Kokkos: Enabling manycore performance portability through polymorphic memory access patterns

H. Carter Edwards*, Christian R. Trott, Daniel Sunderland

Sandia National Laboratories, PO Box 5800 / MS 1318, Albuquerque NM, 87185

Abstract

The manycore revolution can be characterized by increasing thread counts, decreasing memory per thread, and diversity of continually evolving manycore architectures. High performance computing (HPC) applications and libraries must exploit increasingly finer levels of parallelism within their codes to sustain scalability on these devices. A major obstacle to performance portability is the diverse and conflicting set of constraints on memory access patterns across devices. Contemporary portable programming models address manycore parallelism (*e.g.*, OpenMP, OpenACC, OpenCL) but fail to address memory access patterns. The Kokkos C++ library enables applications and domain libraries to achieve performance portability on diverse manycore architectures by unifying abstractions for both fine-grain data parallelism and memory access patterns. In this paper we describe Kokkos' abstractions, summarize its application programmer interface (API), present performance results for unit-test kernels and mini-applications, and outline an incremental strategy for migrating legacy C++ codes to Kokkos. The Kokkos library is under active research and development to incorporate capabilities from new generations of manycore architectures, and to address an growing list of applications and domain libraries.

Keywords: parallel computing, thread parallelism, manycore, GPU, performance portability, multidimensional array, mini-application

1. Introduction

The Kokkos C++ library provides scientific and engineering codes with a programming model that enables performance portability across diverse and evolving manycore devices. Our *performance portability* objective is to maximize the amount of user code that can be compiled for diverse devices and obtain the same (or nearly the same) performance as a variant of the code that is written specifically for that device. Performance portability is our primary objective for a high performance computing (HPC) programming model, and we address *usability* only within this constraint. Future usability studies will be conducted in conjunction with early adoption of Kokkos by applications and domain libraries.

The scope of Kokkos has evolved from a hidden portability layer for sparse linear algebra kernels [1] to a hierarchy of broadly usable libraries. Our earlier implementation of Kokkos' fundamental abstractions was referred to as *KokkosArray* [2, 3, 4]. These fundamental abstractions have persisted to the current version of Kokkos. The semantics, syntax, and implementation of Kokkos has significantly evolved in response to new device capabilities, performance evaluations, and usability evaluations through an expanding suite of mini-applications.

Our fundamental programming model abstractions are as follows:

1. Kokkos executes computational kernels in fine-grain data parallel within an *execution space*.
2. Computational kernels operate on multidimensional arrays residing in *memory spaces*.
3. Kokkos provides these multidimensional arrays with *polymorphic data layout*, similar to the Boost.MultiArray [5] flexible storage ordering.

*Corresponding author

Email addresses: hcedwar@sandia.gov (H. Carter Edwards), crtrott@sandia.gov (Christian R. Trott), dsunder@sandia.gov (Daniel Sunderland)

Kokkos enables computational kernels to be performance portable across manycore architectures (*i.e.*, CPU and GPU) by unifying these abstractions. A data parallel computational kernel's data access pattern can have a significant impact on its performance. On a CPU a computational kernel should have blocked data access pattern; however, on a GPU the computational kernel should have a coalesced data access pattern. This conflicting data access pattern requirement is commonly referred to as the *array of structures* (AoS) versus *structure of arrays* (SoA) problem. We solve the AoS vs. SoA performance portability problem by controlling the data parallel execution of computational kernels on a device, providing a multidimensional array data structure for those kernels to use, and choosing the multidimensional array layout that results in the required memory access pattern. Kokkos enables performance portable user code if that code is implemented with Kokkos' multidimensional arrays and parallel execution capabilities.

Many programming models control fine-grain parallel execution, as enumerated in Table 1. These programming models have a variety of implementation approaches: a library within a standard programming language, directives added to a standard language (*e.g.*, `#pragma` statements), language extensions supported by source-to-source translators, or language variants supported by a compiler. Among the programming models that we surveyed (Table 1), Kokkos is unique in that (1) it is purely a library approach, *and* (2) it enables portability to CPUs and GPUs, *and* (3) it provides polymorphic data layout. These three characteristics of our programming model are essential for performance portability and maintainability of HPC applications and domain libraries that must move to diverse and evolving manycore architectures.

Kokkos has thin *back-end* implementations that map portable user code to lower level, device specialized programming models. This software design allows us to choose the most performant back-end for each target device and optimize Kokkos' implementation for that back-end. Our current back-end implementations include CUDA [31] for NVIDIA GPUs, and pthreads [32] or OpenMP [30] for CPUs and Intel Xeon Phi. Pthreads and OpenMP back-ends optionally use the portable hardware locality (hwloc) library [33] for explicit placement of threads on cores. We use the Intel Xeon Phi co-processor in *self-hosted* mode, where processes run entirely on this device as opposed to using the offload model.

Table 1: Programming models for manycore parallelism.

Programming Model	Portable cpu/gpu	Data Layout	Approach
Kokkos [6]	yes	yes	Library
C++ AMP [7, 8]	yes	yes	Language
Thrust [9]	yes	no	Library
SGPU2 [10]	yes	no	Library
XKAAPI [11, 12]	yes	no	Library
OpenACC [13]	yes	no	Directives
OpenHMPP [14]	yes	no	Directives
StarSs [15]	yes	no	Directives
OmpSs [16, 17]	yes	no	Directives
HOMPI [18]	yes	no	Translator
PEPPHER [19]	yes	no	Translator
OpenCL [20]	yes	no	Language
StarPU [21, 22]	yes	no	Language
Loci [23, 24]	no	yes	Library
Cilk Plus [25]	no	yes	Language
TBB [26, 27]	no	no	Library
Charm++ [28, 29]	no	no	Library
OpenMP [30]	no*	no	Directives
CUDA [31]	no**	no	Language

* OpenMP 4.0 addresses GPU; however, compilers are not yet available.
** PGI compiler can generate x86 code from CUDA.

In this paper, we first describe Kokkos abstractions, API, and extension points. Then, we present performance results for unit-test kernels and mini-applications. Finally, we outline a strategy for legacy C++ codes to migrate to manycore devices.

2. Abstraction of a Manycore Device

Our abstraction of a modern HPC environment is a network of compute nodes where each compute node contains one or more manycore devices. A typical HPC application in this environment has at least two levels of parallelism: (1) distributed memory parallelism typically supported through a Message Passing Interface (MPI) library and (2) fine-grain shared memory parallelism supported through one of the many thread-level programming models.

In our abstraction, an MPI process has a single master thread that performs serial computations, calls MPI functions, and dispatches computational kernels for execution by worker threads of a manycore device. This "work dispatch" or "callback" pattern is common in numerous programming models; for example, function objects are dispatched to C++ Standard Template Library (STL) algorithms [34], TBB, and Thrust. A key element of this abstraction is that a master thread executes on the CPU and worker threads execute on a manycore device. When the master and worker threads execute on a multicore CPU or self-hosted device (how we use the Intel Xeon Phi) then the device is the CPU.

2.1. Execution and Memory Spaces

Threads execute in an *execution space* and data resides a *memory space*. For example, the master thread performs serial computations in the CPU's execution space and operates on data in the CPU's memory space. Similarly, worker threads call computational kernels in a device execution space, and these kernels operate on data in the device memory space.

An execution space has accessibility and performance relationships with memory spaces. For example, a computation in the CPU execution space may be prohibited from accessing data residing in a CUDA memory space. Similarly a computation in a CUDA execution space could access data in a *host-pinned* memory space (a capability supported by newer CUDA-GPU devices) with degraded performance compared to that computation accessing data in the CUDA memory space. A programming model for manycore architectures should include semantics to expose execution spaces, memory spaces, and relationships among these spaces.

2.2. Abstracting Spaces

In the Kokkos API, each execution space and memory space is defined by a unique C++ class. This API enables enforcement of execution-memory space accessibility constraints at compile-time. For example, the master thread in the CPU execution space is prevented from accessing memory in the CUDA memory space, as opposed to generating a runtime memory fault. When devices provide virtual unified addressing across memory spaces (*e.g.*, NVIDIA's host-pinned memory capability), we will define additional memory spaces and the associated execution-memory space relationship.

Our abstraction for execution and memory spaces is an *extension point* in Kokkos' design for expressing and managing increasing complex device architectures. For example, NVIDIA devices have global and shared memory spaces with different performance characteristics. We believe that this execution-memory space abstraction and extension point is critical for "future proofing" codes.

3. Multidimensional Array

A Kokkos *multidimensional array* consists of: (1) a set of datum $\{x_\iota\}$ of the same value type and residing in the same memory space, (2) an index space X_S defined by the Cartesian product of integer ranges, and (3) a *layout* X_L – a bijective map between the index space and the set of datum. (Note that equality of datums' values does not imply the *same* datum: $x_\iota = x_\kappa \not\Rightarrow \iota = \kappa$.)

$$\begin{array}{lcl} X & = & (\{x_\iota\}, X_S, X_L) \\ X_S & = & [0..N_0) \times [0..N_1) \times \cdots \\ X_L & : & X_S \leftrightarrow \{x_\iota\} \end{array}$$

A function typically contains a sequence of nested loops over dimensions of an array X_S and accesses array datum via the layout X_L. Thus, the composition of a function's loop-and-indexing pattern and the array's layout yields a memory access pattern.

$F[X_S]$	function implemented for X_S
$F[X_S] \circ X_L^{-1}$	resulting memory access pattern

To modify the function's memory access pattern one must either (1) change the dimension ordering X_S and the loop-and-indexing pattern of the function $F[X_S]$ or (2) change the layout of the array X_L.

Programming languages with built-in multidimensional arrays have a prescribed layout. For example, FORTRAN and C languages prescribe layouts that are reversed with respect to a similarly declared index space. When using a language's built-in array, a function's memory access pattern can only be changed by reordering dimensions and changing both loop and indexing patterns. Such a function must have distinct, device-dependent versions to satisfy each memory access pattern required by diverse manycore devices.

