneous collection of data members (same data type and same memory space) with data members uniquely identified via N-dimensional multi-indices.

Definition 2.1. A *multi-index* is an ordered list of integer indices denoted by $(i_0, i_1, i_2, \ldots)$.

Definition 2.2. The *rank* of a multi-index is the number of indices; e.g., $(1, 3, 5)$ is a rank 3 multi-index and $(7, 3, 5, 1)$ is a rank 4 multi-index.

Definition 2.3. A Kokkos *multi-index space* I of rank R is a Cartesian product of integer ranges $I = [0, \ldots, N_0) \times [0, \ldots, N_1) \times \cdots \times [0, \ldots, N_{R-1})$. The abbreviated notation of $I = (N_0, N_1, N_2, \ldots)$ is used when there is no ambiguity between denoting a multi-index versus a multi-index space.

Definition 2.4. The *cardinality* of a multi-index space I, denoted by $\#I$, is $\#I = \prod_{j=0}^{R-1} N_j$.

Definition 2.5. A Kokkos *multidimensional array* X consists of:

- a multi-index space $X_I = (N_0, N_1, \ldots, N_{R-1})$,
- a homogeneous collection of $\#X_I$ data members residing in the memory of a compute device, and
- bijective map between the multi-index space and the data members; $X_{\text{map}} : X_I \rightarrow \{\text{data}\}$.

2.2. Multidimensional array map

There are many valid bijective maps between a multi-index space and the collection of data members. 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 in the range $[0, \ldots, \#X_I)$. For example, the Fortran and C language multidimensional array index spaces and offset-maps are as follows.

Fortran multi-index space and offset map:

- space: $[1, \ldots, N_0] \times [1, \ldots, N_1] \times [1, \ldots, N_2] \times \cdots$;
- 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: $[0, \ldots, N_0) \times [0, \ldots, N_1) \times [0, \ldots, N_2) \times \cdots$;
- offset: $((((i_0) * N_1 + i_1) * N_2 + i_2) * \cdots)$.

It can be (and has been) debated as to which of these classical multidimensional array maps is the "right" map. However, any computationally efficient bijective map is "good".

2.3. Data parallel work partitioning

A Kokkos multidimensional array is partitioned among the threads of a compute device. Each thread is responsible for applying a given computational kernel to that thread's assigned partition of the array. Currently, Kokkos partitions multidimensional arrays 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 software design review.

The partitioning of a multidimensional array is defined by the partitioning of its multi-index space $(N_P, N_1, N_2, \ldots)$, where the left-most (N_P) dimension is partitioned into N_P "atomic" units of parallel work. When a thread applies a computational kernel to a unit of work, denoted by $i_P \in [0, \ldots, N_P)$, that kernel must:

- *only* update array data members that are associated with that index $(i_P, *, *, \ldots)$ and
- *not* query array data members that are potentially updated by another thread applying the kernel to a different unit of work.

These constraints are required to prevent thread-parallel race conditions and avoid the need for inter-thread locking.

2.4. Data parallel computational kernels

Computational kernels are currently applied to parallel partitioned work via parallel_for or parallel_reduce operations. A parallel_for 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 generates N_P values which must be summed to a single value.

Definition 2.6. A *parallel_for kernel* is a function that inputs a collection of parameters and data parallel partitioned multidimensional arrays, and outputs a collection of partitioned arrays:

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

Definition 2.7. A *parallel_reduce kernel* is a function f that inputs a collection of parameters and partitioned arrays, and outputs a collection of parameters and partitioned arrays. Each application of a parallel_reduce kernel to the i_P unit of parallel work generates a corresponding contribution to the out-

put parameters. These contributions are reduced by a *mathematically* commutative and associative reduction function f_Θ. Note that an implementation f_Θ may be non-associative due to round-off in its floating point operations:

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

$$f(\{\alpha\}, \{X(i_P, \ldots)\}) \rightarrow (\{\beta[i_P]\}, \{Y(i_P, \ldots)\})\ \forall i_P$$

and then

$$f_\Theta(\{\beta[i_P]\ \forall i_P\}) \rightarrow \{\beta\}.$$

2.5. Compute device

A manycore compute device (1) owns memory which is separate from the host main memory and (2) 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® accelerator) or merely be a logical view of the same physical memory (e.g., multicore CPU). Computations performed by the compute device only access and update data which are in the compute 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 compute 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 threads will call the multiple times until the *NP* required calls are completed.

An HPC application run in an HPC environment with a distributed-memory network of manycore compute devices will have at least two heterogeneous levels parallelism: distributed memory parallelism typically supported through a Message Passing Interface (MPI) implementation and thread level parallelism. In this environment it is assumed that a distributed-memory parallel process; e.g., the process associated with a particular MPI rank, has exclusive use of at most one manycore compute device. This assumption is made to avoid introducing complexity associated with managing multiple compute devices within the same process. However, the abstraction for a single manycore compute device can aggregate multiple hardware devices into a single, logical device.

2.6. Device polymorphic multidimensional array maps

A compute device has a "preferred" multidimensional array map that yields the best performance for most computational kernels. The key concept for the Kokkos Array programming model is the use of device polymorphic multidimensional array maps. Many valid multidimensional array maps may exist; however, for a domain of computational kernels a particular map may yield the best memory access performance for a particular compute device. For example, an NVIDIA® device must have a multidimensional array map that results in a *coalesced* global memory access pattern for `parallel_for` or `parallel_reduce` kernels. Device polymorphism is implemented in the Kokkos Array programming model through C++ template meta programming, as opposed to C++ virtual interfaces, to eliminate runtime overhead. This implementation strategy allows computational kernels to have performance-portable implementations such that the best performing multidimensional array map for a given device is transparently compiled into computational kernels.

2.7. Higher rank parallel partitioning

Parallel partitioning of more than one dimension, such as for three-dimensional finite difference grids, is not currently within the scope of the Kokkos Array programming model. Such an expansion of scope will require extension of conceptual models for multidimensional `parallel_for` and `parallel_reduce` operations, and for multidimensional array maps. This extension will be challenging in that it must address the performance of data access patterns associated with concurrent parallel kernel calls over the "atomic" work space ($NP_1, NP_2, \ldots$) and the corresponding multidimensional array mapping of array data members.

3. Kokkos Array API

The Kokkos Array API is defined in three parts:

(1) Index space, data, and mapping of the index space to data (Section 3.1).

```
namespace Kokkos {
template < typename ValueType , class DeviceType >
class MDArray {
public:
  typedef ValueType value_type ; // Restricted to simple numerical types
  typedef DeviceType device_type ; // Compute device
  typedef ... size_type ; // Preferred index type for the device

  size_type rank() const ; // Rank of the array
  size_type dimension( irank ) const ; // Dimension of the 'irank' ordinate
  size_type size(); // Cardinality of the array

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

Fig. 1. Kokkos Array API for an array's multi-index space and access to array data members via multi-index mapping.

(2) View and shared ownership semantics for memory management (Section 3.2).
(3) Mirroring and copying data between device and host memory (Section 3.3).

3.1. Index space and data

A Kokkos Array composes a collection data members residing the memory of a compute device, an index space, and a bijective mapping from the index space to the data members. This portion of the API given in Fig. 1 defines the type of the data members, compute devices in which that data resides, rank and dimension of the index space, and mapping from multi-index to data member. The member data type, given by the `ValueType` template parameter, is restricted to the simple mathematical types. This limitation is imposed so that data members can be simply and optimally mapped onto compute devices with performance-sensitive memory access patterns, such as NVIDIA devices.

The performance-critical function in this API is the multi-index to data member mapping implemented by `operator()`. This fundamental mapping is heavily used and must be as efficient as possible. As such a compile-time, in-lined implementation of this function is critical.

The `DeviceType` template parameter identifies a compute device which defines the memory space, parallel execution mechanism, and preferred multi-index space mapping. The compute device is specified by a compile-time template parameter to allow the device-polymorphic multi-index space mapping to be compiled in-line. This API design avoids imposing the run-time overhead associated with run-time polymorphism; e.g., C++ virtual functions.

Kokkos Array provides a base set of types for the `DeviceType` parameter. This set is extensible to allow expert-users to define their own multi-index space mappings for existing parallel execution mechanisms and memory spaces. The `DeviceType` set is extensible (by Kokkos Array developers) to new parallel execution mechanisms, memory spaces, and devices.

3.2. View and shared ownership semantics

An `MDArray` object provides a *view* to array data, but does *not* exclusively own the that data. Multiple `MDArray` objects may view the *same* array data. These view share ownership of that array data. The `MDArray` constructors, assignment operator, and destructor given in Fig. 2 implement view and shared ownership semantics.

A Kokkos multidimensional array is created with the `create_mdarray` function (Fig. 2). This function allocates memory in the compute device's memory space for the data of the array and returns a view to that data. The calling code will retain a view of the allocated array.

