SAND2011-3953C

Manycore/GPGPU Portable Computational Mechanics Kernels via Multidimensional Arrays

H. Carter Edwards
Computing Research Center
Sandia National Laboratories
Albuquerque, New Mexico USA
hcedwar at sandia.gov

Daniel Sunderland
Engineering Sciences Center
Sandia National Laboratories
Albuquerque, New Mexico USA
dsunder at sandia.gov

Chris Amsler
Elec. Eng. Dept.
Kansas State University
Manhattan, Kansas USA
camsler at ksu.edu

Sam Mish
Mathematics Dept.
California State University
Chico, California USA
smish at mail.csuchico.edu

***Abstract*—Large, complex scientific and engineering application code have a significant investment in software kernels to implement their computational mechanics models. Porting these computational mechanics 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 *Trilinos-Kokkos* programming model provides library-based approach to implement computational mechanics kernels that are performance-portable to manycore and GPGPU accelerator devices. This programming model is based upon three fundamental concepts: (1) there exists one or more manycore *compute devices* each with its own memory space, (2) data parallel kernels are executed via parallel_for and parallel_reduce operations, and (3) kernels operate on multidimensional arrays. Kernel execution performance is, especially for NVIDIA® devices, extremely dependent on data access patterns. An optimal data access pattern can be different for different manycore devices – potentially leading to different implementations of computational kernels specialized for different devices. The Trilinos-Kokkos programming model support performance-portable kernels by separating data access patterns from computational kernels through a multidimensional array API. Through this API device-specific mappings of multi-indices to device memory are introduced into a computational kernel through compile-time polymorphism; *i.e.*, without modification of the kernel.**

***Keywords*-HPC Programming model ; multicore ; manycore ; GPGPU**

I. Introduction

Manycore compute devices provide a significant 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 programming models. Many projects have successfully addressed these challenge by writing distinct versions of their codes that are specialized for particular compute devices [1]. However, this approach incurs the cost of developing, verifying, and maintaining a specialized code base for each class of compute device. For large, complex scientific and engineering applications this may be an unacceptable cost.

The *Trilinos-Kokkos* programming model provides library-based approach to implement computational mechanics kernels that are performance-portable to manycore and GPGPU accelerator devices. This approach uses C++ *template meta-programming*, 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 provides data structures, parallel_for operation, and parallel_reduce operation that are parameterized with respect to the multicore / manycore device. This approach to multicore / manycore portability mirrors the approach taken by the Thrust library [2]. However, the Thrust programming model and API implements C++ Standard Template Library (STL) container and iterator semantics as opposed to multidimensional array semantics.

Performance-portability includes source code portability of a kernel implementation *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 modification of the kernel's source code.

II. Programming Model

The Kokkos multidimensional array programming model is *data parallel* in that a computational kernel can be applied in parallel to members of a partitioned data set. This programming model consists of the following conceptual components.

1) compute device with many computational threads and shared memory,
2) multidimensional array,

3) partitioning and mapping of multidimensional arrays onto the memory space of a compute device, and
4) application of data parallel computational kernels to partitioned multidimensional arrays.

A. Manycore Compute Device

A manycore compute device provides many threads of execution and owns memory that is shared by those computational threads. Computations performed by the compute device only access and update data which are resident in the compute device's shared memory. A *heterogeneous parallel* application utilizes both distributed-memory process parallelism among a network of compute nodes and shared-memory thread parallelism within a manycore compute device. In such an application it is assumed that a distributed-memory parallel process; *e.g.*, the process associated with a message passing interface (MPI) rank, has exclusive use of at most one manycore compute device. This assumption is made to avoid the 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.

B. Multidimensional Array

There are several abstractions for multidimensional arrays which are differentiated by how they are used within an application. Abstractions currently addressed by Kokkos include the multivector and the "array valued array." A multivector is a homogeneous collection of vectors of the same length, with data members of the same simple mathematical data type, and those data members resided in the memory space of a compute device. A multidimensional array (array valued array) is a similar homogeneous collection of data members where data members are uniquely identified by a multi-index within a multi-index space. Both the multivector and multidimensional array have a one dimensional partitioning into units of data parallel work: a multivector is parallel-partitioned along its length and a multidimensional array is parallel-partitioned along one dimension of its index space. The member data type is restricted to the simple intrinsic mathematical types of the computational languages (*i.e.*, `C++` and CUDA) so that data members can be simply and optimally mapped onto compute devices such as NVIDIA GPGPU. A formal definition for Kokkos multidimensional array follows.