Layout Polymorphism. Kokkos array layouts (X_L) are chosen at compile time. Thus, a function's memory access pattern ($F[X_S] \circ X_L^{-1}$) can be changed without modifying that function's code. Layout polymorphism requires a function to strictly adhere to Kokkos' mapping operator – the function does not bypass the Kokkos API nor assumes a particular layout. Layout polymorphism enables Kokkos to choose array layouts that lead to device-appropriate memory access patterns. Choosing layouts at compile-time allows back-end compilers to in-line or optimize layout mapping computations within a function.

3.1. Declaration, Allocation, and Access

Kokkos arrays are implemented with a C++ template class named `View` (we explain why this name was chosen in the next section). The design pattern of encapsulating multidimensional array semantics in a C++ class, or even a C struct, has been fundamental to well-engineered numerically-oriented C and C++ libraries for decades [35, 36]. We tailor this pervasive design pattern to compactly declare multidimensional array dimensions, identify memory spaces for the datum, specify layouts, and annotate behavioral traits.

Array declaration, allocation, and value access operations are illustrated in Figure 1.

```
// The View constructor allocates an array
// in Device memory space with dimensions
// N*M*8*3, where each '*' token denotes a
// dimension to be supplied at runtime.
// The label "A" is used in error messages
// which may occur in regard to this array.
View<double**[8][3],Device> a("A",N,M);

// The parentheses operator implements the
// layout map.
a(i,j,k,l) = value ;
```

Figure 1: The fundamental declaration, allocation, and access operators shown here are designed to be compact, intuitive, and strictly compliant with C++ language standards.

The first `View` template argument specifies the value type, number of dynamic dimensions (denoted by the number of '*' tokens), and static dimensions denoted by '[#]' expressions. The second template argument defines the memory space in which the values of the array are allocated. The `View` constructor allocates memory according to the layout chosen for that space, static dimensions, and dynamic dimensions input to the constructor. The `View` parentheses operator implements the layout, enforces execution-memory space relationships, and optionally enforces index space bounds (used when debugging code).

Unconventional Syntax. The syntax using '*' tokens and '[#]' expressions to declare array dimensions is unconventional with respect to other multidimensional array APIs. The driving factor in this API design was to allow a mix of static and dynamic dimensions. Performance testing, early in development of Kokkos, showed that layout computations can be optimized in the presence of static dimensions, and that such an optimization can have a significant impact on the performance of computational kernels. The combined goals of compact notation and mixed dynamic and static dimensions, and strict conformance to C++ syntax standards, led to this unconventional API. The potential confusion of interpreting a dynamic dimension ('*' token) to mean that a value type is a pointer has not been an issue. This is because our numerical kernels have consistently been clearer and perform better when they do not "pointer chase," and instead use indices for indirect addressing. Furthermore, we want to discourage the use of pointer chasing as it impedes a compiler's ability to optimize computations.

Const-ness. The "const-ness" semantics of `View`s is analogous to the "const-ness" semantics for C++ pointers, as illustrated in Figure 2. Just as a `const` pointer cannot be reassigned a `const View` cannot be reassigned. However, the memory referenced by a `const` pointer can be modified, likewise the datum referenced by a `const View` can be modified. A "pointer to `const`" is different, it declares that what the pointer references cannot be modified. A "`View` to `const`" is declared by associating the `const` keyword with the value type of the `View`. Datum referenced by a `View` to `const` cannot be modified.

```
typedef double *       PtrT ;
typedef double * const ConstPtrT ;
typedef const double * PtrToConstT ;

typedef View<double*,Device>       ViewT ;
typedef const View<double*,Device> ConstViewT ;
typedef View<const double*,Device> ViewToConstT ;

// const-ness is enforced at compile-time:
ViewT        x("myx",N); // allocate
ConstViewT   y ;
ViewToConstT cx = x ;  // this assignment is OK
ViewT        e  = cx ; // error: violate const-ness
             y  = x ;  // error: reassign const
```

Figure 2: The "const-ness" semantics of Views and pointers are analogous. This is illustrated by similar declarations for pointer and View types and "const-ness" violations

3.2. View and Deep Copy Semantics

The class name `View` was chosen to inform and remind users that `View` objects have shared ownership semantics as shown in Figure 3. In contrast to C++ standard container semantics, multiple `View` objects can reference the same allocated array. `View` semantics are analogous to

C++ `std::shared_ptr` semantics where allocated memory is deallocated when the last view of that memory is destroyed or reassigned.

```
typedef View<double**[8][3],Device> my_type ;

typedef View<const double**[8][3],Device>
  my_const_type ;

my_type a("a",N,M); // Allocate an array

{ // Begin a nested scope
  // Create more views of the same array
  my_type       a2 = a ;  // shallow copy assignment
  my_const_type a3 = a2 ; // compatible shallow copy

  // 'a' and 'a2' are cleared (set to 'null')
  a  = my_type();
  a2 = my_type();
  // 'a3' still views the array

} // The View destructor is called on 'a3'.
  // As the last view, it deallocates the array.
```

Figure 3: Shared ownership semantics are illustrated with multiple views of the same allocated array being created and cleared. The last cleared or destroyed view is responsible for deallocating the array.

The `View` assignment operator is a *shallow copy* operation – only the reference to allocated memory and layout metadata are copied. Kokkos provides *deep copy* functions to copy allocated values between two arrays. A deep copy operation is most often used to copy array values between memory spaces, from CPU to device and vice versa.

Deep copies between arrays with different layouts (or index spaces) have a performance penalty of remapping data between the layouts, and an additional performance penalty of allocating a temporary array when copying between different memory spaces. Recall that Kokkos chooses a layout (by default) for the array's device – thus a GPU and CPU have different layouts. We address this performance issue by defining `HostMirror` view types as shown in Figure 4. A `HostMirror` defines a view that has the device's array layout but allocates memory in the CPU memory space. Thus, deep copies between a view and its host mirror never require remapping, and they can be implemented by the most efficient memory-to-memory copy capability of the device.

3.3. Performance Tuning Extension Points

In the previous sections we described fundamental capabilities for declaring, allocating, accessing, and managing arrays in specified memory spaces with device dependent layouts. Kokkos supports array access performance tuning features through

```
typedef View<double**[8][3],Device> my_array_type;

my_array_type a("a",N,M); // Allocate on Device

// my_array_type::HostMirror defines an array
// in CPU memory space with a layout mirroring
// my_array_type.  If the device != Host then
// create_mirror_view allocates a compatible array,
// otherwise a view of the same array is returned.
my_array_type::HostMirror host_a =
                          create_mirror_view( a );

// Deep copy to a mirror does not require remap.
// If a == a_host deep copy is skipped.
deep_copy( a , host_a ); // Copy device <- host
deep_copy( host_a , a ); // Copy host <- device
```

Figure 4: Deep copy operations between memory spaces can lead to remapping operations. This performance penalty is avoided by using `HostMirror`, a view with the device layout but values in the CPU memory space.

an optional advanced API. These features leverage the extension points in Kokkos' software design.

Overriding the Layout. Kokkos chooses a default layout for a given device. A user may override this default layout through an optional template argument on the `View` class. In addition, an advanced user may develop new layouts for their arrays.

Figure 5 presents an example of a `View` with a tiled matrix layout that is (for example) used by PLASMA [37]. With this layout, the parentheses operator maps input indices to a tile and then to a value within that tile. An application or library that implements matrices with the `View` class can change from a traditional column major layout to a tiled layout simply by introducing a template parameter. Existing code that did not assume a particular layout will not have to be modified and will produce the same results. At this point, new layout-specific code can be introduced to further improve performance of computation kernels.

Behavioral Traits. The `View` class has an optional behavioral trait template parameter as a second extension point. A user can use behavior traits to inform a Kokkos back-end to utilize device-specific capabilities. For example, NVIDIA devices have a texture cache which can be utilized to improve performance for random accesses that frequently read values. This portable interface for utilizing NVIDIA texture cache is illustrated in Figure 6.

Other behavioral traits could include non-temporal hints (*e.g.*, `StreamingLoad` and `StreamingStore`) to avoid cache pollution when data is not reused. This behavioral trait extension

```
// Old matrix type:
// typedef View<double**,Device> my_matrix ;

// Change matrix type to an 8x8 tiled layout.
typedef View< double** ,
              LayoutTileLeft<8,8> ,
              Device >  my_matrix ;

my_matrix A("A",N,N); // Allocation is unchanged.

value = A(i,j); // Indexing unchanged.

// New layout-leveraging code can be introduced
// to optimize performance.  Such code should be
// protected via template partial specialization.
// tile_type is View<double[8][8],LayoutLeft,Device>
my_matrix::tile_type t = A.tile(iTile,jTile);
```

Figure 5: A default array layout may be overridden through an optional `View` template arguments. In this example, a view is specified to have an 8x8 tiles layout. Existing layout-agnostic code is unchanged and specialized code leveraging the tiled layout can be introduced.

```
// Allocate an array with an overridden layout.
typedef View< double ** ,
              LayoutRight ,
              Device > x("x",N,M);

// Define a compatible view with
// const value type and RandomRead trait.
typedef View< const double** ,
              LayoutRight ,
              Device ,
              RandomRead >  read_x = x ;

// If Device is CUDA then this operator
// uses NVIDIA texture cache capability.
value = read_x(i,j);
```

Figure 6: The parentheses operator of a View with `const` value type, `Cuda` device, and `RandomRead` trait is implemented with the NVIDIA texture cache capability. If any of these three conditions are not satisfied then the standard parentheses operator implementation is used.

point is expected to gain importance to portably take advantage of future architectures' increasingly complex memory subsystems.

Aggregate Data Types. An advanced computational software strategy is to embed sensitivity or uncertainty quantification computations into computational kernels by replacing intrinsic scalar data types (*e.g.*, `float` or `double`) within the kernel with aggregate data types [38, 39, 40]. For example, replacing a variable's scalar data type with an automatic differentiation data type embeds derivative calculations for that variable without having to modify the remainder of the kernel's source code. Replacing an intrinsic scalar type with an aggregate type will alter the memory access pattern of a computation, potentially resulting in a loss in performance. Even worse, when the size of the aggregate type is defined at runtime (*i.e.*, the degree of automatic differentiation), then an implementation of that aggregate type may cause frequent small memory allocations and deallocations which are detrimental performance.