```
MDArray<double,DeviceHost> x =
create_mdarray< MDArray<double,DeviceHost>
   >( nP , n1 , n2 );
```

Views are not containers. View and shared ownership semantics are fundamentally different from *con-*

```
namespace Kokkos {
template < typename ValueType , class DeviceType >
class MDArray {
public:
  MDArray(); // A NULL view.

  // New view of the same array 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 array viewed by RHS
  MDArray & operator = ( const MDArray & RHS );

  // Query if 'this' is a non-NULL view
  operator bool() const ;

  // Query if view to the same array
  bool operator == ( const MDArray & RHS ) const ;
};

// Allocate an array on the device
template< MDArrayType >
MDArrayType create_mdarray( nP , n1 , n2, ... );
}
```

Fig. 2. Kokkos Array API for memory management view and shared ownership semantics.

tainer semantics. A container has exclusive ownership of its data, versus a view which shares ownership of data with other views. An example of container versus shared ownership semantics contrast commonly used C++ containers (e.g., `std::vector`, `std::list`, `std::set`) [7] with the C++ shared pointer (`std::shared_ptr`) [2].

The copy constructor and assignment operator of `MDArray` perform a *shallow copy* – they set the current object to be a view of same data viewed by the input object (`RHS` in Fig. 2). 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* performs a *deep copy* – they allocate their own array data as needed and then copy each data member from the input container.

Why view semantics. In large complex application codes arrays are allocated on the compute 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 compute 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 a multidimensional 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 Fig. 2) 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. Mirroring and copying data

The `create_mdarray` function (Fig. 2) is called on the host process to allocate array data in the memory space of a compute device. 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 thread pool device [3]). 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 to access data. The Kokkos Array API for deep-copying and mirroring array data is given in Fig. 3.

Arrays can be allocated in the host process memory space in order to *mirror* selected arrays' data which as been allocated on a compute device. Such a mirror allows array data to be initialized on the host, copied to a compatible array on the compute device where computations are performed, and then resulting array data is copied back to the host process for output. These mirror arrays in the host memory space have the same data type, same index-space, and same multi-index map as the device resident array. Note that this multi-index map is chosen for the device and may be different that the multi-index map which may not be optimal for computations on the host.

The `MDArray` class defines the `HostMirror` type (Fig. 3). This is the type for a compatible array (*a.k.a.* same data type, index space, and multi-index map) in the host memory space. When arrays are compatible their member data may deep-copied as a block – without conversion, permutation, or other remapping.

The `create_mirror` and `deep_copy` functions simplify creation and deep copies of mirror arrays, and provide an opportunity for performance optimizations when mirroring data between a compute device and host process. The `create_mirror` function creates (allocates) an array in the host memory space which has a multi-index space and mapping compatible with the input array. The `deep_copy` function copies array data from the source array to the destination array (see Fig. 3). The usage pattern for these functions is illustrated in Fig. 4.

If the compute device shares the same memory space and multi-index mapping with the host process then this mirroring pattern can introduce unnecessary array allocations and deep copy operations. In this situation the `create_mirror` and `deep_copy` functions can, at the calling program's request, eliminate these unnecessary allocations and deep copies by causing the "`xh`" and "`yh`" to be additional *views* of the array data as viewed by "`x`" and "`y`". These views fully conform to the view and shared ownership semantics described in Section 3.2.

3.4. Illustrative test function

Array creation, view shallow copy, and deep copy operations are all illustrated in Fig. 5. In this illustrative test function an array's member data is filled on

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

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

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

Fig. 3. Kokkos Array API for mirroring and deep-copying data between host and device memory.

```
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'

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

Fig. 4. Example of mirroring and deep-copying array data between host and device memory.

```
template< class Device >
void illustrative_test_function( size_t N )
{
  typedef MDArray<double,Device> array_type ;

  array_type             dev_z ;
  array_type::HostMirror host_z ;

  {
    array_type             dev_x ;
    array_type::HostMirror host_x , host_y ;

    dev_x = create_mdarray<double,Device>( N, 10, 20 );
    dev_z = create_mdarray<double,Device>( N, 10, 20 );

    host_x = create_mirror( dev_x );
    host_y = create_mirror( dev_z );

    host_z = host_y ; // View the same data (shallow copy)

    fill( host_x ); // Fill 'host_x' with test data ...

    deep_copy( dev_x , host_x); // Copy device <- host
    deep_copy( dev_y , dev_x ); // Copy device <- device
    deep_copy( host_y, dev_z ); // Copy host   <- device

    verify_equal_member_data( host_x , host_z );
  } // The destructors deallocate the host_x and dev_x data.

  // Data originally allocated for 'host_y' still exists
  // because this data is still viewed by 'host_z'
}
```

Fig. 5. Test function illustrating creating, shallow copying, and deep copying Kokkos Arrays between the host and device.

the host, copied to the device, copied between arrays on the device, and then copied back to a different array on the host. As an illustration of view semantics the view `host_z` is assigned to also view the array al-

located for view `host_y`. When view `host_y` is destroyed the member data originally allocated for it is not deallocated because the `host_z` view to that data still exists.

3.5. Multivector and value views

The Kokkos Array API provides two additional C++ interface classes for device-resident data: `Value` and `MultiVector`. The `MultiVector` is a restricted multidimensional array – it is a collection of one-dimensional vectors all the same length. A multivector is at most rank-two (vector length and vector count) and has a "hard-wired" multi-index space mapping; thus a multivector has a simpler abstraction and API than a multidimensional array. The `Value` is the simplest – it is rank-zero. A value view is used to create and manage persistent parameters on the compute device which are typically shared by all calls to a computational kernel. For example, the $\{\beta\}$ parameters of a parallel_reduce kernel in Definition 2.7 can be maintained in the device's memory space as a value view.

Both `MultiVector` and `Value` APIs use the same shared-ownership view semantics as the `MDArray`. There APIs include corresponding C++ constructors, assignment operator, destructor, `create_multivector`, `create_value`, `create_mirror`, and `deep_copy` functions as were defined for the `MDArray`. For brevity details of the simpler `MultiVector` and `Value` APIs have not been included here.

4. Computational kernel *functor*

Computational kernels are implemented as *functors* for execution by parallel_for or parallel_reduce operations. A functor is the composition of a computation, its parameters, and views to array data to which the computation is applied – recall Definitions 2.6 and 2.7. Functor semantics are common to several programming models; for example, the C++ Standard Template Library (STL) algorithms [7], Intel Threading Building Blocks [11] and Thrust [13].

In the Kokkos Array programming model a functor is created on the host process, copied to the compute device, and then run thread-parallel on the compute device. A functor is required to (1) identify the intended compute device, (2) provide computational function(s) and (3) have views of the data on which it will operate. Functor API requirements are defined for `parallel_for` and `parallel_reduce` functors in Sections 4.1 and 4.3.