Definition 1: A Kokkos **multi-index** is an ordered list of integer indices denoted by $(i_0, i_1, i_2, \cdots)$.

Definition 2: The **rank** of a multi-index is the number of indices; *e.g.*, $(1, 3, 5)$ is a multi-index of rank 3 and $(7, 3, 5, 1)$ is a multi-index rank 4.

Definition 3: A Kokkos **multi-index space** I of rank R is a Cartesian product of integer ranges

$$I = [0 \,..\, N_0) \times [0 \,..\, N_1) \times \cdots \times [0 \,..\, N_{R-1}) \,.$$

The abbreviated notation of $I = (N_0, N_1, N_2, \cdots)$ is used when there is no ambiguity between denoting a multi-index versus a multi-index space.

Definition 4: The **cardinality** of a multi-index space I, denoted by $\#I$, is

$$\#I = \prod_{j=0}^{R-1} N_j$$

Definition 5: A Kokkos **multidimensional array** X consists of

1) a multi-index space $X_I = (N_0, N_1, \cdots, N_{R-1})$,
2) a homogeneous collection of $\#X_I$ data members, and
3) bijective map between the multi-index space and the data members; $X_{map} : X_I \leftrightarrow \{\text{data}\}$.

C. Multidimensional Array Map

There are many valid bijective maps between a multi-index space and the collection of data members. Typically 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 $[0..\#X_I)$. 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: $[1 \,..\, N_0] \times [1 \,..\, N_1] \times [1 \,..\, N_2] \times \cdots$
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: $((((i_0) * N_1 + i_1) * N_2 + i_2) * \cdots)$

While many valid multidimensional array maps may exist their may be one map which yields the best performance for a computational kernel on a particular compute device, and a different map which yields the best performance on a different compute device. The Kokkos multidimensional array programming model allows this map to be specified when a computational kernel is compiled and without modifying the kernel's source code.

D. Data Parallel Work Partitioning

A Kokkos multivector or multidimensional array are partitioned among the threads of a compute device. Each thread then applies a given computational kernel to that thread's assigned partition of the array. Multivectors are partitioned along their length, under the assumption that operations applied to each vector are data paralle and that the length of the multivector is greater than the count of vectors. Multidimensional arrays are partitioned along exactly one dimension of the multi-index space. For Kokkos the left-most dimension was chosen for parallel partitioning by a consensus of computational kernel developers participating

in a Kokkos software design review; however, it would have been equally valid to have chosen any other dimension.

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

E. Data Parallel Computational Kernels

Computational kernels are currently applied to parallel partitioned work in two ways: `parallel_for` and `parallel_reduce`. A `parallel_for` application is "trivially" parallel in that the computational kernel's work is fully disjoint. In a `parallel_reduce` application 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 6: A **`parallel_for` kernel** is a function that inputs a collection of parameters and data parallel partitioned multidimensional arrays and outputs a corresponding collection of partitioned arrays.

$$\begin{array}{l} f : (\{\alpha\}, \{X\}) \rightarrow \{Y\} \\ \quad \text{where} \quad \begin{array}{rcl} \{\alpha\} & \equiv & \text{input parameters} \\ \{X\} & \equiv & \text{input arrays} \\ \{Y\} & \equiv & \text{output arrays} \end{array} \end{array}$$

Definition 7: A **`parallel_reduce` kernel** is a function that inputs a collection of parameters and data parallel partitioned arrays and outputs a collection of parameters and data parallel partitioned arrays.

$$\begin{array}{l} f : (\{\alpha\}, \{X\}) \rightarrow (\{\beta\}, \{Y\}) \\ \quad \text{where} \quad \begin{array}{rcl} \{\alpha\} & \equiv & \text{input parameters} \\ \{X\} & \equiv & \text{input arrays} \\ \{\beta\} & \equiv & \text{output parameters} \\ \{Y\} & \equiv & \text{output arrays} \end{array} \end{array}$$

Each application of a `parallel_reduce` kernel generates a contribution to the output parameters. These contributions are reduced by a *mathematically* commutative and associative parameter reduction function f_Θ (an implementation may be non-associative due to round-off in floating point operations).