The `View` class incorporates aggregate data types into the data layout. In the example shown in Figure 7, the `AutoDeriv` class contains a scalar value and its derivatives. The `View` implementation includes this aggregate data type as an additional (mostly) hidden dimension in the index space and polymorphic layout. In this example the parentheses operator returns a reference to the aggregate data type as if the additional dimension did not exist.

```
// Replace the 'scalar' type with
// automatic differentiation type.
typedef View< AutoDeriv<double>**, Device> my_type;

// Allocate an array with the runtime-defined
// degree of differentiation dimension.
my_type x("x",N,M,nDeg);

// Parentheses operator returns a view that is
// compatible with the AutoDeriv<double> type.
AutoDeriv<double> value = x(i,j);
```

Figure 7: A View instantiated with an aggregate pseudo scalar type of a dynamic size can incorporate that length as an additional dimension that is hidden from the parentheses operator but must be supplied to the constructor for correct allocation.

The capability to embed aggregate data types within a `View` is a design extension point to support advanced computational software strategies such as embedding automatic differentiation or stochastic bases data types.

4. Parallel Execution

Parallel execution patterns [41] are divided into two categories: (1) data parallel or single instruction multiple data (SIMD) and (2) task parallel or multiple instruction multiple data (MIMD). Kokkos currently implements data parallel execution with `parallel_for`, `parallel_reduce`, and `parallel_scan` operations. The `parallel_scan` operation was implemented after initial submission of this paper and is not described here. Research and development is in progress for hierarchical task-data parallelism where interdependent data parallel tasks are scheduled to execute on the device.

A data parallel operation maps `NWork` units of works onto threads that execute on the device.

Units of work are independent if they do not write, or write-and-read, the same data. For example, adding two vectors of length N can be performed in parallel by independently adding its N members. Units of work may be dependent by updating the same data via global or local reduction operations. Kokkos supports deterministic global reductions (*e.g.*, an inner product) with the `parallel_reduce` operation and local reductions (*e.g.*, a map reduce) with atomic updates. Atomic updates must be used cautiously as they will introduce non-deterministic behavior, and may lead to race conditions.

A data parallel computational kernel is currently implemented as a *functor*. A functor is a C++ class that contains one or more callback functions, shared parameters, and references to data upon which the callback function operates. The C++11 standard introduced *lambda* language feature which can significantly improve the syntax and usability of the functor pattern. Extension of Kokkos to accept lambda-based computational kernels will be straight-forward when vendor support for lambdas is sufficient.

4.1. Parallel For

A `parallel_for` functor is a C++ class that contains a work callback function, shared input parameters, and views to arrays upon which the callback function operates. The functor's work callback function is called to perform `NWork` independent units of work where each unit is identified by a unique *work index* in the range `[0..NWork)`. Default array layouts are chosen assuming that the left-most index of an array is the parallel work index.

The API of a `parallel_for` functor has two simple requirements illustrated in Figure 8: to (1) identify the execution space and (2) provide a work callback. We recommend that a functor's class be templated on the execution space for device portability. This allows a functor to be compiled for two different devices in the same executable to enable hybrid execution.

4.2. Parallel Reduce

A `parallel_reduce` functor has a work callback, a reduction callback, shared input parameters, views to arrays upon which the work callback operates, and reduction parameters. Each call to a `parallel_reduce` work callback generates a contribution to the reduction parameters that must be reduced by a commutative and *mathematically* associative reduction callback. The numerical implementation of a reduction callback could be non-associative due to numerical round-off in floating point operations.

```
// Template on the Device for portability.
template< class Type , class Device >
class AXPY_Functor {
public:
  // Requirement: Identify execution space.
  typedef Device device_type ;

  // Requirement: Provide work callback as
  // 'void operator()( integer_type iw ) const'
  // where 'iw' is the work index.
  // KOKKOS_INLINE_FUNCTION is a #define macro
  // for compiler directives such as
  // 'inline __device__' for Cuda.
  KOKKOS_INLINE_FUNCTION
  void operator()( int iw ) const
    { y(iw) = alpha * x(iw) + y(iw) ; }

  const View<      Type*,Device> y ;
  const View<const Type*,Device> x ;
  const Type alpha ;
};

// Call the functor NWork times on up to NWork
// worker threads.  Each call is passed a unique
// work index in the range [0..NWork).
parallel_for( NWork ,
  AXPY_Functor<double,Cuda>( a , X , Y ) );
```

Figure 8: Interface requirements for parallel_for functors are illustrated with this example AXPY functor that performs "$Y = \alpha X + Y$" on arrays. The trivial constructor for initializing `alpha`, `x`, and `y` data members is omitted.

The API requirements for a `parallel_reduce` functor are illustrated in Figure 9. These requirements are: (1) identify the execution space, (2) identify a `value_type` for the reduction parameters (the result), (3) provide a work callback, and (4) provide two reduction callback functions. The reduction parameter `value_type` must satisfy the *plain old data type* conditions; *e.g.*, a plain memory-copy of values will have the correct result. This type is typically a simple intrinsic value such as `double`. The example `value_type` in Figure 9 is a struct to illustrate the need for reductions of non-intrinsic types. The reduction callback functions have two responsibilities: initialize a temporary reduction value to the appropriate value (*e.g.* zero) and join two reduction values into a single value.

The `parallel_reduce` functor API requirements are defined so that Kokkos can provide scalable and deterministic global reductions. For large thread counts the global reduction follows a traditional $\log_2(NT)$ fan-in algorithm (NT = number of threads). The fan-in algorithm requires thread-local copies of the reduction parameters which are

```
template< class Scalar , class Device >
class CentroidFunctor {
public:
  // Required: Identify execution space.
  typedef Device device_type ;
  // Required: Identify reduction parameters
  typedef struct { Scalar point[3], mass ; }
    value_type ;

  // Required: Work callback contributes to the
  // reduction via the 'update' argument.
  KOKKOS_INLINE_FUNCTION
  void operator()( int iw ,
                   value_type & update ) const
  { update.mass       += mass(iw);
    update.point[0:2] += mass(iw)*point(iw,0:2); }

  // Required: Reduction callback to join 'update'
  // with 'input' from a different thread.
  // These arguments are 'volatile' to force
  // communication of values among threads.
  KOKKOS_INLINE_FUNCTION
  static void join( volatile value_type & update ,
              const volatile value_type & input )
    { /* update += input */ }

  // Required: Reduction callback to initialize
  // temporary reduction parameters' values.
  KOKKOS_INLINE_FUNCTION
  static void init( value_type & update )
    { /* update = 0 */ }

  View<Scalar*,   Device> mass ;
  View<Scalar*[3],Device> point ;
};

// Reduction 'value_type' is output in 'result'.
parallel_reduce( NWork,
  CentroidFunctor<double,Cuda>(mass,point), result);
// Final serial step for centroid computation.
result.point[0:2] /= result.mass ;
```

Figure 9: Interface requirements for `parallel_reduce` functors are illustrated with an example computation of the mass weighted sum of points. Such a functor is the parallel portion of a centroid computation. Portions of this example have been abbreviated; *e.g.*, a loop from 0 to 2 has been abbreviated with 0:2.

reduced to a single global value through a defined sequence of concurrent pair-wise reductions. This sequence is derived from the number of threads, NT, and number of work items, `NWork`, and guarantees a deterministic result when given the same NT and `NWork`.

4.3. Local Parallel Reductions via Atomics

Local parallel reductions are supported through atomic reduction operations; *e.g.*, atomic addition. An atomic operation serializes concurrent updates to a datum but does not guarantee the ordering of these updates among threads. Thus a non-associative local reduction operation (*e.g.*, floating point addition) is likely to yield nondeterministic results for local parallel reductions.

Atomic operations' serialization will introduce scalability bottlenecks when there are too many concurrent atomic reductions; where "too many" is dependent upon the number of threads and the device's capabilities. Typically atomic operations should only be used when the number of atomic updates to a particular datum is much smaller than the number of work items. Otherwise functors with reductions should be implemented with atomic-free algorithms where feasible, such as using parallel_reduce.

4.4. Threaded Scalability and Performance

The composition of parallel work dispatch and polymorphic array layout capabilities enables performance portable implementations of parallel algorithms. Atomic operations supports thread-safe implementations of algorithms with local parallel reductions, which could be performant given an adequate ratio of computation to atomic operations and a low frequency of collisions. However, Kokkos cannot automatically make serial algorithms, or algorithms with serial bottlenecks, scalable with respect to the number of threads. Such a computation will require a redesign of its algorithm to achieve threaded scalability.

One such kernel is the molecular dynamics force computation evaluated in Section 5.2. The original implementation of this algorithm was optimized for a non-threaded environment and thread-safety was not a concern. Migration of this kernel to a threaded environment (described in Section 7.4) required a different thread-scalable algorithm that performs 2x redundant computations but is now fully scalable to thousands of concurrent threads.

Threaded scalability is necessary to effectively utilize devices' increasing core counts and decreasing memory per thread. However, this may not result in the best achievable performance for a particular computation on a given device with its particular capabilities and limitations. A computation that is critical for an application's performance may require tailoring to work around limitations of a device or leverage device specific capabilities. This situation was encountered during the migration of the molecular dynamic force computation described in Section 7.4. Fortunately the required algorithmic specialization for this particular computation had no negative impact across architectures and was applied to the portable version of the computation. Based upon this experience we recommend that the performance of threaded algorithms be evaluated across diverse architectures (which Kokkos facilitates with its portable API) and with large thread counts.

Kokkos supports device specific specializations of functors through C++ template partial specialization – a functor that is templated with respect to the device can have a device-specific implementation that is automatically and transparently picked up by the C++ compiler. Such specializations are most frequently used to replace portable implementations of commonly used functions with calls to equivalent functions in device-optimized libraries, typically provided by the device's vendor. Such a specialization is described in Section 7.5.

5. Performance Evaluation with Simple Kernels

We evaluate Kokkos performance with simple kernels and mini-applications (Section 6). Performance testing is carried out on our Compton and Shannon testbed clusters. Compton is used for Intel Xeon and Intel Xeon Phi tests, and Shannon is used for NVIDIA Kepler (K20x) tests. Testbed configuration details are given in Table 2. Note that in these configurations *device* refers to a dual socket Xeon node, a single Xeon Phi, and a single Kepler GPU respectively.