4.1. Parallel for functor interface

A Kokkos Array `parallel_for` functor is a C++ class which (1) has a template parameter for the manycore device, (2) identifies that device via "`typedef ··· device_type`" and (3) provides a computation via "`operator()(iP)`". The example `parallel_for` functor given in Fig. 6 illustrates these functor API requirements.

For compile-time portability to different manycore devices a functor must:

- have a template parameter for the manycore device,
- define all array data through MDArray or MultiVector views which are instantiated with the device template parameter,
- implement the `operator()` with the subset of C++ that is supported by the compute devices (i.e., CUDA), and
- only access array data through the array views' multi-index map operator.

Instantiation of the array views causes a device-preferred multi-index mapping to be compiled in-line into the functor's `operator()(iP)` function.

A functor's `operator()(iP)` function is called `nP` times, where `nP` is the integer value passed to the `parallel_for` (or `parallel_reduce`) statements. Each call to the functor is passed a unique index `iP` in the range `[0..nP]`. For thread-safe parallel execution the functor's `operator()(iP)` computation must:

- *only* update array data associated with the parallel work index `iP`,
- *not* query array data that is being updated by another call to the functor (i.e., another value of `iP`),
- *never* assume a particular multi-index mapping to data members.

The example functor in Fig. 6 only accesses appropriate parallel partitioned data members of the input and output arrays (`X(iP,*,*)` and `Y(iP,*)`), and only accesses array data through the array API.

4.2. Functor performance considerations

Minimize global memory reads and writes. This consideration is critical for NVIDIA® compute devices, and noticeable even for host-memory multicore devices. In a CPU multicore device several cores share a memory controller and thus share access to global memory. Minimizing these cores' global mem-

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

  typedef MDArray<double,device_type> array_type ;
  const array_type X , Y ; // Input and output arrays
  int n1 , n2 ;            // constant input parameters

  KOKKOS_MACRO_DEVICE_FUNCTION    // REQUIRED qualifier
  void operator()( int iP ) const // REQUIRED operator
  {
    // Average values from rank-3 array X into
    // the corresponding values of rank-2 array Y
    for ( int i = 0 ; i < n1 ; ++i ) {
      ValueType tmp = 0 ;
      for ( int j = 0 ; j < n2 ; ++j ) { tmp += X(iP,i,j); }
      Y(iP,i) = tmp / n2 ; // write only once
    }
  }
  // Construct functor with views to input and output arrays
  // and other useful parameters
  ExampleParalleForFunctor( const array_type & arg_x ,
                            const array_type & arg_y )
    : X( arg_x ), Y( arg_y ) // view shallow copy
    , n1( X.dimension(1) ) , n2( X.dimension(2) ){}
};
// Call this functor nP times on the manycore device:
//   parallel_for( nP , ExampleParallelForFunctor(myX,myY) );
```

Fig. 6. Pseudo code for an example `parallel_for` kernel that averages terms from a rank-3 array into the corresponding terms of a rank-2 array.

ory reads and writes reduces demands for this shared resources, and thus reduces the cores' contention for access to global memory. A simple example of minimizing global memory access is illustrated in Fig. 6 where a local temporary variable is used to accumulate the `X(iP,i,*)` values so that the corresponding `Y(iP,i)` global data is written exactly once.

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 in Fig. 7. In the first thread execution profile every thread (or GPU thread-block) 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 are serialized. In the second thread (or GPU thread-block) 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.

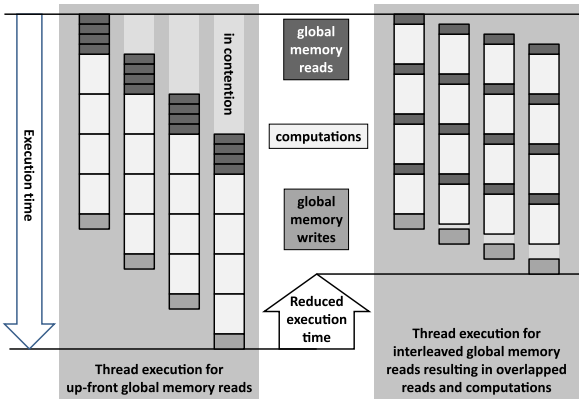

Fig. 7. Conceptualization of overlapping global memory access and computations for threads or thread-blocks sharing access to global memory. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0343.)

```
MDArray< double , Device > input_coordinates ;
MDArray< double , Device > output_gradients ;
void operator()( int ielem ) const
{
  double local_coordinates[3][8] ;
  // Read coordinates from global memory once
  // and use them many times in the computations
  local_coordinates[0..2][0..7] = input_coordinates(ielem,0..2,0..7);
  // Compute and write 'X' gradients to global memory
  output_gradient(ielem,0,0..7) = ... ;
  // Compute and write 'Y' gradients to global memory
  output_gradient(ielem,1,0..7) = ... ;
  // Compute and write 'Z' gradients to global memory
  output_gradient(ielem,2,0..7) = ... ;
}
```

Fig. 8. Pseudo code summary of initial hexahedral finite element uniform gradient functor with all global reads up front.

This memory contention behavior and subsequent performance improvement is demonstrated in the hexahedral finite element uniform gradient unit performance test case. In this functor each unit of work requires 24 global memory reads, performs 318 floating point operations, and then outputs via 24 global memory writes. An initial implementation of this functor read all global data "up front" before performing its computations; as illustrated in Fig. 8.

Dispersing global memory reads to "just in time" locations within the work function reduces the block of time during which threads are in contention for global memory access. This allows the threads' aggregate global memory reads and computations to be overlapped. The dispersed global memory read version of the functor summarized in Fig. 9 demonstrated improved aggregate performance as compared to the initial version.

Compile-time knowledge of dimensions. In the hexahedral finite element uniform gradient example the computation is explicitly defined for arrays with an index space of (#Elements $\times 3 \times 8$). However, the `MDArray` objects have runtime, not compile-time, knowledge of the index space. As such an implementation of the index space mapping cannot leverage this compile-time knowledge to pre-compute portions of the mapping. 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 enhanced API only the parallel work dimension is still declared at runtime, and all other dimensions declared at compile-time. This planned enhancement will allow a multi-index mapping to use compile-time defined dimensions, a good compiler to pre-compute operations using those dimensions.

4.3. Parallel reduce functor interface

In addition to the basic `parallel_for` functor requirements identified in Section 4.1 a Kokkos Array `parallel_reduce` functor must implement the f_Θ reduction function from Definition 2.7. These additional requirements are illustrated in Fig. 10 and explicitly enumerated here.

Reduce functor requirements to implement f_Θ in Definition 2.7.

(1) Define the value type of the $\{\beta\}$ parameters to be reduced.

- `typedef ··· value_type;`
- The `value_type` must be a "plain old data" type so that the `parallel_reduce` operation can create and simply copy temporary values.
- Multiple parameters may be reduced by defining `value_ type` to be a simple aggregate, as illustrated in Fig. 10.

(2) Provide a *join* function to reduce parameters from two different threads into a single value.

- `static void join(`
  `volatile value_type & update,`
  `volatile const value_type &`
  `input);`

```
MDArray< double , Device > input_coordinates ;
MDArray< double , Device > output_gradients ;
void operator()( int ielem ) const
{
  double local_coordinates[8] ;
  // Read 'Z' coordinates from global memory
  local_coordinates[0..7] = input_coordinates(ielem,2,0..7);

  // Compute intermediate 'Z' values in temporary variables
  // Read 'Y' coordinates from global memory
  local_coordinates[0..7] = input_coordinates(ielem,1,0..7);
  // Compute and write 'X' gradients to global memory
  output_gradient(ielem,0,0..7) = ... ;

  // Compute intermediate 'Y' values in temporary variables
  // Read 'X' coordinates from global memory
  local_coordinates[0..7] = input_coordinates(ielem,0,0..7);
  // Compute and write 'Z' gradients to global memory
  output_gradient(ielem,0,0..7) = ... ;