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

III. Multidimensional Array API

The multidimensional array API given in Figure 1 composes (1) a runtime defined multi-index space, (2) a collection of data members of a compile-time defined simple mathematical type, and (3) a compile-time device-polymorphic map from the multi-index space to data members. Each device has a default map which may be overridden to support investigation of alternative maps to improve memory access performance.

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

  // Query rank and dimensions of the array.
  KOKKOS_MACRO_DEVICE_AND_HOST_FUNCTION
  size_type rank() const ;

  KOKKOS_MACRO_DEVICE_AND_HOST_FUNCTION
  size_type dimension( irank ) const ;

  // Query cardinality of the array.
  KOKKOS_MACRO_DEVICE_AND_HOST_FUNCTION
  size_type size();

  // Access data member on the device referenced
  // by its mapped multi-index.
  KOKKOS_MACRO_DEVICE_FUNCTION
  value_type & operator()(iP,i1,...) const ;

  // A NULL view.
  MDArrayView();

  // A new view of the same member data viewed by RHS.
  MDArrayView( const MDArrayView & RHS );

  // Clear this view: if it is the last view to an
  // allocated array then deallocate the member data.
  ~MDArrayView();

  // Clear this view and then assign it to be a
  // new view of the same member data viewed by RHS.
  MDArrayView & operator = ( const MDArrayView & RHS );
};

// Allocate member data on the device and return
// an array view to that member data.
template< typename ValueType , class DeviceType >
MDArrayView<ValueType,DeviceType>
create_mdarray( nP , n1 , n2, ... );

// Copy member data between two arrays with potentially
// different devices.  One device can be the Host.
template< typename ValueType , class DeviceDestination ,
                               class DeviceSource >
void deep_copy(
  const MDArrayView<ValueType,DeviceDestination> & ,
  const MDArrayView<ValueType,DeviceSource> & );
}
```

Figure 1. The Kokkos multidimensional array API composes a member data type, multi-index space, multi-index map to device memory, and view semantics.

A. View and Shared Ownership Semantics

The `MDArrayView` C++ class API (Figure **??**) is a multidimensional array *view* to member data on the manycore device. In view semantics there may exist multiple views to the same member data such that all views share ownership of that member data. Multiple views to the same member data are created via the copy constructor and assignment operator; these functions perform "shallow" copies of the input view. This in contrast to *container* semantics where a container exclusively owns its member data and the copy constructor and assignment operator perform a "deep" copy by allocating new member data and copying all members of the input container.

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 eventually 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 loose 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 (*e.g.*, see `shared_ptr` in [3]) 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 **??**) deallocates the allocated data.

B. Device versus Host Allocated Data

The `create_mdarray` function is called by the application on the host process to allocate array member data in the memory space of a designated device. The `DeviceType` may be a manycore device (*e.g.*, NVIDIA® device) or the host "device"; *i.e.*, the host process running the application code. Views to allocated member data will exist on both the host process and within kernels running on the manycore device. However, ember data allocated on one device may not be directly accessible to another device; *e.g.*, data allocated on the host is not directly accessible to an NVIDIA® device and vice-versa.

Kokkos API functions in Figure 1 are designated as available only on the device (`DEVICE_FUNCTION`), available on both the device and host (`DEVICE_AND_HOST_FUNCTION`), or available only on the host (no macro designation). Functions which describe the shape of an array (rank and dimensions) are available on both the device and host. However, functions which provide access to member data are only available on the device.

The `deep_copy` function is called on the host process to copy array member data between arrays allocated on the manycore device and host. These arrays must have conformal multi-index spaces; *i.e.*, their rank and dimensions must be equal. However, the source and destination arrays may have different maps from the conformal multi-index space to the data members. Thus the `deep_copy` both copies member data between memory spaces and potentially permutes member data through the composition of the two array's multi-index space maps.

C. Illustrative Test Function

Array creation, view "shallow" copy, and "deep" copy semantics are illustrated in Figure 2. In this illustrative test function an array's member data is filled on 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 allocated 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.

```
template< class Device >
void illustrative_test_function( size_t N )
{
  typedef Kokkos::DeviceHost Host ;
  MDArrayView<double,Host>   host_z ;
  MDArrayView<double,Device> dev_z ;
  {
    MDArrayView<double,Host>   host_x , host_y ;
    MDArrayView<double,Device> dev_x ;

    host_x = create_mdarray<double,Host>(   N, 10, 20 );
    host_y = create_mdarray<double,Host>(   N, 10, 20 );
    dev_x  = create_mdarray<double,Device>( N, 10, 20 );
    dev_z  = create_mdarray<double,Device>( N, 10, 20 );
    host_z = host_y ; // View the same allocate data

    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 );
  } // deallocation of host_x and dev_x data

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

Figure 2. Illustrative test case for creating, "shallow" copying, and "deep" copying Kokkos arrays between the host and manycore device.