Results presented in this paper are for pre-production Intel Xeon Phi co-processors (code-named Knights Corner) and pre-production versions of Intel's Xeon Phi software stack. Performance and configuration of the co-processors may be different in final production releases.

5.1. Modified Gram-Schmidt Kernel

The modified Gram-Schmidt (MGS) algorithm orthonormalizes a collection of vectors through a sequence of inner products and scaled vector addition computations. These computations are performed with a sequence of `parallel_reduce` and `parallel_for` operations. The ratio of floating point operations to memory access operations is approximately 2/3. Thus, performance is limited by memory bandwidth reading and writing vectors and the overhead of dispatching and synchronizing parallel operations. The collection of M vectors of length N is stored in contiguous memory with padding between vectors for appropriate memory alignment.

For a given $N * M$ problem size the MGS net bandwidth, B_{mgs}, is a function of the read / write memory bandwidth (assumed to be equal), B, and the parallel dispatch-synchronization time, S. The MGS algorithm reads and writes vectors multiple times. When the problem size is small enough for some of the M vectors to reside in the Xeon's L3 cache memory then reads and writes of those vectors will occur at cache memory bandwidth, as opposed to main memory bandwidth. For the MGS algorithm, memory bandwidth is a function of problem size, $B(N, M)$, ranging from L3 cache memory bandwidth for small problem size to main memory bandwidth for large problem size. For the Kepler and Xeon Phi we assume B is constant.

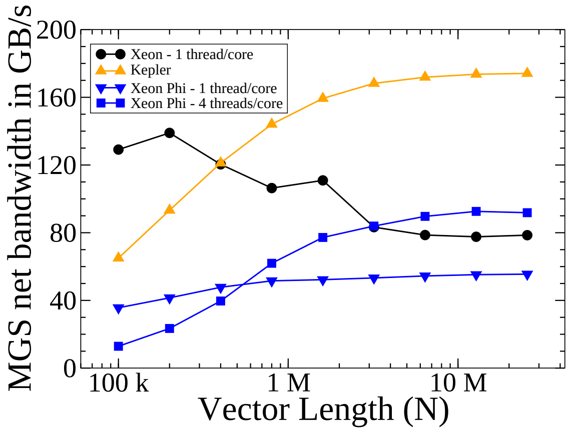

Figure 10: Modified Gram-Schmidt algorithm's net bandwidth on NVIDIA Kepler (K20x), Intel Xeon, and Intel Xeon Phi for $M = 16$ double precision vectors. For large vectors performance is limited by bandwidth to main memory. For small vectors performance is limited by a dispatch-synchronization time, or bandwidth to L3 cache memory for the Xeon.

We derive the following simple performance model for B_{mgs} (b = byte size of a vector element).

$$B_{mgs} \approx \left(\frac{1}{B(N,M)} + \frac{2S}{5Nb} \right)^{-1}, \; 1 \ll M \qquad (1)$$

For small N, performance is limited by the parallel dispatch-synchronization time. For large N, performance approaches bandwidth to global memory as vectors are no longer cache resident, and the global memory read+write time dominates on all devices.

The MGS net bandwidth was measured with 16 double precisions vectors over a range of vector lengths N, as presented in Figure 10. For large N (limited by bandwidth to main memory) we achieve 174 GB/s or 78% of peak on the Kepler, 78 GB/s or 71% of peak on the Xeon, and 92 GB/s or 46% of peak on the Xeon Phi using 4 threads/core. On

Table 2: Configurations of testbed clusters.

Name	Compton	Shannon
Nodes	32	32
CPU	2x Intel E5-2670 HT-on	2x Intel E5-2670 HT-off
Co-Processor	2x Intel Xeon Phi 57c 1.1GHz	2x K20x ECC on
Memory	64 GB	128 GB
Interconnect	QDR IB	QDR IB
OS	RedHat 6.1	RedHat 6.2
Compiler	ICC 13.1.2	GCC 4.4.6 + CUDA 5.5 RC
MPI	IMPI 4.1.1.036	MVAPICH2 1.9
Other	MPSS 2.1.3653-8	Driver: 319.23

Kepler and Xeon Phi small N performance is limited by the parallel dispatch-synchronization as expected. On Xeon, where the core-count is low and the small N problem is fully cache resident, the increase in bandwidth compensates for the dispatch-synchronization time, resulting in higher net bandwidth.

We fit Equation 1 to the data in Figure 10. Resulting parameters for the bandwidth $B(N, M)$ are 54.9 GB/s, 96.9 GB/s, and 176.2 GB/s respectively on Xeon Phi with 1 thread per core, Xeon Phi with 4 threads per core and the K20x Kepler GPU. The corresponding average dispatch-synchronization times are 21.1 μs, 108.0 μs, and 20.0 μs per kernel, most of which is likely spend for the reduction in the dot products.

We use the MGS test case as a tool to identify, quantify, and improve performance with respect to a devices' dispatch-synchronization capabilities, or Kokkos back-end's use of those capabilities. As demonstrated in Figure 10, when the Xeon Phi is fully populated with threads (4 threads/core) the dispatch-synchronization time is noticeably larger for small N. However, reducing thread count results in significant performance loss for large N problems. Thus improvement of Kokkos' dispatch-synchronization performance on Xeon Phi and Kepler K20x is an active research focus.

5.2. *Molecular Dynamics Force Kernel*

The Lennard Jones molecular dynamics force kernel (LJ-kernel), shown in Figure 11, loops over atoms and calculates the forces between pairs of atoms with a distance d_{ij} smaller than a cutoff r_{cut}. Performance of this algorithm is improved by a search phase that, for each atom i, precomputes a list of neighbor atoms that will likely be within that cutoff radius. In Figure 11 this list is implemented by the two dimensional `neighbors` array.

```
// Parallel iteration of all atoms in the system
for(i=0;i<natoms;i++) {
  double x_i[3], f_i[3];
  x_i[0:2] = x(i,0:2);
  f_i[0:2] = 0;
  // Iterate the precomputed list of neighbors
  for(jj=0;jj<num_neighbors(i);jj++) {
    int j = neighbors(i,jj);
    double d_ij[3] , d ;
    d_ij[0:2] = x_i[0:2] - x(j,0:2);
    d = norm(d_ij);
    if(d<r_cut) {
      const double sr2 = 1.0/(d*d);
      const double sr6 = sr2*sr2*sr2;
      const double force = 48.0*sr6*(sr6-0.5)*sr2;
      f_i[0:2] += force * d_ij[0:2];
    }
  }
  f(i,0:2) = f_i[0:2];
}
```

Figure 11: Pseudo code for the thread safe Lennard Jones molecular dynamics kernel (LJ-kernel) using full neighbor lists implemented in MiniMD. Note that the 0:2 abbreviates replication of the statement with 0, 1, and 2 as the index.

The LJ-kernel test case parameters are presented in Table 3. In this test case there are, on average, 77 neighbors of which 55 pass the distance check. This results in an average of 1408 Flops and 311 memory accesses per atom i. In the LJ-kernel, `neighbors(i,jj)` has a regular memory access pattern and `x(j,0:2)` has a random memory access pattern. However, when atoms are ordered according to spatial locality it is possible to achieve high cache reuse for random reads of `x(j,0:2)`. This reuse, of up to 77 times on average, drastically reduces the actual number of loads from main memory.

Table 3: LJ-Kernel and MiniMD test problem configuration.

Atoms (LJ-kernel test)	864,000
Atoms (miniMD test)	2,048,000
Units	Lennard Jones
Density	0.8442 σ_0^{-3}
Initial Temperature	1.44
Initial Config	FCC lattice
Force cutoff	2.5 σ_0
Neighbor cutoff	2.8 σ_0
Neighbor type	full
Reneighboring every	20
Sorting every	20
Thermo calculation every	100
Threads on Xeon	32
Threads on Xeon Phi	224

The importance of polymorphic array layouts is highlighted in reading the neighbor index array, `neighbors(i,jj)`. On a CPU or Xeon Phi this array should have row major ordering so that a read cache line contains values for the next iteration step of the inner loop over `jj`. On a GPU this array should have column major ordering so that the read is coalesced for threads working on different atoms i. In addition, on a GPU the random read performance for `x(j,0:2)` is significantly improved through the use of texture cache.

Figure 12 shows gigaflops/second performance of this kernel applied to a 864,000 atom problem on a single node of the Compton and Shannon testbeds (Table 2). We run this performance test with the appropriate layout and use of GPU texture cache, and compare to results obtained from manually forcing Kokkos to use the wrong layout and not use GPU texture cache. Note that the wrong layout on a CPU is the appropriate layout on a GPU, and vice versa. Forcing the wrong layout causes a performance drop of 1.9x, 3.4x, and 6.6x on the CPU, Xeon Phi, and GPU testbed nodes respectively. In addition, using the correct layout but not using texture cache results in a 3.6x slowdown for on the GPU node.

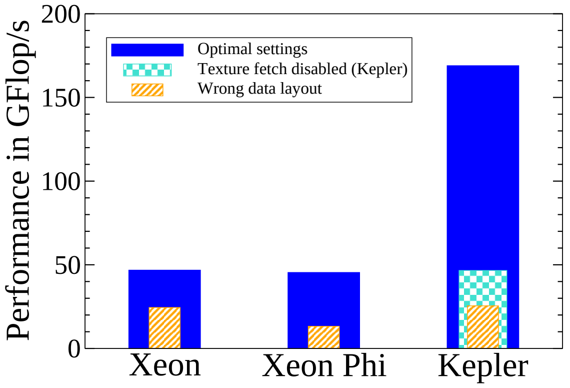

Figure 12: LJ-kernel performance in miniMD on Intel Xeon CPU, Intel Xeon Phi, and an NVIDIA Kepler GPU for the miniMD default test problem with 864,000 atoms. The solid blue bars show performance with device-appropriate data layout, the striped orange bars show performance with the wrong data layout for the `neighbors` array, and the green checkerboard bar shows performance on the GPU with the correct layout but without using texture cache.