  // Compute intermediate 'X' values in temporary variables
  // Read 'Z' coordinates from global memory
  local_coordinates[0..7] = input_coordinates(ielem,2,0..7);
  // Compute and write 'Y' gradients to global memory
  output_gradient(ielem,2,0..7) = ... ;
}
```

Fig. 9. Pseudo code summary of final hexahedral finite element uniform gradient functor with dispersed "just in time" global reads.

- This function reduces the `input` value into the `update` value according the f_Θ reduction function. For example, the summation of a scalar value is simply `update += input;`.
- The arguments are `volatile` to prevent a compiler from "optimizing away" thread-concurrent updates to these values.

(3) Provide an *init* function to initialize reduce parameters to the "identity" value of the `join` operation.

- `static void init(value_type & update);`
- For example, identity for a scalar summation operation is zero so an *init* operation is simply `update = 0;`.

(4) Extend the computation function to contribute $\{\beta[i_P]\}$ parameters into the global reduction.

- `void operator()(`*integer_type* `iP, value_type & update)const;`
- The function's contribution to the `update` parameter must conform to the `join` function. For example: let `input_iP` be the contribution generated by a call to this function, then the contribution of this value conforms to `join(update,input_iP)`.
- The function contributes to `update`, as opposed to returning a `input_iP` value to avoid copying potentially large `value_type` parameters.

4.4. Serial finalization of a reduction

A parallel reduce operation's output parameter(s) $\{\beta\}$ can be returned to the host or serially processed on the device. Three versions of the parallel reduce APIs are provided for these options, as illustrated in Fig. 11. The first version simply outputs the reduction value as a "return" parameter. The second version copies the reduction value to a calling argument so aggregate reduction values can avoid the extra copy operation associated with "return" values. The third version does not output the reduced parameter(s) – instead it calls an application-provided *reduction finalization* functor to perform a final, serial operation on those parameters.

The serial reduction finalization operation could simply store the resulting value on the device through a `Value` object, as illustrated in Fig. 11. The finalization functor may perform a more involved serial computa-

```
template< class Device >
class ExampleReduceFunctor {
public:
  typedef Device device_type ;
  typedef struct { double cm[3] , m ; } value_type ;

  typedef MDArray< double , Device > array_type ;
  array_type mass , coord ;

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int iP , value_type & update ) const
  { const double m = mass(iP);
    update.cm[0..2] += m * coord(iP,0..2); update.m += m ; }

  KOKKOS_MACRO_DEVICE_FUNCTION
  static void join( volatile value_type & update ,
                    volatile const value_type & input )
  { update.cm[0..2] += input.cm[0..2]; update.m += input.m ; }

  KOKKOS_MACRO_DEVICE_FUNCTION
  static void init( value_type & output )
  { update.cm[0..2] = 0 ; update.m = 0 ; }

  ExampleReduceFunctor( const array_type & arg_mass ,
                        const array_type & arg_coord ,
    : mass( arg_mass ), coord( arg_coord ) {}
};
// Call this functor nP times on the device:
//   parallel_reduce( nP, ExampleReduce(mass,coord));
```

Fig. 10. Pseudo code example `parallel_reduce` functor for a center-of-mass computation that accumulates mass and mass-weighted coordinates.

tions and access multiple input parameters and output to multiple value views. This functionality is intended to allow sequences of functors to improve performance by eliminating unnecessary device-to-host data movement and synchronization operations; as illustrated in the next section.

4.5. Sequence of functors performance considerations

Execution of a `parallel_reduce` functor produces a set of reduction parameters $\{\beta\}$. In a sequence of computational kernels these parameters are typically output from one functor, processed by some small serial computation, and the results become input parameters for one or more subsequent functors. The Modified Gram–Schmidt unit performance test case executes a sequence of `parallel_for` and `parallel_reduce` functors and serial computations as illustrated in Fig. 12.

This example uses the third version of the `parallel_reduce` operator which accepts both a reduction functor and a reduction finalization functor. Each reduction operation is given a finalize functor which serially processes the result of the reduction *on the device*. As such, the result of this serial computation remains on the device for use by the next functor. This use allows the complete modified Gram–Schmidt algorithm to execute without copying data from device memory to host memory. For an NVIDIA manycore accelerator this strategy allows the sequence of functors to be asynchronously dispatched to the device, and thus eliminates all device-host synchronizations which would interrupt the execution of the algorithm on the device.

5. Implicit thermal conduction finite element mini-application

Performance-portability and usability of the programming model and API are demonstrated with a finite element mini-application that forms and solves a simple thermal conduction problem. The mini-application's flow of computations and data between the host and manycore device are summarized in Table 1.

```
template< class ReductionFunctor >
typename ReductionFunctor::value_type
parallel_reduce( size_t NP , const ReductionFunctor & functor );

template< class ReductionFunctor , class FinalizeFunctor >
void parallel_reduce( size_t NP ,
                      const ReductionFunctor & functor ,
                      typename ReductionFunctor::value_type & result );

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

template< class Device >
class ExampleFinalizeFunctor {
public:
  typedef Device device_type ;
  typedef ... value_type ;
  Value< value_type , Device > result ;
  // REQUIRED reduction finalization operator
  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( const value_type & input ) const
  { *result = input ; }
  ExampleReduce( const Value< value_type , Device > & arg )
    : result( arg ) {}
};
```

Fig. 11. Three versions of a `parallel_reduce` operation and an example parallel reduce finalization functor which stores the result of a parallel reduction on the device.

```
int K = ... ; // Number of vectors
int N = ... ; // Length of vectors
MultiVector<double,Device> Q = ... ; // Matrix
MultiVector<double,Device> R = ... ; // Coefficients
Value<double,Device> tmp ;
for ( int i = 0 ; i < K ; ++i ) {
  // Reduction: result = dot( Q(*,i) , Q(*,i) );
  // Finalize: tmp = sqrt(result); R(i,i) = tmp ; tmp = 1 / tmp ;
  parallel_reduce( N, Dot(Q,i), Finalize_A(tmp,R,i) );
  // Q(*,i) *= tmp ;
  parallel_for( N, Scale(Q,i,tmp) );
  for ( int j = i + 1 ; j < K ; ++j ) {
    // Reduction: result = dot( Q(*,i) , Q(*,j) );
    // Finalize: R(i,j) = result ; tmp = - result ;
    parallel_reduce( N, Dot(Q,i,Q,j), Finalize_B(tmp,R,i,j) );
    // Q(*,j) += tmp * Q(*,i);
    parallel_for( N, Add(Q,j,tmp,Q,i) );
  }
}
```

Fig. 12. Pseudo code example for modified Gram–Schmidt orthonormalization algorithm using reduction finalization functors for serial post-processing of reduction results.

Table 1
Implicit thermal conduction finite element mini-application computations and data movement between the host and manycore device

Serial on Host	copy	**Parallel on Manycore Device**
Generate finite element mesh.	$\Longrightarrow$	
Define Dirichlet boundary conditions.	$\Longrightarrow$	
Generate sparse linear system graph and element $\rightarrow$ graph mapping.	$\Longrightarrow$	
		Compute element contributions to the sparse linear system.
		Assemble element contributions into sparse linear system.
		Update sparse linear system to enforce boundary conditions.
		Solve sparse linear system.
	$\Longleftarrow$	Copy out solution.