IV. Computational Kernel Functor API

Computational kernels conform to *functor* semantics and APIs for either `parallel_for` or `parallel_reduce` operations. A functor is the composition of a computation and the data, or views of data, to which the computation is applied (recall Definitions 6 and 7). Functor semantics are to many programming models; for example, the C++ Standard Template Library (STL) algorithms [4], Intel Threading Building Blocks [5], and Thrust [2] use functors.

In the Trilinos-Kokkos programming model a functor is created on the host process, copied to the device for execution, and then run thread-parallel on the device. In the multidimensional array API a `parallel_for` functor must identify a target manycore device and provide a computational function, as illustrated in Figure 3. API requirements for a `parallel_reduce` functor are more involved to support the thread-parallel reduction operation f_Θ in Definition 7.

```
template< typename ValueType , class Device >
class ExampleFunctor {
public:
  typedef Device device_type ; // Required

  // Average values from rank-3 array X into
  // the corresponding values of rank-2 array Y
  const MDArrayView<ValueType,Device> X ;
  const MDArrayView<ValueType,Device> Y ;
  int n1 , n2 ;

  ExampleFunctor(
    const MDArrayView<ValueType,Device> & arg_x ,
    const MDArrayView<ValueType,Device> & arg_y )
  : X( arg_x ), Y( arg_y ) // view shallow copy
  , n1( X.dimension(1) ) , n2( X.dimension(2) ){}

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int iP ) const // Required
  {
    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 ;
    }
  }
};
// Run this functor on the manycore device:
//   parallel_for( nP , ExampleFunctor(myX,myY) );
// which calls operator() nP times in parallel
// with iP = 0..nP-1
```

Figure 3. An example `parallel_for` kernel that averages terms from a rank-3 array into the corresponding terms of a rank-2 array.

The example functor in Figure 3 has the device as a template parameter. A functor must be templated with respect to the device and use the `DEVICE_FUNCTION` macro for the functor to be compile-time portable to multiple devices. When the functor template is instantiated with a device the internal array views are also instantiated with the device. This compile-time instantiation of the array views causes a device-specific multi-index $\rightarrow$ data member map to be compiled into the functor, resulting in an optimal memory access pattern by the `operator()( int iP )` computation.

For thread-safe parallel execution the functor's `operator()( int iP )` computation (1) must only access the data members of the input and output arrays associated with the parallel work index `iP` and (2) do *not* assume a particular map from multi-indices to data members. For well-performing execution the computation should only access each data member of an array exactly once. In the example functor a temporary variable is used to accumulate the `X(iP,i,*)` values so that the corresponding `Y(iP,i)` data member is updated exactly once. This "access only once" principle is used due to the large global memory access times of some manycore devices (*e.g.*, NVIDIA®), and the small overhead associated with the multi-index map operation.

A. Reduction Functors

The functor API requirements for a reduction computational kernel (*i.e.*, no reduction) are illustrated in Figure 4. The type definitions are required for the `parallel_for` driver to select and compatibly run the functor on the compute device. The work function is implemented by the functor's parenthesis operator. This function is called `work_count()` times where each call is given a unique work index value in the range `[0..(work_count()-1)]`. The work operator is expected to access appropriate parallel partitioned data members of the input and output arrays; *i.e.*, `X(*,*,iwork)` and `Y(*,*,*,iwork)`.

API requirements for a `parallel_reduce` kernel include the device and computation operator similar to the `parallel_for` API requirements. In addition the reduction computation f_Θ API is defined by (1) the `value_type` of the values to be accumulated, (2) a `join` function that reduces values accumulated by two different threads into a single value, and (3) an `init` function that initializes a value prior to beginning accumulation. The `parallel_reduce` operation creates and manages a `value_type` accumulation value for each thread. As such the `value_type` must be a simple "plain old data" type; *i.e.*, a raw memory copy of `value_type` values yields the correct copy-result. These values are initialized with the `init` function and reduced with the `join` function. The arguments of the `join` function are qualified with `volatile` to prevent the compiler from introducing thread-unsafe optimizations within the minimal-overhead inter-thread reduction operations.