6. Performance Evaluation with Mini-Applications

6.1. MiniFE

MiniFE is a hybrid parallel (MPI+X) finite element mini-application that (1) constructs a linear system of equations for a 3D heat diffusion problem and (2) performs 200 iterations of a conjugate gradient (CG) solver on that linear system. This mini-application is designed to capture important performance characteristics of an implicit parallel finite element code. MiniFE has been implemented in numerous programming models, some of which are available through the Mantevo suite of mini-applications[1] available at **mantevo.org**.

We compare the performance of miniFE implemented with Kokkos (miniFE-Kokkos) with miniFE implemented directly with OpenMP (miniFE-OpenMP) and CUDA (miniFE-CUDA). The miniFE-Kokkos back-end used on Xeon and Xeon Phi nodes is OpenMP, and the back-end use on Kepler GPUs is CUDA. The miniFE-CUDA variant is based upon miniFE-Kokkos where all linear algebra subprogram functions are replaced with calls to NVIDIA's cuBLAS and cuSparse functions. Both miniFE-Kokkos and miniFE-OpenMP are part of the 2.0 release of miniFE. The majority of miniFE optimization efforts have concentrated

[1]Recipient of a 2013 R&D 100 Award, www.rdmag.com/award-winners/2013/07/2013-r-d-100-award-winners

on the CG-solver; therefore, we limit performance testing to this phase of miniFE execution.

Our miniFE test case is a weak scaling problem with 8M elements per device, consuming 3.3GB of main memory per device. Tests are run with a single MPI process per device, except for miniFE-OpenMP tests on the dual socket Xeon nodes which are run with only one MPI process per socket. We make this exception because the execution time more than doubles when miniFE-OpenMP is run with one MPI process per node. This slowdown is a result of the problem construction phase performing an implicit nonuniform memory access (NUMA) first touch on the linear system that is incompatible with the subsequent access pattern during the CG-solve phase. Consequently, during the CG-solve phase threads access memory in the wrong NUMA domain and incur the associated cross-NUMA bandwidth penalty. Kokkos transparently handles this NUMA issue by running a `parallel_for` first touch initialization on each new allocation, where this first touch is compatible with subsequent data parallel kernels. For tests on the Kepler GPU, we use the MVAPICH2 1.9 [42, 43] implementation of MPI so that MPI can directly access GPU memory via CUDA GPU-Direct. Thus, explicit data copies are not necessary between device and CPU during the CG-solve.

Weak scaling performance of the miniFE CG-solve phase are shown in Figure 13. These tests are run on the testbeds (Table 2). For each data point, the best time out of twelve runs was used.

Overall, miniFE-Kokkos delivers similar performance as the native implementations of miniFE. It is faster than miniFE-CUDA in Kepler tests by roughly 13%, it is marginally slower than miniFE-OpenMP in Xeon tests, and it is about 10% slower than miniFE-OpenMP on Xeon Phi. Excellent weak scaling is observed on both on Xeon and Kepler GPU test-beds, where miniFE-Kokkos has about 95% parallel efficiency with 32 MPI ranks. MiniFE-OpenMP shows slightly worse scaling efficiency, which is likely due to using two MPI ranks per CPU node. The scaling issue on Xeon Phi can be attributed to the poor MPI performance on our Xeon Phi testbed. Peak bandwidth between two Xeon Phi co-processors is as low as 300 MB/s if at least one of the co-processors sits in a socket without an Infiniband adapter. In comparison, the Xeon and Kepler GPU runs' peak MPI bandwidth is about 3.5 GB/s. This Xeon Phi MPI issue is expected to be solved with a new runtime software

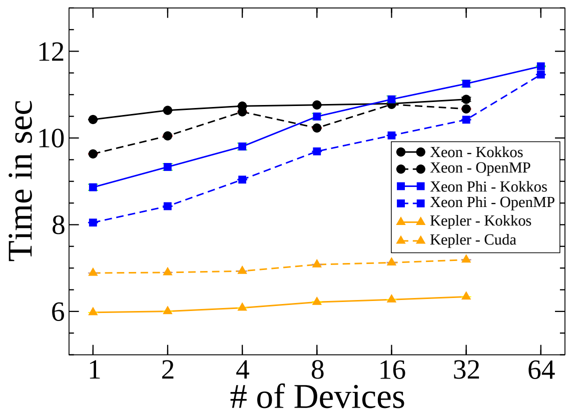

Figure 13: Time for CG-solve 200 iterations with miniFE variants on different testbeds. The problem size is weak scaled, with 8M elements per device. The solid lines represent runs using miniFE-Kokkos, while the dashed lines show results with peer variants. For each data point the best time out of 12 runs was used.

stack. At that point a Xeon Phi based systems should see similar scaling behavior as the Kepler GPU based system.

When repeatedly running the same performance test on the Xeon Phi there are occasional outliers in the runtimes. These outliers are almost twice as long as the mean time, without the outliers. The underlying cause of this performance anomaly is under investigation.

As previously mentioned, GPU-Direct capabilities were used for the GPU tests. In Figure 14 we compare performance running miniFE with and without GPU-Direct. Without GPU-Direct the CG-solve algorithm must deep copy vectors from GPU to CPU memory, communicate vectors in CPU memory via MPI, and then deep copy vectors from CPU back to GPU memory. The addition of these deep copy operations causes more than a 2x slowdown in the CG-solve. This performance loss could be mitigated, but would not be eliminated, through a more complex operation that deep copies only the portions of the vectors that are communicated.

6.2. MiniMD

MiniMD serves as a proxy for classical molecular dynamics (MD) codes. It closely resembles some of the core functionalities of the MD code LAMMPS [44], but is much more limited. In particular miniMD implemented only two types of models: (1) a

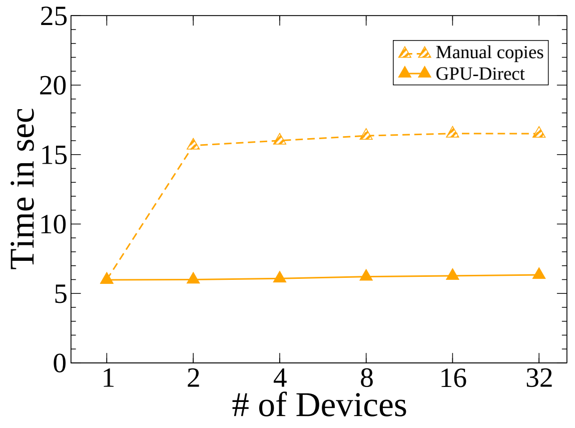

Figure 14: Time for 200 iterations of miniFE-Kokkos CG-solve on the Shannon GPU testbed The problem size is weak scaled, with 8M elements per device. The solid line represent runs using the GPU-Direct capabilities of MVAPICH2-1.9, while the dashed line shows results with manual deep copies during the communication phase. For each data point the best time out of 12 runs was used.

simple Lennard Jones system using the microcanonical ensemble and (2) a EAM simulation using the microcanonical ensemble. The miniMD code has four main components: (a) the force class calculates atoms' interactions, (b) the neighbor class creates the list of neighbors j for each atom i, (c) the comm class handles communication, and (d) the integrate class performs the time integration.

We compare miniMD-Kokkos, with OpenMP back-end, to the miniMD-OpenMP variant. Since there is no pure CUDA version of miniMD, no comparison is done on GPUs. For all tests the respective version 1.2 of miniMD has been used. The miniMD performance test is for strong scaling with 2,048,000 atoms, in contrast to the miniFE weak scaling test. This problem size falls into the range of typical MD simulations between 10^5 and 10^7 atoms. Details of the test problem configuration are given in Table 3.

The code was run with a single MPI rank per device, with 32 and 224 threads on Xeon and Xeon Phi respectively. MiniMD-OpenMP was compiled with a chunk size of 64 for the static OpenMP scheduling. Each test is run twelve times and the best eight times are included in the results.

Total Time Consumed performance metric. Our performance metric for strong scaling tests is the total time consumed (or total time), which is the wall-clock time of the test *times* the number of devices used. This metric is similar to the commonly used billing metric of CPU hours. The traditional parallel efficiency measure is the inverse of this total time metric normalized to some reference time (*e.g.*, time to run on a single device). We present results in the total time consumed metric to allow a direct comparison of performance across devices, and a break down of performance among components of miniMD.

The total time consumed for 1,000 simulation steps on 1 to 32 devices is given in Figure 15. The first observation from these results is that strong scaling is much worse on Xeon Phi and Kepler GPUs than on the Xeon CPUs. The Xeon Phi result can be explained by the poor MPI performance previously noted in Section 6.1. A comparison between miniMD-Kokkos and miniMD-OpenMP shows that Kokkos introduces minimal runtime overhead versus using OpenMP directly. While miniMD-Kokkos is about 10% slower than miniMD-OpenMP on the Xeon Phi system; however, the reverse is true for the CPU runs.

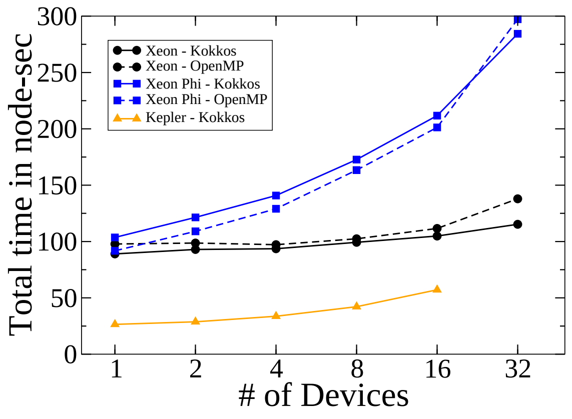

Figure 15: Total time consumed for running 1,000 simulation steps of a 2,048,000 atom Lennard-Jones simulation with miniMD variants on different test-beds. Note that a horizontal line indicates perfect strong scaling and an upward trend indicates a loss in parallel efficiency. The solid lines represent runs using miniMD-Kokkos, while the dashed lines show results with peer variants. For each data point the average of the best 8 runs out of 12 was used.