```
// Element->node connectivity array:
MDArray<int,Device> element_node_identifier = ... ;
// Node->element connectivity arrays:
MDArray<int,Device> node_element_offset = ... ;
MDArray<int,Device> node_element_identifier = ... ;
// Converse relationship of connectivity arrays:
for ( int node_id = 0 ; node_id < node_count ; ++node_id ) {
  for ( int i = node_element_offset( node_id ) ;
            i < node_element_offset( node_id + 1 ) ; ++i ) {
    int element_id = node_element_identifier( i , 0 );
    int node_local = node_element_identifier( i , 1 );
    assert( node_id == element_node_identifier( element_id , node_local ) );
  }
}
```

Fig. 13. Code fragment verifying the converse relationship between the element $\rightarrow$ node and the node $\rightarrow$ element connectivity arrays.

5.1. Unstructured finite element mesh

The mini-application uses a simple unstructured finite element mesh consisting of nodes (i.e., vertices), elements (i.e., volumes) and connectivity between nodes and elements. The mesh data structure consists of four multidimensional arrays.

(1) A rank-two node coordinate array dimensioned (nodes $\times$ spatial-coordinates).
(2) A rank-two element $\rightarrow$ node connectivity array dimensioned (elements $\times$ nodes-per-element) containing the integer identifiers for each node of each element.
(3) A pair of node $\rightarrow$ element connectivity arrays which follow a "compressed sparse row" scheme: a rank-one *node element offset* array and a rank-two *node element identifier* array. These arrays represent the converse of the element $\rightarrow$ node arrays, as described in Fig. 13.

The node $\rightarrow$ element arrays are used to efficiently generate the sparse linear system graph from the mesh and perform the parallel gather-assemble of the linear system (Section 5.4).

5.2. Sparse matrix graph

This simple thermal conductivity mini-application defines one degree of freedom at each node in the mesh; i.e., each node has an associated row and column in the linear system of equations. The compressed sparse row (CSR) matrix graph is efficiently generated from the element $\rightarrow$ node and node $\rightarrow$ element connectivity arrays as specified in Fig. 14. This implementation is serial on the host. A parallel, manycore device implementation would straight-forward – given an extension of the Kokkos Array programming model to include a `parallel_scan` operation.

Kokkos Arrays (multidimensional array and multivector) are simple fixed-dimension objects that once

```
MDArray<int,Host> element_node_identifier = ... ;
MDArray<int,Host> node_element_offset = ... ;
MDArray<int,Host> node_element_identifier ... ;
MultiVector<int,Host> A_offset = ... ;
MultiVector<int,Host> A_col_ids ;

std::vector<int> A_col_ids_tmp ;

A_offset(0) = 0 ;
// One matrix row per node in this mini-application problem
for ( int row_node = 0 ; row_node < node_count ; ++row_node ) {
  std::set<int> row_col_ids ;
  // Loop over each element connected to this node
  for ( int i = node_element_offset( row_node ) ;
            i < node_element_offset( row_node + 1 ) ; ++i ) {
    int element_id = node_element_identifier( i , 0 );

    // Add matrix columns for row_node->element->col_node relations
    for ( int k = 0 ; k < nodes_per_element ; ++k ) {
      int col_node = element_node_identifier( element_id , k );
      row_col_ids.insert( col_node ); // A sorted and unique set
    }
  }
  append( A_col_ids_tmp , row_col_ids ); // Sorted by node identifier

  A_offset(row_node+1) = A_col_ids_tmp.size();
}
A_col_ids = create_multivector< MultiVector<int,Host> >( A_offset(node_count) );
A_col_ids(...) = A_col_ids_tmp[...] ; // Copy
```

Fig. 14. Pseudo code for the generation of a compressed sparse row (CSR) matrix graph from the element → node and node → element connectivity arrays.

created cannot be dynamically resized. As such the generation of a CSR matrix graph in Fig. 14 dynamically fills an `std::vector` container and then copies its contents to a created Kokkos multi-vector.

5.3. Element computations

The thermal conduction finite element computation is applied to each element via a `parallel_for` operation. Each element computation produces contributions to the sparse linear system of equations associated with the nodes of that element. The functor for this computation gathers data from global memory, performs computations using local memory, and then outputs results back to global memory. These memory access patterns and the associated multidimensional arrays for global data are summarized in Fig. 15.

The resulting `element_matrix` and `element_vector` arrays hold *unassembled*, per-element contributions to the linear system of equations. In a serial computation these contributions can be immediately assembled into the linear system by identifying the correct locations in the sparse linear system and summing the corresponding contributions into those locations. In a shared-memory thread-parallel an immediate-assembly operation is not thread-safe as two or more threads may attempt to sum into the same location at the same time – a race condition. One solution is to use mutually exclusive locking (e.g., pthread mutex) to protect against such a race condition. However, this locking solution has a potentially large performance penalty relative to the simple arithmetic sum operation that it is protecting. In addition, the locking solution introduces a serialization point in the element loop. Furthermore, such a locking solution allows contributions to be summed in any order leading to nondeterministic results.

5.4. Gather assemble

Assembly of the sparse linear system is a thread-safe and deterministic implementation of the gather-

```
template< typename Scalar , class Device >
class Element {
public:
  MDArray<int,Device> element_node_identifier ;
  MDArray<double,Device> node_coordinates ;
  MDArray<Scalar,Device> element_matrix , element_vector ;

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int element_id ) const
  {
    // Gather element's node coordinates from scattered
    // global-memory into contiguous local-memory:
    double coordinates[8][3] ;
    for ( int j = 0 ; j < nodes_per_element ; ++j ) {
      int node_id = element_node_identifier( element_id, j);
      coordinates[j][0..2] = node_coordinates(node_id,0..2);
    }
    // Perform thermal conduction computations to populate
    // matrix and vector contributions in local-memory.
    Scalar matrix_contrib[8][8] , vector_contrib[8] ;
    // ... EXTENSIVE ELEMENT COMPUTATIONS HERE ...
    // Output matrix and vector contributions to global-memory:
    element_vector( element_id, 0..7) = vector_contrib[0..7] ;
    element_matrix( element_id, 0..7, 0..7) = matrix_contrib[0..7][0..7] ;
  }
}
```

Fig. 15. Pseudo code overview of the element computation functor's global and local memory data interactions.

assemble algorithm (see Chapter 16 in [5]). This implementation has two components: (1) a mapping of element contributions to the sparse linear system given in Fig. 16 and (2) a parallel gather-assemble operation using that mapping given in Fig. 17. Each call to the gather-assemble work operator (Fig. 17) has exclusive access to its given row of the linear system; thus the operation is scalable, thread-safe and lock-free. Element contributions are sorted by element identifier; thus the summation is deterministic with respect to thread execution ordering.

5.5. Boundary conditions

Application of boundary conditions follows the same strategy as the gather-assemble (Section 5.4) operation. A boundary condition enforcement functor is executed on the manycore device where a unit of parallel work is defined by a row of the linear system. In the current mini-application Dirichlet boundary conditions are enforced through algebraic elimination of the associated degrees of freedom. The manycore device-parallel, thread-safe, lock-free, and efficient functor given in Fig. 18 is applied to perform this algebraic elimination within the sparse linear system from finite element contributions.

5.6. Solve Sparse Linear System

The sparse linear system is now fully defined and ready to be solve via an iterative method. Data for the "$Ax = b$" linear system includes three sparse matrix arrays, right-hand side vector, and solution vector. 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.

5.7. Performance

Performance of the implicit thermal conduction finite element mini-application is evaluated on the Westmere, Magny-Cours and NVIDIA manycore compute devices. The element computations (Section 5.3) and sparse linear system gather assemble, or "fill", operation (Section 5.4) are timed over a range of prob-

```
MDArray<int,Host> element_node_identifier = ... ;
MDArray<int,Host> node_element_offset = ... ;
MDArray<int,Host> node_element_identifier = ... ;

MultiVector<int,Host> A_offset = ... ;
MultiVector<int,Host> A_col_ids = ... ;