V. Performance of Initial Implementation

Performance-portability of the initial implementation is evaluated through two test cases. These performance-evaluation test cases will be used to analyze and improve the runtime performance of the programming model, and the usability of the API. The first case applies the `parallel_for` to a kernel which computes the gradients of a linear hexahedral finite element's basis functions. Each work unit loads an element's eight vertex coordinates from global memory (24 values), performs 231 floating point operations, and writes the eight basis function gradients to global memory (24 values). The coordinate and basis function gradients data are managed in rank-three arrays with dimensions of number of elements, spatial dimension, and number of vertices (or linear basis functions) per

```
template< class Device >
class ExampleReduce {
public:
  typedef Device                          device_type ;
  typedef struct { double total[4] ; } value_type ;

  MDArrayView<double,device_type> mass ;
  MDArrayView<double,device_type> coord ;

  // Functor-specific constructor copies views
  ExampleReduce(
    const MDArrayView<double,device_type> & arg_mass ,
    const MDArrayView<double,device_type> & arg_coord ,
  :  mass( arg_mass ), coord( arg_coord ) {}

  KOKKOS_MACRO_DEVICE_FUNCTION
  void operator()( int iP , value_type & update ) const
  {
    const double m = mass(iP);
    update.total[0] += m * coord(iP,0);
    update.total[1] += m * coord(iP,1);
    update.total[2] += m * coord(iP,2);
    update.total[3] += m ;
  }

  //  update = ReduceOperation( update , input )
  KOKKOS_MACRO_DEVICE_FUNCTION
  static void join( volatile       value_type & update ,
                    volatile const value_type & input )
  {
    update.total[0] += input.total[0];
    update.total[1] += input.total[1];
    update.total[2] += input.total[2];
    update.total[3] += input.total[3];
  }

  //  output = ReduceOperatorIdentity
  KOKKOS_MACRO_DEVICE_FUNCTION
  static void init( value_type & output )
  {
    update.total[0] = 0 ;
    update.total[1] = 0 ;
    update.total[2] = 0 ;
    update.total[3] = 0 ;
  }
};
// Run this functor on the manycore device:
//   parallel_reduce( nP, ExampleReduce(mass,coord));
// which calls operator() nP times in parallel
// with iP = 0..nP-1
```

Figure 4. An example `parallel_reduce` kernel that accumulates mass-weighted coordinates for a center-of-mass computation.

element. The second case implements a Modified Gram-Schmidt orthogonalization algorithm through a sequence of `parallel_for` and `parallel_reduce` operations on simple "level one" basic linear algebra operations (*i.e.*; vector scaling, addition, and inner products). In this test case the orthogonalization algorithm is applied to a multivector of dimension $N \times 32$.

The two test cases are implemented with the portable multidimensional array programming model and API. These test cases are instantiated for double precision data and computations, and run on the following multicore / manycore devices.

NVIDIA	Tesla C2070	448 cores
Intel	Xeon X5680	2 sockets × 6 cores
AMD	Opteron 6172	2 sockets × 12 cores

For the Xeon and Opteron devices a range of Pthread counts were run and results obtained from the best performing Pthread counts are presented in Figure 5 and Figure 6.

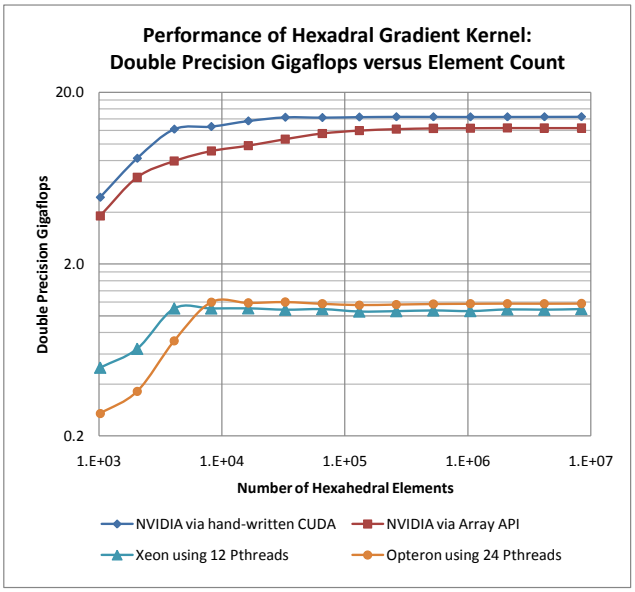

Figure 5. Initial performance results for linear hexadral basis function gradient computational kernel using the multidimensional array API