Timings from the four miniMD computations (force calculation, neighborlist construction, communication, and time integration), shown in Figure 16, are obtained to gain additional insight into miniMD performance. First, the increase in total time correlated with increasing MPI ranks is almost entirely caused by the communication rou-

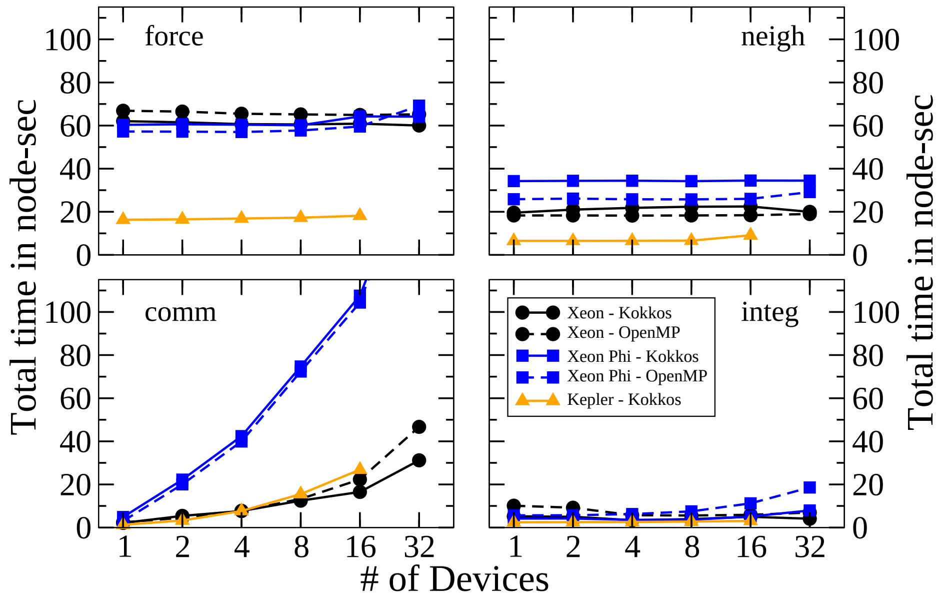

Figure 16: Breakdown of total time consumed for running 1,000 simulation steps of a 2,048,000 atoms Lennard-Jones simulations with miniMD variants on different test-beds. Note that a horizontal line would indicate perfect strong scaling and an upward trend indicates a loss in parallel efficiency. The solid lines represent runs using miniMD-Kokkos, while the dashed lines show results with peer variants. For each data point the average of the best 8 runs out of 12 used. The subfigures show the time for the force calculation (upper left), neighborlist construction (upper right), interprocess communication (lower left), and integration (lower right).

tines. Second, on the Xeon Phi miniMD-Kokkos is slower than miniMD-OpenMP due to the neighborlist computation; even though this computation is virtually identical in both variants. Third, on the Xeon Phi miniMD-Kokkos is slightly faster than miniMD-OpenMP in the integration computation; even though this computation is virtually identical in both variants. Analysis of these differences will require comparison via performance analysis tools or inspection of generated assembly code.

7. Legacy Code Migration Strategy

The legacy code migration strategy presented here was developed based upon our experience implementing Kokkos variants of miniMD and miniFE. This strategy has five steps: (1) change data structures, (2) develop functors, (3) enable dispatch (offload model) for GPU execution, (4) optimize algorithms for threading, and (5) specialize kernels for specific architectures. These steps can be carried out either for the whole legacy, or incrementally within components of the legacy code. We described these steps using MiniMD and MiniFE examples.

7.1. Data Structures

The original MiniFE and miniMD data structures represent two typical situations found in a wide range of legacy codes. MiniMD uses "raw" allocated memory accessed via pointers for its arrays. MiniFE uses storage containers (in particular `std`::vector) for its arrays.

We created the `Kokkos::vector` as a thin wrapper of a one dimensional `Kokkos::View` to allow simple replacement of `std::vector` objects with `Kokkos::vector`. This class does not have the full functionality of `std::vector`. Code executing in device space has access to `operator[]`, `b`egin(), and `e`nd() functions. Code execution in CPU space has access to `b`egin() and `e`nd() functions, and also has access to common modifying functions such as resize() and push_back(). `Kokkos::vector` also functions to manage deep copy operations when compiling for a GPU device.

MiniMD uses one and two dimensional "raw" arrays. The most significant miniMD arrays are the positions, velocities and forces of particles (`double **x, **v, **f;`), the number of neighbors for each particle (`int* numneighs;`), and the neighborlist (`int** neighbors;`). We introduce appropriate `typedef` declarations as shorthands for the various `View` types, as shown in Figure 17. We declare separate types position, velocity and force arrays to allow mixed precision computations. For example, force calculations can be performed in single precision and precision critical time integration can still be performed in double precision. In addition, we declare `View` types with traits such as `const` and random access.

```
typedef Kokkos::Host DefaultDevice;

// Precision for position, velocity, and force
typedef double X_Float;
typedef double V_Float;
typedef double F_Float;

// Particle positions always use right layout
// to improve cache line usage with random access
typedef View< X_Float*[3],
              LayoutRight,
              DefaultDevice>  t_x_array ;
typedef View< const X_Float*[3],
              LayoutRight,
              DefaultDevice,
              ReadRandom > t_x_array_rnd;
typedef t_x_array::HostMirror t_x_array_host ;

// Particle velocities use default layout for
// appropriate contiguous access pattern
typedef View<V_Float*[3],DefaultDevice> t_v_array ;
typedfe t_v_array::HostMirror t_v_array_host ;

// Neighborlist uses default layout for
// device-appropriate access pattern
typedef View<int** ,DefaultDevice> t_neighs;
typedef View<const int** ,
             DefaultDevice> t_neighs_const;
```

Figure 17: C++ type definitions for some MiniMD-Kokkos array data structures.

After all required array type declarations are introduced, the miniMD code is incrementally changed. First, every array allocation statement is changed to a `View` allocation as shown in Figure 18. Incremental migration to the new data structures is achieved by temporarily wrapping MiniMD's old array-of-pointers data structures around `View` allocated data, as showin in Figure 18. This temporary wrapping strategy introduces two restrictions: (1) the wrapped array-of-pointers data structure can only be accessed on the CPU and (2) the `View` layout is forced to match the original data structure's layout. This phase is complete when miniMD passes its test suite.

```
// Original array variables
double **x, **v, **f;

// Kokkos array variables
t_x_array d_x; t_x_array_host h_x;
t_v_array d_v; t_v_array_host h_v;
t_f_array d_f; t_f_array_host h_f;

// Allocate on the device
t_x_array d_x = t_x_array("X",natoms);

// View or allocate a CPU copy
t_x_array_host h_x = create_mirror_view(d_x);

// Temporarily wrap old data structure:
double **x = new double*[natoms];
for(int i = 0; i<natoms; i++)
  x[i] = & h_x(i,0);
```

Figure 18: Migration of miniMD "raw" arrays to corresponding `View` types requires replacing array declarations and allocations. Incremental migration can be accomplished by temporarily wrapping the original array data structures around `View` allocated data.

"Raw" array-of-pointer variables are now incrementally replaced with `View` variables. This requires replacing the array-of-pointer access syntax, `x[i][j]`, to `View` syntax, `x(i,j)`. After these replacements are completed the temporary wrappers can be removed.

Some parts of the original MiniMD use flat array access to enable vectorizing. This is possible since the "raw" two-dimensional arrays are allocated as two arrays: (1) a one dimensional array of data and (2) an array-of-pointers into the array of data. With `View` variables this manual optimization is no longer necessary.

```
// Original access syntax
double** x = atom->x;
const X_Float ytmp = x[i][1];

// Optimized original access syntax
double* x = &atom->x[0][0];
const X_Float ytmp = x[i * 3 + 1];

// Kokkos access syntax
t_x_array x = atom->x;
const X_Float ytmp = x(i,1);
```

7.2. *Functors*

Threaded execution with `parallel_for` or `parallel_reduce` requires encapsulation of computations in functors, until C++11 lambda capability is available. In miniMD, nearly all computations are performed by class member functions that loop over arrays. We developed the following porting strategy in order to minimize changes to the original miniMD code.

Given a class member function containing a loop, we modify the class and create a corresponding functor, as shown in Figure 19. This modification can be introduced in the following steps.

1. Modify the original function to copy all incoming arguments into class member data.
2. Create a new class member function (`C::f_item(int i)` in Figure 19) that contains the original function's loop body and a loop index as its only argument.
3. Replace the loop body within the function with a call to the new loop body function and test the modified code.
4. Create a wrapper functor (f_functor in Figure 19) that has an instance of the original class as a member and a parentheses operator that calls the new loop body function.
5. Change the original class member function to create and dispatch the wrapper functor via `parallel_for` or `parallel_reduce`.

Note that when C++11 lambda functionality is sufficiently supported this strategy can be superseded by a syntactically simpler lambda strategy.

In miniFE functions are not members of a class. The functor-wrapper migration strategy is similar, as shown in Figure 20. The difference is that incoming arguments of the original function become members of the functor and the original loop-body is directly copied into the functor's operator.

These porting steps are typically sufficient to use Kokkos with a single thread. However, this strategy does not guarantee that the loop body functions are thread-safe – that thread parallel execution of the loop bodies does not have write conflicts or other race conditions. A developer must still identify all write conflicts or other race conditions in the loop body function and appropriately mitigate those conditions. In many cases the simplest way to mitigate write conflicts is to use atomic update functions. Kokkos wraps a collection of commonly available and compiler dependent atomic update functions under a portable interface.