MDArray<int,Host> element_CSR_map =
  create_mdarray< MDArray<int,Host> >(
    element_count , nodes_per_elem , nodes_per_elem );

for ( int elem_id = 0 ; elem_id < elem_count ; ++elem_id ) {
  for ( int i = 0 ; i < nodes_per_elem ; ++i ) {
    int row_node = elem_node_ids( elem_id , i );
    for ( int j = 0 ; j < nodes_per_elem ; ++j ) {
      int column_node = elem_node_ids( elem_id , j );
      // Find 'index' in the sorted range such that:
      // A_offset(row_node) <= index < A_offset(row_node+1)
      // column_node == A_col_ids(index)
      // A log( NZ ) search where
      // NZ = A_offset(row_node+1)- A_offset(row_node)
      element_CSR_map( elem_id, i, j) =
        find_index( A_offset, A_col_ids, row_node, column_node );
    }
  }
}
```

Fig. 16. Algorithm to set up the map for the thread-safe and deterministic gather-assemble of element contributions into the sparse linear system.

lem sizes. The same, unmodified, computational kernels are compiled for, and run on, the three manycore devices using the Kokkos array interface and library.

Westmere:	Intel Xeon X5670 at 2.93 GHz; 24 pthreads on 2 cpus × 6 cores × 2 hyperthreads compiled with Intel v11 using -O3 optimization
Magny-Cours:	AMD Opteron 6136 at 2.4 GHz; 16 pthreads on 2 cpus × 8 cores compiled with Intel v11 using -O3 optimization
NVIDIA 2070:	NVIDIA C2070 at 1.2 GHz; 448 cores compiled with CUDA v4 using -O3 -arch=sm_20

Performance results presented in Fig. 19 compare element throughput for the three manycore devices. Element throughput is measured as the "number of elements contributions computed per second" and "number of element contributions filled per second", where *filled* refers to the gather assemble operation to fill the sparse linear system coefficients. Element contribution computations gather nodal coordinates and element quadrature data from global memory, performs approximately 7000 floating point operations, and writes 72 element contribution coefficients to device global memory. In contrast the element fill computations read from device global memory the element→linear system map arrays as well as the 72 element contribution coefficients, and then update 72 sparse linear system coefficients at random locations in device global memory. Thus element computations are dominated by compute performance while fill operations are dominated by device global memory access performance.

Performance of the combined element computation and fill operation is the *complete* performance of interest – the time between problem specification and linear system solution (Table 1). This combined performance is presented in Fig. 20 for a problem with 1.06 million elements. On all three manycore devices the time required to fill the sparse linear system is roughly equal to the time required to compute the element contributions. This result is an example of the HPC colloquialism that "floating point operations are free", in comparison to memory access performance.

```
template< typename Scalar , class Device >
class GatherAssemble {
public:
  enum { nodes_per_elem = 8 };
  MDArray<Scalar,Device> element_matrix , element_vector ;
  MDArray<int,Device> element_CSR_map ;
  MDArray<int,Device> node_element_offset , node_element_identifier ;
  MultiVector<int,Device> A_offset , A_col_ids ;
  MultiVector<Scalar,Device> A , b ;

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int node_id ) const
  { // Iterate the sorted list of elements connected to this node.
    int iElemBeg = node_element_offset(node_id);
    int iElemEnd = node_element_offset(node_id+1);
    for ( int i = iElemBeg ; i < iElemEnd ; ++i ) {
      int element_id = node_element_identifier(i,0);
      int elem_row = node_element_identifier(i,1);
      b( node_id ) += element_vector(element_id,elem_row);
      for ( int elem_col = 0 ; elem_col < nodes_per_elem ; ++elem_col ) {
        int index = element_CSR_map( element_id , elem_row , elem_col );
        A( index ) += element_matrix( element_id , elem_row , elem_col );
      }
    }
  }
}
```

Fig. 17. Algorithm for the thread-safe and deterministic gather assemble of element contributions into the sparse linear system.

6. Finite element explicit dynamics mini-application

The programming model and the API were also tested using an explicit dynamics finite element mini-application that solves an elastic bar impact problem. This mini-application uses a simple unstructured finite element mesh consisting of nodes (i.e., vertices), elements (i.e., volumes), and connectivity between nodes and elements. The mesh data structure described in Section 5.1 is also used in this mini-application. The mini-application's flow of computations and data between the host and the manycore device are summarized in Table 2.

6.1. Internal force computation

This simple explicit dynamics mini-application models the structural response of a bar with an initial impulse on one end and having the other end fixed. The elements used are uniform gradient hexahedral elements with hourglass mode stabilization. The impulse is applied as an initial velocity and the kinematic boundary condition on the far end is imposed by computing accelerations to match the fixed condition rather than through an external force. Therefore the only forces that are computed on the nodes are the internal forces that arise from the divergence of the stress.

The internal force is computed for each element using a sequence of two `parallel_for` functors and a `parallel_reduce` functor over the elements. This computation is split into three functors in an attempt to (1) maximize sharing global memory accesses among multiple operations and (2) minimize the amount of local memory required by the functor. These first two functors compute the element deformation gradient and element stretches and rotations. The last functor computes the stress, the corresponding internal forces for the element's nodes, and the minimum stable time step for next time step iteration.

First functor. The first `parallel_for` functor computes the gradient, velocity gradient, and the hourglass stabilization operators. It gathers from global memory the element's nodal coordinates and velocities and stores them in local arrays from which this data will is repeatedly used in the computation. Gradient operators computed in local methods are also stored in

```
template< typename Scalar , class Device >
class BoundaryConditions {
public:
  MultiVector<int,Device>    bc_flags ;
  MultiVector<Scalar,Device> bc_values ;
  MultiVector<int,Device>    A_offset , A_col_ids ;
  MultiVector<Scalar,Device> A , b ; // Solving b = A*x