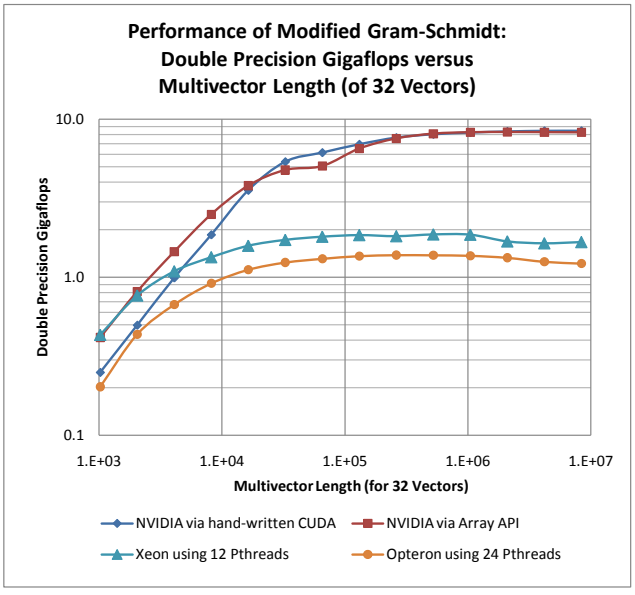

Figure 6. Initial performance results for Modified Gram Schmidt factorization sequencing simple vector kernels using the multidimensional array API

In addition to the three multicore / manycore device test cases, "native" CUDA versions of these test cases are implemented and run on the NVIDIA device. These hand-written CUDA test cases are used to compare performance

and usability with the portable array API versions. For the hexahedral gradient test case performance of the array-API version, for a large number of elements, is within 20% of the hand-written CUDA versions (Figure 5). A significant difference between the array-API and hand-written version is that the hand-written version eliminates all multi-index mapping calculations by leveraging *a priori* coding-time knowledge of array dimensions $(N, 3, 8)$ and NVIDIA coalesced memory access strides. In contrast the multi-index mapping of the array-API version use runtime knowledge of array dimensions, and selects the mapping computation at compile-time.

Performance differences between the array-API version and hand-written CUDA versions of the Modified Gram-Schmidt test case are small (Figure 6). This result is expected as the test case does not use a multi-index map calculation when indexing into the multivectors. Performances differences for smaller multivector lengths (*e.g.*, 1E3 to 1e5) are due to differences in the GPGPU reduction algorithms.

These initial results demonstrate the performance-feasibility of the Kokkos portable multidimensional array programming model; especially for an NVIDIA GPGPU device where improper memory access patterns (*i.e.*, non-coalesced memory access) will dramatically degrade performance.

VI. Research & Development In Progress

The Trilinos-Kokkos multidimensional array programming model and API provides a "classical" look-and-feel for computational kernels to use multidimensional arrays on multicore and manycore devices. Memory management within this programming model and API is non-traditional shared-ownership *views* semantics, as opposed to exclusive-ownership *container* semantics. View semantics are exclusively used to simplify the programming model and API, and mitigate risks of memory management errors.

The Trilinos-Kokkos multidimensional array programming model, API, and implementation is in its first year of research & development (R&D). Current R&D activities are focused on (1) investigating and improving performance for the current multicore and manycore devices and (2) demonstrating usability through more complex computational kernels and use within mini-applications. Anticipated R&D activities include (1) new device implementations such as the Intel Knights Ferry, (2) integration of multiple heterogeneous kernels within a single concurrent parallel_for or parallel_reduce operation, (3) integration into applications and libraries to support performance-portable multicore and manycore parallelism.

Trilinos-Kokkos is available in the public domain at http://trilinos.sandia.gov.

Acknowledgment

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

References

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

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

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

[4] Information Technology Industry Council, *Programming Languages — C++, International Standard ISO/IEC 14882*, 1st ed. 11 West 42nd Street, New York, New York 10036: American National Standards Institute, 1998.

[5] J. Reinders, *Intel Threading Building Blocks*. O'Reilly, Jul. 2007.