An example where atomic updates could be used is porting the original loop body of the Lennard Jones kernel given in Figure 21. In its original non-thread parallel form, the LJ-kernel takes advantage of Newton's third law: it computes the force between a pair of atoms once and then adds the opposite force to both atoms (compute f_{ij} on atom i due to atom j and then $f_{ji} = -f_{ij}$). In this algorithm, the neighborlist of a particle only needs to contain half of its actual neighbors, thus we refer to it as the *half-neighbor* algorithm. Accumulation of forces in the half-neighbor algorithm has several write conflicts, as identified in Figure 21.

```
// Original class member function
class C {
public:
  void f(double k, int N) {
     for(int i = 0; i<N; i++) {
        // loop body
     }
  }
};

// Modified class with a functor-wrapper
class C {
public:
  double k ;
  void f(double ktmp, int N) {
    k = ktmp;
    f_functor<DefaultDevice> func(*this);
    parallel_for(N,func);
  }

  void f_item(int i) const {
    // loop body
  }
};

template<class Device>
struct f_functor {
  typedef Device device_type
  C c;
  f_functor(C &c_in): c(c_in) {};
  operator()(int i) const {
    c.f_item(i);
  }
};
```

Figure 19: Illustration of our strategy to port miniMD to Kokkos and minimize changes to miniMD class' source code by wrapping loop bodies within functors. Given C++11 lambda capability a syntactically cleaner strategy could be used.

The original kernel can be made thread-safe by replacing the force updates with calls to atomic_fetch_add().

```
// old force update
f(j,0:2) -= d_ij[0:2] * force;

// new code
atomic_fetch_add(&f(j,0:2) , -d_ij[0:2] * force);
```

Atomic functions can be a straight forward and expedient approach to make serial code thread-safe. However, this approach may not result in the most performant implementation. Atomic update functions introduce a noticeable performance overhead, even when executing on a single thread. Modifying kernels to be thread-safe and performant often

```
// old code
void f(double k, int N) {
  for(int i = 0; i<N; i++) {
    // loop body
  }
}

// new code
template<class Device>
struct f_functor {
  typedef Device device_type;
  double k;
  f_functor(double ktmp): k(ktmp) {};
  operator()(int i) const {
    // loop body
  }
};

void f(double k, int N) {
  f_functor<DefaultDevice> func(k);
  parallel_for(N,func);
}
```

Figure 20: Our strategy for porting miniFE to Kokkos minimized changes to miniFE functions' source code by wrapping loop bodies within functors.

involves redesigning their algorithms with strategies such as coloring, redundant computation, or redundant storage with subsequent reductions. Such algorithmic redesign should be delayed until it is known that a kernel's performance has a significant negative impact on the overall performance of the application.

7.3. Enable GPU execution

The migration strategy presented so far is typically sufficient to enable threaded execution on CPUs or Xeon Phi. For execution on GPUs, it is often necessary to add explicit management of the different memory spaces, especially when the code is not fully migrated to Kokkos. For example, the current setup phase of miniFE is not GPU ready due to the use of `std`:map. Thus, all arrays are currently maintained on the CPU in `View::HostMirror` variables. For the miniFE CG-Solve phase, all arrays associated with the linear system are deep copied to the GPU and CG-Solve computations are performed on the GPU.

7.4. Optimizing algorithms for thread scalability

Once a code is running thread parallel on all devices, it can be necessary to redesign some of the original serial algorithms for thread scalability. Kernels which show poor strong scaling performance, with respect to the number of threads, are candidates for redesign. In miniMD, the original LJ force kernel is one such candidate since using

```
  int numneighs = numneigh[i];
  double x_i[0:2] = x(i, 0:2);
  double f_i[0:2] = 0;

  for(int k = 0; k < numneighs; k++) {
    int j = neighbors(i, k);
    double d_ij[0:2] = x_i[0:2] - t_x(j, 0:2);
    double rsq = d_ij[0]*d_ij[0] + ... ;

    if(rsq < cutforcesq) {
      double sr2 = 1.0 / rsq;
      double sr6 = sr2 * sr2 * sr2;
      double force = 48.0 * sr6 * (sr6 - 0.5) * sr2;

      // Write conflict is not thread safe !!!
      f_i[0:2] += d_ij[0:2] * force;

      if(j < nlocal) {
        // Write conflict is not thread safe !!!
        f(j,0:2) -= d_ij[0:2] * force;
      }
    }
  }

  // Write conflict is not thread safe !!!
  f(i,0:2) += f_i[0:2];
```

Figure 21: The original Lennard Jones kernel in miniMD is not thread-safe due to multiple statements with potential write conflicts.

atomic updates in the inner-most loop is very expensive.

When running the standard miniMD problem with 864k atoms, this kernel requires about 22.4 seconds using 32 MPI ranks (single thread) on our Compton test-bed's Xeon CPUs. Running the same problem with 1 MPI rank, 32 threads, and using atomic updates results in the kernel's time increasing to 71.4 seconds. After modifying the kernel to eliminate all possible write conflicts, at the cost of doing twice as much calculations (using full neighbor lists), the same computation takes 27.7 seconds. For a 32 MPI ranks versus 32 threads test case on a single compute node the threaded version is still slower than the original MPI-only kernel. However, it is not always possible or performant to execute an MPI process on each core. For example, running MPI-only on a GPU or on every core in a cluster of Xeon Phi is not feasible due to memory and network interface limitations.

Results from this miniMD migration process is given in Figure 22. The total time for the miniMD 2,048,000 atom test problem is plotted for one and 16 compute nodes using (a) MPI-only with half neighborlists, (b) a single MPI rank per node with threads and half neighborlist using atomic operations, and (c) a single MPI rank with threads and full neighborlists. MPI-only is the most efficient way to run the problem when using only CPUs on a single node. In every other configuration, the write conflict free full neighborlist approach has better performance.

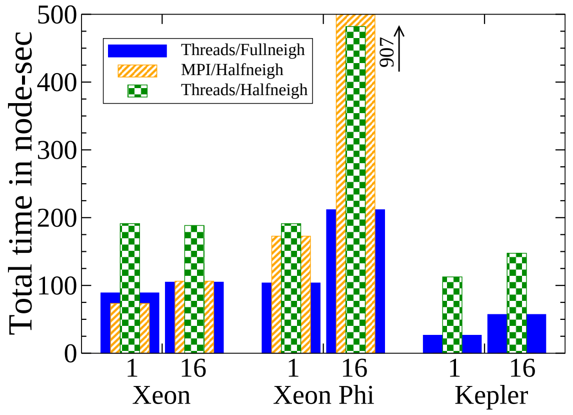

Figure 22: Total time consumed for running 1,000 simulation steps of a 2,048,000 atom Lennard-Jones simulation with miniMD variants on different test-beds. A comparison is shown of running miniMD on 1 and 16 nodes or devices with MPI-only using half neighborlists, threads with half neighborlists, and threads with full neighborlists. On Xeon and Xeon Phi testbeds 32 and 224 MPI ranks or threads have been used per node or device. An exception is the Xeon Phi run with 16 devices, where 224 MPI ranks per device did not run and 56 (one per core) had to be used instead. On the Kepler GPU testbed running in MPI only mode is not possible.

Performance limitations of MPI-only on a Xeon Phi are apparent using a single device. Threaded execution of the half neighborlist and atomic update kernel is competitive with the MPI-only approach, and the full neighborlist kernel is 40% faster. When trying to run more than one Xeon Phi in MPI-only mode, limitations of the current software stack prevented us from using more than 56 ranks per Xeon Phi (1 rank per core). Consequently, MPI-only performance becomes almost 80% worse than the full neighborlist kernel.

Three common strategies for redesigning algorithms for improved thread scalability are: redundant computations, redundant storage followed by thread-safe reductions, and coloring. Redundant computations were used for the LJ-Kernel because it has only a 2x redundancy, is more scalable and memory efficient than redundant storage, and is much simpler to implement than strategy coloring. Redundant storage can work well on CPUs with ample main memory and cache; however, it can be inherently unscalable on devices with a small amount of memory per thread.

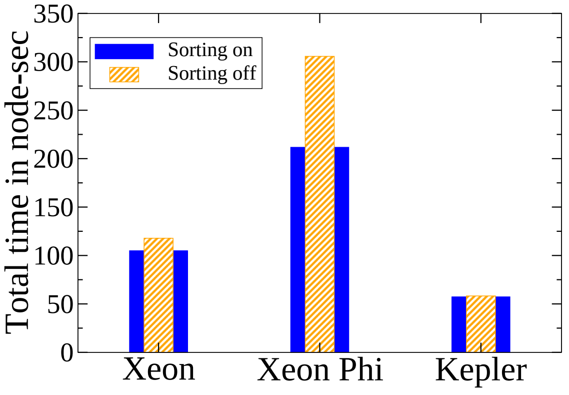

Figure 23: MiniMD performance is significantly improved on Xeon Phi by sorting atom arrays to improve locality of neighbor atoms.

As mentioned in Section 4.4, threaded scalability may not be sufficient for an algorithm to obtain the best performance on a given architecture. This was the situation with miniMD where the array of atoms is sorted to improve data locality for neighbor atoms. Figure 23 shows the total time with and without sorting activated for the previously described miniMD tests using 16 nodes or devices. For the CPU and GPU the performance difference is less than 10%; however, without sorting performance on the Xeon Phi decreases by more than 30%. In this situation, an algorithmic modification was critical for Xeon Phi performance (30%), and was also slightly beneficial for CPU and GPU performance.

7.5. Specialize kernels for specific architectures

In some cases, it is not possible to design a single algorithm which is nearly optimal on all architectures. For example, a device specific feature can be leveraged for a more performant implementation of an algorithm on that device. In other situations, optimal algorithms might be different due to the large difference in the number of concurrent threads (*e.g.*, $O(10^5)$ for GPU, $O(10^2)$ for Xeon Phi, and $O(10^1)$ for CPU). In these situations, it can be necessary to introduce a device-specialized version of a computation.