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int row ) const
  { // This thread has exclusive access to this row
    const int iBeg = A_offset(row), iEnd = A_offset(row+1);
    if ( bc_flags( row ) ) {
      // Set this row's equation to: bc_value = 1 * x
      b(row) = bc_values(row);
      for ( int i = iBeg ; i < iEnd ; ++i ) A(i) = row == i ? 1 : 0 ;
    }
    else {
      for ( int i = iBeg ; i < iEnd ; ++i ) {
        int column = A_col_ids(i);
        if ( bc_flags(column) ) {
          // Eliminate 'x' from the equation:
          b(row) -= bc_value(column) * A(i); A(i) = 0 ;
        }
      }
    }
  }
}
```

Fig. 18. Manycore device parallel C++ functor for a thread-safe and lock-free algebraic elimination of boundary conditions.

local arrays for the nodes of each element. Then the deformation gradient is computed using a series of three local function calls. These memory access patterns and the associated multidimensional arrays for global data are summarized in Fig. 21.

Second functor. The second functor computes the kinematic quantities needed for strain rate. This functor copies another set of global variables from global to local memory; e.g., velocity gradient (output by the first functor), rotations, stretch, and vorticity. The functor computes the strain rate of the element which is input to the last functor.

Last functor. The last functor computes the element stress, hourglass resistance, and internal forces applied at the element's nodes. The functor also computes an estimate of the stable time step for the element. This stable time step value is reduced over all elements to determine the minimum stable time step for the next time step iteration. The call to the `parallel_reduce` is illustrated in Fig. 22 and the definitions of the functors that comprise this call are shown in Figs 23 and 24.

The `divergence` functor in Figs 22 and 23 does all of the parallel work of the computation. The `set_next_time_step` functor is a serial parallel reduce finalization functor described in Section 4.4. This finalization functor takes the result of the reduction operation, the minimum of the elements' stable time steps, and places it in an variable on the manycore device. This time step value is now ready to be used on the device in the next time step iteration.

6.2. Nodal forces and time integration

The time step is finished by a functor that sums the internal forces at the nodes, computes the nodal accelerations from the forces and masses, and updates the velocities and displacements using a central difference time integration. This functor, illustrated in Fig. 25, is applied to each node via the `parallel_for` operation. The summation of internal forces at the nodes uses the gather-assemble algorithm (Section 5.4) for scalability and determinism.

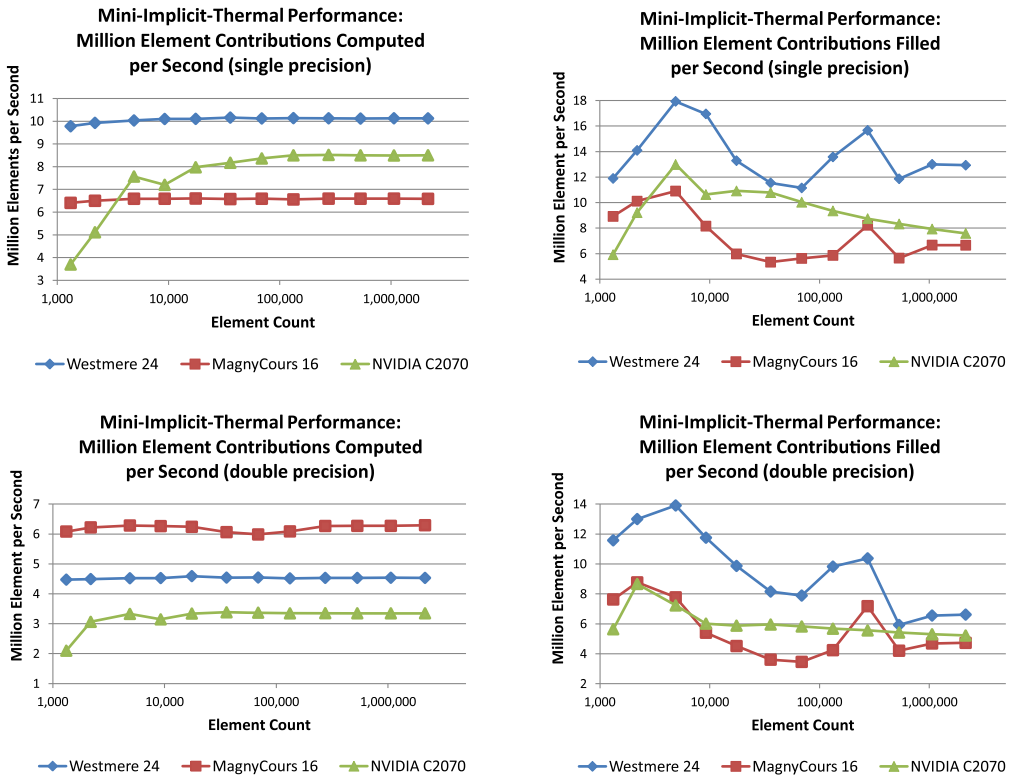

Fig. 19. Element computation and fill performance on Westmere, Magny-Cours and NVIDIA for (1) single and double precision linear system and (2) problem sizes from one thousand to two million elements. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0343.)

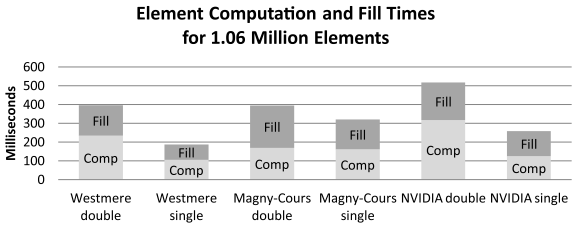

Fig. 20. Comparison of Westmere, Magny-Cours and NVIDIA element computation and fill performance for 1.06 million elements.

6.3. *Performance*

Performance of the explicit dynamics finite element mini-application is evaluated on the Westmere, Magny-Cours and NVIDIA manycore compute devices. The sequence of element internal force computations (Section 6.1) and nodal forces and time integration computations (Section 6.2) are timed over a range of problem sizes. The same, unmodified, computational kernels are compiled for, and run on, the three manycore devices using the Kokkos Array interface and library.

Westmere 24: Intel Xeon X5670 at 2.93 GHz; 24 pthreads on 2 cpus × 12 cores × 2 hyperthreads compiled with Intel v11 using -O3 optimization

Magny-Cours 16: AMD Opteron 6136 at 2.4 GHz; 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

Performance results presented in Fig. 26 compare element throughput for the three manycore devices. Element throughput is measured in "element-steps per second" which is defined as the total number of elements in the problem divided by the runtime for all elements' internal force computations plus all nodes' force and time integration computations. The mini-application is run for one thousand application-steps, the runtime is accumulated across all application-steps, and the mean element-steps per second is re-

Table 2
Explicit dynamics finite element mini-application computations and data movement between the host and manycore device

Serial on Host	copy	**Parallel on Manycore Device**
Generate finite element mesh.	$\Longrightarrow$	
Define initial and prescribed boundary conditions.	$\Longrightarrow$	
		Compute element mass.
		Compute nodal mass.
Time step loop:		
		Compute element gradients and hour glass operator.
		Compute element tensor decomposition and rotations.
		Compute element stresses and internal forces.
		Compute and reduce the elements' minimum stable time step.
		Gather forces and update acceleration, velocity and displacement.
	$\Longleftarrow$	Copy out selected variables every N time steps.

```
template< typename Scalar , class Device >
struct grad_hgop {
public:
  MDArray<int,Device> element_node_connectivity ;
  MDArray<double,Device> node_coordinates ;
  MDArray<Scalar,Device> model_coords ;
  MDArray<Scalar,Device> displacement ;
  MDArray<Scalar,Device> velocity;
  MDArray<Scalar,Device> vel_grad;
  MDArray<Scalar,Device> hgop;
  MDArray<Scale,Device>  dt;

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int element_id ) const
  {
    // Declare local data for frequent access to reduce
    // global memory reads and writes.
    Scalar      x[8],      y[8],      z[8]; // nodal coordinates
    Scalar     vx[8],     vy[8],     vz[8]; // nodal velocities
    Scalar grad_x[8], grad_y[8], grad_z[8]; // gradient_operator

    // Gather this element's nodal coordinates and velocities.
    // Compute element gradient operator, volume, velocity gradient,
    // and hourglass operator.
  }
}
```

Fig. 21. Pseudo code overview of the deformation gradient functor's global and local memory data interactions.

corded:

runtime(application-step)
= runtime(all elements' internal force)
+ runtime(all nodes' force summation)
+ runtime(all nodes' time integration),

element-steps per second
= #elements ÷ runtime(application-step).

Given sufficient parallel work (a sufficient number of elements) the 448 core NVIDIA C2070 had dramatically better performance in single precision – nearly 15 million element-steps per second versus the 4–6 million element-steps per second of the Westmere and Magny-Cours. This result is due to the kernels having a favorable computational intensity; i.e., having a favorable ratio of floating point operations versus global memory accesses.

```
Kokkos::parallel_reduce( num_elements ,
        divergence<Scalar, device_type>(...arguments...),
        set_next_time_step<Scalar,device_type>(...arguments...) );
```

Fig. 22. Pseudo code overview of the call combining a reduce functor for stress computation and global minimum time step.

```
//Structure for computing element stress, stable time step, and element
internal forces.
template< typename Scalar , class DeviceType >
struct divergence;

template<typename Scalar>
struct divergence<Scalar, KOKKOS_MACRO_DEVICE>{
  typedef KOKKOS_MACRO_DEVICE device_type ;
  typedef typename Kokkos::MDArray<Scalar,device_type> array_type ;
  typedef typename Kokkos::MDArray<int,device_type>    int_array_type ;
  typedef Kokkos::Value<Scalar,device_type>            scalar_view;
  typedef Scalar value_type ;

  const int_array_type elem_node_connectivity;

  // numerous other member array view declarations
  const Scalar user_dt;
  const scalar_view dt;

  KOKKOS_MACRO_DEVICE_FUNCTION
  static void init(value_type &update) {
    update = 1.0e32; // initialize to large value for minimum time step
  }
  KOKKOS_MACRO_DEVICE_FUNCTION
  static void join(volatile value_type &update, const volatile value_type & source) {
    update = update < source ? update : source; // keep the minimum
  }
  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int ielem, value_type & update ) const {
    // local memory
    Scalar x[8], y[8], z[8]; //Element nodal coordinates in local memory.

    // member functions to compute end of step gradient operators
    // computation for computing the element stable time step, cur_time_step

    update = update < cur_time_step ? update : cur_time_step;

    // compute stress and internal force
  }
}
```

Fig. 23. Pseudo code overview for computation of element stress and global minimum time step.

7. Conclusion

7.1. Summary

The Trilinos–Kokkos Array performance-portable programming model provides a "classical" multidimensional array abstraction and API for computational kernels to organize and access computational data. This abstraction is partitioned into component abstractions of a data set, multi-index space, multi-index space mapping and data parallel partitioning. A multi-index space mapping that provides device-specific op-

```
template<typename Scalar>
struct set_next_time_step<Scalar ,KOKKOS_MACRO_DEVICE>{
  typedef KOKKOS_MACRO_DEVICE device_type;
  typedef Scalar value_type;
  // ... Members views and constructor ...
  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()(Scalar & result) const {
    *prev_dt = *dt;
    *dt = result;
  }
}
```

Fig. 24. Pseudo code overview of the finalization functor to set minimum stable time step.

```
template<typename Scalar>
struct finish_step<Scalar ,KOKKOS_MACRO_DEVICE>{
  typedef KOKKOS_MACRO_DEVICE device_type;
  typedef device_type::size_type size_type;
  // ... member views and constructor to pass in those views ...

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()(int inode) const {

    // Getting offset and count as per 'CSR-like' data structure
    const int elem_offset = node_elem_offset(inode);
    const int elem_count = node_elem_offset(inode+1) - elem_offset ;

    // for each element that contains the node to accumulate internal force
    double local_force[] = {0.0, 0.0, 0.0};
    for(int i = 0; i < elem_count ; i++){
      const int elem_id = node_elem_ids(elem_offset+i 0);
      const int elem_node_index = node_elem_ids(elem_offset+i,1);
      local_force[0..2] += element_force(elem_id, 0..2, elem_node_index);
    }
    internal_force(inode, 0..2) = local_force[0..2]; // copy local to
    global memory

    // Computations for acceleration including at fixed boundary
    // Each line stores local a_current for fast access later in the
    velocity update
    // and copies the values to the global acceleration array.

    Scalar a_current[3]; // Local storage for subsequent use

    if ( /* ... a fixed boundary condition ... */ ) {
      // enforce fixed BC by zeroing out acceleration
      acceleration(inode,0..2) = a_current[0..2] = 0;
    } else { // not on a fixed boundary
      acceleration(inode,0..2) = a_current[0..2] = - local_force[0..3]/nodal_mass ;
    }

    // Computations for central difference integration of velocities and displacements
  }
}
```

Fig. 25. Pseudo code overview for functor to gather internal forces and update acceleration, velocity and displacement.

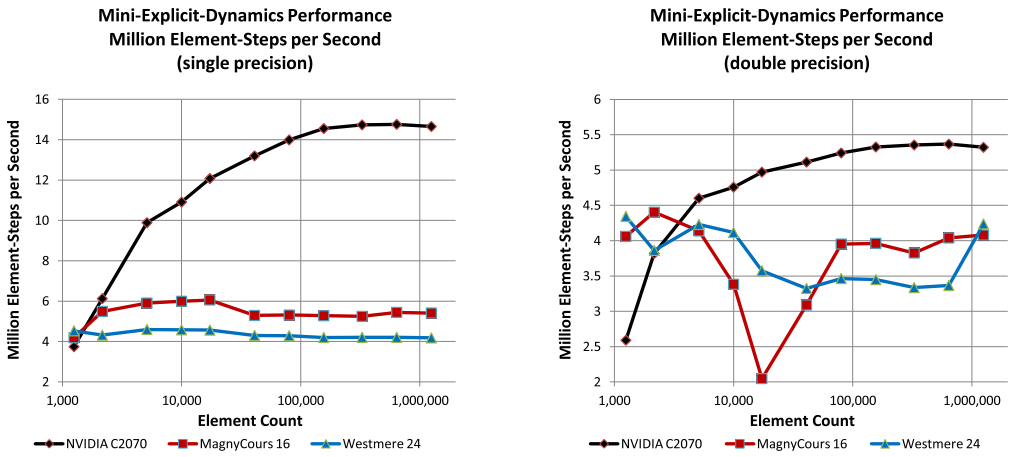

Fig. 26. Explicit dynamics element internal force plus node force summation and time integration performance Westmere, Magny-Cours and NVIDIA for (1) single and double precision and (2) problem sizes from one thousand to one million elements. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2012-0343.)

timal memory access patterns is transparently introduced at compile-time through C++ template metaprogramming.

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.

The Kokkos Array programming model has been implemented on several manycore devices and evaluated with unit performance test cases, implicit thermal finite element mini-application, and explicit dynamics finite element mini-application. These mini-application test cases provide evidence that application kernel performance-portability can be achieved among Westmere and Magny-Cours multicore devices and NVIDIA C2070 manycore device. Evaluation of the usability of the programming model and API is pending use and feedback from an "alpha user" community. The Trilinos–Kokkos is available for such an evaluation at http://trilinos.sandia.gov.

8. Plans

The Kokkos Array programming model, API and implementations presented here are the result of one year (October 10–September 11) of research and development. Ongoing and planned research & development activities are focused on

- more in-depth performance analysis and improvements on current devices,
- usability analysis and improvements through mini-application [4] focused dialogues,
- introduction of compile-time dimensions to allow a compiler to optimize multi-index space mappings,
- support concurrent execution of multiple heterogeneous kernels within `parallel_for` and `parallel_reduce` operations, and
- expand the scope of the programming model to include two-level heterogeneous parallelism with message passing at the outer level and manycore devices at the inner level.

Acknowledgements

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the US Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This paper is cross-referenced at Sandia as SAND2011-8102J.

References

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

[2] Draft technical report on C++ library extensions, available at: http://www.openstd.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf, June 2005.

[3] H.C. Edwards, Trilinos ThreadPool Library v1.1, Technical Report SAND2009-8196, Sandia National Laboratories, Albuquerque, NM, December 2009.

[4] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist and R.W. Numrich, Improving performance via mini-applications, Technical Report SAND2009-5574, Sandia National Laboratories, Albuquerque, NM, September 2009.

[5] W.-M.W. Hwu, ed., *GPU Computing Gems Jade Edition*, 1st edn, Elsevier, Waltham, MA, 2012.

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

[7] Information Technology Industry Council, *Programming Languages – C++*, 1st edn, International Standard ISO/IEC 14882, American National Standards Institute, New York, 1998.

[8] Intel unveils new product plans for high-performance computing, available at: http://www.intel.com/pressroom/archive/releases/2010/20100531comp.htm, May 2010.

[9] A. Klöckner, PyCUDA multidimensional arrays on the GPU, available at: http://documen.tician.de/pycuda/array.html, December 2011.

[10] NVIDIA CUDA homepage, http://www.nvidia.com/object/cuda_home.html, February 2011.

[11] J. Reinders, *Intel Threading Building Blocks*, O'Reilly, Sebastopol, CA, 2007.

[12] D. Stewart, Numeric Haskell: a repa tutorial, available at: http://www.haskell.org/haskellwiki/Numeric_Haskell:_A_Repa_Tutorial, October 2011.

[13] Thrust homepage, http://code.google.com/p/thrust/, May 2011.

[14] Trilinos website, http://trilinos.sandia.gov/, August 2011.

[15] S. van der Walt, S.C. Colbert and G. Varoquaux, The numpy array: a structure for efficient numerical computation, *Comput. Sci. Eng.* **13**(2) (2011), 22–30.

[16] T.L. Veldhuizen, Arrays in Blitz++, in: *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments*, Santa Fe, NM, USA, Springer-Verlag, 1998, pp. 223–230.

[17] T.L. Veldhuizen, Blitz++ website, http://www.oonumerics.org/blitz/, March 2006.

[18] T.L. Veldhuizen, Parallel computing with Blitz++, available at: http://www.oonumerics.org/blitz/odocs/blitz_8.html, March 2006.