This situation occurred in miniFE for the sparse matrix-vector multiplication shown in Figure 24. On a CPU or Xeon Phi each row of the matrix is handled by a single thread; however, on a GPU it is beneficial to use multiple threads per row [45]. Furthermore, on NVIDIA's Kepler generation of GPUs the shuffle operations can be used to optimize intra-row reductions. By using the number of threads per row as a template argument to this function we are able to differentiate the code for GPUs without having a negative impact on CPU execution.

```
template<int TPRow> // threads per row
void operator()(const int threadId) const {
  //For OpenMP or pthreads back-end iRow=threadId;
  //and lane=0;
  const int iRow = threadId/TPRow;
  const int lane = threadId%TPRow;
  scalar_type sum = 0;

  if(doalpha != -1) {
    const SparseRowView<CrsMatrix> row=m_A.row(iRow);

    #pragma loop count (15)
    #pragma unroll
    for(int i=lane; i<row.length; i+=TPRow) {
      sum +=  row.value(i) * m_x(row.colidx(i));
  } else {
    const SparseRowView<CrsMatrix> row=m_A.row(iRow);

    #pragma loop count (15)
    #pragma unroll
    for(int i=lane; i<row.length; i+=TPRow) {
      sum -=  row.value(i) * m_x(row.colidx(i));
  }

  //Perform reduction on GPUs
  //For OpenMP or pthreads back-end, compiler will
  //optimize reduction away since TPRow=1
  if(TPRow > 1) sum += shfl_down(sum, 1,TPRow);
  if(TPRow > 2) sum += shfl_down(sum, 2,TPRow);
  if(TPRow > 4) sum += shfl_down(sum, 4,TPRow);
  if(TPRow > 8) sum += shfl_down(sum, 8,TPRow);
  if(TPRow > 16)sum += shfl_down(sum,16,TPRow);

  //On GPUs only one thread writes result back
  if(lane==0) {
    if(doalpha * doalpha != 1) sum *= alpha(0);

    if( dobeta == 0)     m_y(iRow) = sum ;
    else if(dobeta == 1) m_y(iRow) += sum ;
    else if(dobeta == -1)m_y(iRow) = -m_y(iRow)+sum;
    else m_y(iRow) = beta(0) * m_y(iRow) + sum;
  }
}
```

Figure 24: Sparse matrix-vector multiplication specialized according to the number of threads used per row. This number always equal to one on CPU and Xeon Phi, and is greater than one on a GPU. The GPU implementation utilizes the inter-thread shuffle operation to optimize inter-row reductions.

A similar situation occurs in miniMD's neighborlist construction. The first part of this construction algorithm assigns atoms to bins in space and is the same across all devices. The second part of the algorithm builds the neighborlist from these bins and must have a specialized version for a GPU. On CPUs or Xeon Phi, each thread works independently on a single atom at a time. It determines the bin of its atom, and then loops over all bins within the neighbor cutoff to find all neighbor atoms. On a GPU shared memory can be used to make this process more efficient. Instead of assigning single atoms to threads, bins are assigned to team of threads with access to GPU shared memory (*i.e.*, a CUDA thread block is assigned to each bin). Threads in a team load the coordinates of atoms in neighboring bins into shared memory and then work from this shared memory. In that way the coordinates are used multiple times for each original load from the slower main memory.

8. Conclusion

The Kokkos C++ library implements our strategy for manycore performance portable HPC applications and libraries. Two foundational abstractions are implemented: (1) dispatching parallel functors to a manycore device and (2) managing the layout of multidimensional arrays so that those functors have device-appropriate memory access patterns. We defined Kokkos' manycore parallel abstractions and summarized the C++ API.

We demonstrated performance portability unit test kernels and mini-applications that achieve at least 90% of the performance of architecture specific, optimized variants of those test cases. Finally we described a strategy by which legacy C++ applications and libraries can use Kokkos to migrate to manycore parallelism. Migration of our mini-applications to Kokkos demonstrated that sometimes legacy computational kernels are inherently not thread scalable and must be redesigned.

Kokkos will update existing, or adopt new, back-end implementations as manycore architectures and their programming models evolve. In this way, HPC applications and libraries using Kokkos can immediately benefit from new manycore capabilities. Furthermore, our ongoing analysis of manycore architectures' performance drives continued optimization of back-end implementations.

Kokkos is under active research and development focusing on improving performance, supporting new layouts and aggregate "scalar" value types (Section 3.3), and enabling hierarchical task-data parallel dispatch. Development has begun on higher level libraries such as sparse linear algebra and array-based containers (*e.g.*, hash-maps).

Kokkos is publicly available through the Trilinos repository at **www.trilinos.org**, and is being used to migrate the Trilinos suite of libraries to manycore architectures. An effort has begun to refactor

LAMMPS [44] to use Kokkos for thread level parallelism. MiniMD and miniFE are available through the Mantevo repository at **www.mantevo.org**.

Acknowledgment

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This paper is cross-referenced at Sandia as SAND2013-5603J.

References

[1] C. G. Baker, M. A. Heroux, H. C. Edwards, A. B. Williams, A Light-weight API for Portable Multicore Programming, in: Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, IEEE, 2010, pp. 601–606. doi:10.1109/PDP.2010.49.

[2] H. C. Edwards, D. Sunderland, C. Amsler, S. Mish, Multicore/gpgpu portable computational kernels via multidimensional arrays, in: Cluster Computing (CLUSTER), 2011 IEEE International Conference on, IEEE, 2011, pp. 363–370. doi:10.1109/CLUSTER.2011.47.

[3] H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, S. Mish, Manycore performance-portability: Kokkos multidimensional array library, Scientific Computing (2012) 89–114doi:10.3233/SPR-2012-0343.

[4] H. C. Edwards, D. Sunderland, Kokkos array performance-portable manycore programming model, in: PMAM, 2012, pp. 1–10.

[5] R. Garcia, J. Siek, A. Lumsdaine, Boost.MultiArray, www.boost.org/libs/multi_array (Jun. 2013).

[6] Trilinos website, trilinos.sandia.gov/ (Aug. 2011).

[7] K. Gregory, A. Miller, C++ Amp, Accelerated Massive Parallelism with Microsoft Visual C++, Microsoft Press, 2012.

[8] C++ Amp home page, http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx (Jul. 2013).

[9] Cuda Toolkit Thrust documentation, docs.nvidia.com/cuda/thrust/ (Jun. 2013).

[10] M. Ospici, D. Komatitsch, J.-F. Mehaut, T. Deutsch, Sgpu 2: a runtime system for using of large applications on clusters of hybrid nodes, in: Second Workshop on Hybrid Multi-core Computing, held in conjunction with HiPC, 2011, pp. 1–8.

[11] T. Gautier, J. V. F. Lima, N. Maillard, B. Raffin, et al., Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures, in: 27th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2013.

[12] XKAAPI home page, http://kaapi.gforge.inria.fr (Jul. 2013).

[13] OpenACC home page, http://openacc.org/ (Jul. 2013).

[14] OpenHMPP home page, http://www.caps-entreprise.com/openhmpp-directives/ (Jul. 2013).

[15] J. Planas, R. M. Badia, E. Ayguadé, J. Labarta, Hierarchical task-based programming with starss, International Journal of High Performance Computing Applications 23 (3) (2009) 284–299.

[16] A. Duran, E. Ayguad, R. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, Ompss: A proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (2) (2011) 173–193, cited By (since 1996)18.

[17] OmpSs home page, http://pm.bsc.es/ompss (Jul. 2013).

[18] V. V. Dimakopoulos, P. E. Hadjidoukas, Hompi: a hybrid programming framework for expressing and deploying task-based parallelism, in: Euro-Par 2011 Parallel Processing, Springer, 2011, pp. 14–26.

[19] PEPPHER home page, http://www.peppher.eu/ (Jul. 2013).

[20] OpenCL home page, http://www.khronos.org/opencl/ (Jul. 2013).

[21] C. Augonnet, S. Thibault, R. Namyst, P. Wacrenier, Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. concurrency and computation: Practice and experience, Special Issue: Euro-Par 2009 23 (2011) 187–198.
URL `http://runtime.bordeaux.inria.fr/StarPU/`

[22] StarPU home page, http://runtime.bordeaux.inria.fr/StarPU/ (Jul. 2013).

[23] E. A. Luke, T. George, Loci: a rule-based framework for parallel multi-disciplinary simulation synthesis, Journal of Functional Programming 15 (2005) 477–502. doi:10.1017/S0956796805005514.

[24] Loci home page, http://www.cse.msstate.edu/ luke/loci/ (Jul. 2013).

[25] Cilk Plus home page, http://cilkplus.org/ (Jul. 2013).

[26] J. Reinders, Intel Threading Building Blocks, O'Reilly, 2007.

[27] Intel threading building blocks home page, (Jul. 2013).

[28] L. Kalé, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, in: A. Paepcke (Ed.), Proceedings of OOPSLA'93, ACM Press, 1993, pp. 91–108.

[29] Charm++ home page, http://charm.cs.illinois.edu/ (Jul. 2013).

[30] The OpenMP API Specification for Parallel Programming, openmp.org/ (Jun. 2013).

[31] CUDA home page, www.nvidia.com/object/ cuda_home_new.html (Jun. 2013).

[32] IEEE Std 1003.1, 2004 Edition, <pthread.h> (2004).

[33] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications, in: IEEE (Ed.), PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, Pisa, Italie, 2010. doi:10.1109/PDP.2010.67.

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

[35] A. Chtchelkanova, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. van de Geijn, Towards usable and lean parallel linear algebra libraries, Technical report TR-96-09, Department of Computer Sciences, University of Texas at Austin (May 1996).

[36] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.

[37] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, A class of parallel tiled linear algebra algorithms for multicore architectures, Parallel Computing 35 (2009) 38–53.

[38] E. T. Phipps, R. A. Bartlett, D. M. Gay, R. J. Hoekstra, Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation, in: C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 351–362.

[39] R. P. Pawlowski, E. Phipps, A. G. Salinger, Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, Part I: Template-based generic programming, Scientific Programming 20 (2012) 197–219.

[40] R. P. Pawlowski, E. T. Phipps, A. G. Salinger, S. J. Owen, C. M. Siefert, M. L. Staten, Automating embedded analysis capabilities and managing software complexity in multiphysics simulation part II: application to partial differential equations, Scientific Programming 20 (2012) 327–345.

[41] T. Mattson, B. Sanders, B. Massingill, Patterns for parallel programming, 1st Edition, Addison-Wesley Professional, 2004.

[42] mvapich home page, mvapich.cse.ohio-state.edu (Jun. 2013).

[43] J. Liu, J. Wu, D. K. Panda, High performance rdma-based mpi implementation over infiniband, Int. J. Parallel Program. 32 (3) (2004) 167–198.

[44] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, Journal of Computational Physics 117 (1995) 1–19.

[45] N. Bell, M. Garland, Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (Dec. 2008).