Kokkos: Enabling manycore performance portability through polymorphic memory access patterns

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

HIGHLIGHTS

• We developed a performance portable programming model (PM) for manycore devices.
• Unifying parallel dispatch and data layout is mandatory for performance portability.
• The Kokkos C++ library implements this PM with pthreads, OpenMP, and CUDA back-ends.
• Demonstrate Xeon Phi and NVIDIA GPU performance portability with mini-applications.
• Recommend a strategy for legacy application codes to migrate to manycore.

ARTICLE INFO

Article history:
Received 18 July 2013
Received in revised form 7 July 2014
Accepted 10 July 2014
Available online 22 July 2014

Keywords:
Parallel computing
Thread parallelism
Manycore
GPU
Performance portability
Multidimensional array
Mini-application

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 a growing list of applications and domain libraries.

© 2014 Elsevier Inc. All rights reserved.

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 [2] to a hierarchy of broadly usable libraries. Our earlier implementation of Kokkos’ fundamental abstractions was referred to as KokkosArray [16,17,15]. 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 [18] flexible storage ordering.

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 [44] for NVIDIA GPUs, and pthreads [21] or OpenMP [45] for CPUs and Intel Xeon Phi. Pthreads and OpenMP back-ends optionally use the portable hardware locality (hwloc) library [5] 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.

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 [22], 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 in 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.

### Table 1

<table>
<thead>
<tr>
<th>Programming model</th>
<th>Portable cpu/gpu</th>
<th>Data layout</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kokkos [44]</td>
<td>Yes</td>
<td>Yes</td>
<td>Library</td>
</tr>
<tr>
<td>C++ AMP [20,7]</td>
<td>Yes</td>
<td>Yes</td>
<td>Language</td>
</tr>
<tr>
<td>Thrust [12]</td>
<td>Yes</td>
<td>No</td>
<td>Library</td>
</tr>
<tr>
<td>SGPUZ [34]</td>
<td>Yes</td>
<td>No</td>
<td>Library</td>
</tr>
<tr>
<td>XEAP [19,45]</td>
<td>Yes</td>
<td>No</td>
<td>Library</td>
</tr>
<tr>
<td>OpenACC [31]</td>
<td>Yes</td>
<td>No</td>
<td>Directives</td>
</tr>
<tr>
<td>OpenMP [33]</td>
<td>Yes</td>
<td>No</td>
<td>Directives</td>
</tr>
<tr>
<td>StarSs [39]</td>
<td>Yes</td>
<td>No</td>
<td>Directives</td>
</tr>
<tr>
<td>OpenP8 [14,30]</td>
<td>Yes</td>
<td>No</td>
<td>Directives</td>
</tr>
<tr>
<td>HOMP [13]</td>
<td>Yes</td>
<td>No</td>
<td>Translator</td>
</tr>
<tr>
<td>PEPPHER [37]</td>
<td>Yes</td>
<td>No</td>
<td>Translator</td>
</tr>
<tr>
<td>OpenCL [32]</td>
<td>Yes</td>
<td>No</td>
<td>Language</td>
</tr>
<tr>
<td>StarPU [1,42]</td>
<td>Yes</td>
<td>No</td>
<td>Language</td>
</tr>
<tr>
<td>Loci [27,26]</td>
<td>No</td>
<td>Yes</td>
<td>Library</td>
</tr>
<tr>
<td>Cilk Plus [10]</td>
<td>No</td>
<td>Yes</td>
<td>Language</td>
</tr>
<tr>
<td>TBB [41,23]</td>
<td>No</td>
<td>No</td>
<td>Library</td>
</tr>
<tr>
<td>Charm++ [24,8]</td>
<td>No</td>
<td>No</td>
<td>Library</td>
</tr>
<tr>
<td>OpenMP [43]</td>
<td>No</td>
<td>No</td>
<td>Directives</td>
</tr>
<tr>
<td>CUDA [11]</td>
<td>No</td>
<td>No</td>
<td>Language</td>
</tr>
</tbody>
</table>

- a OpenMP 4.0 addresses GPU; however, compilers are not yet available.
- b PGI compiler can generate x86 code from CUDA.
3. Multidimensional array

A Kokkos multidimensional array consists of: (1) a set of datum \( x_i \) of the same value type and residing in the same memory space, (2) an index space \( X_i \) defined by the Cartesian product of integer ranges, and (3) a layout \( X_i \) a bijective map between the index space and the set of datum. (Note that equality of datums’ values does not imply the same datum: \( x_i = x_k \) \( i \neq k \).)

\[
X = (x_1, x_2, x_3) \rightarrow (x_1, x_2, x_3)
\]

\[
X_3 = \{0.0, N_0\} \times \{0.0, N_1\} \times \cdots
\]

\[
X_1 = x_1 \rightarrow \{x_1\}
\]

A function typically contains a sequence of nested loops over dimensions of an array \( X_3 \) and accesses array datum via the layout \( X_1 \). Thus, the composition of a function’s loop-and-indexing pattern and the array’s layout yields a memory access pattern.

\[
F[X_3] \text{ function implemented for } X_3
\]

\[
F[X_3] \circ X_3^{-1} \text{ resulting memory access pattern.}
\]

To modify the function’s memory access pattern one must either (1) change the dimension ordering \( X_3 \) and the loop-and-indexing pattern of the function \( F[X_3] \) or (2) change the layout of the array \( X_1 \).

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 (\( X_i \)) are chosen at compile time. Thus, a function’s memory access pattern \( (F[X_3] \circ X_3^{-1}) \) 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 [9,3]. 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 Fig. 1.

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 Views and pointers are analogous. This is illustrated by similar declarations for pointer and View types and “const-ness” violations.

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

Fig. 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.

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 Views and pointers are analogous to the “const-ness” semantics for C++ pointers, as illustrated in Fig. 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.

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 Fig. 3. In contrast to C++ standard container semantics, multiple View objects can reference the same allocated array. View


Fig. 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.

```cpp
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.
```

Fig. 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.

```cpp
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
```

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 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.

**Behavioral traits.** The `View` class has an optional behavioral trait template parameter as a second extension point. A user can use behavioral 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 Fig. 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 point is expected to gain importance as portable 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

```cpp`
```

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

```cpp`
```

Fig. 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.

```cpp`
```
into computational kernels by replacing intrinsic scalar data types (e.g., float or double) within the kernel with aggregate data types [38,35,36]. 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 to performance.

The View class incorporates aggregate data types into the data layout. In the example shown in Fig. 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.

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 [28] 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 the 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 work 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

```cpp
// Replace the ‘scalar’ type with 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.

// 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.

auto x = AutoDeriv<double> x(N,M,N); // Parentheses operator returns a view that is compatible with the AutoDeriv<double> type.

auto value = x(i,j);
```

Fig. 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 View class incorporates aggregate data types into the data layout. In the example shown in Fig. 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.

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 [28] 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 the 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 work 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

```cpp
// Template on the Device for portability.
template< class Type , class Device >
class AXPY_Funct { public:
  // Requirement: Identify execution space.
  typedef Device device_type;
  // Requirement: Provide work callback as
  void operator() ( int N, int Y ) const
  {
    y = x + y;
  };
  // 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_Funct<device,Cuda>( a , x , Y ) );
```

Fig. 8. Interface requirements for parallel_for are illustrated with this example AXPY functor that performs “\( Y = a X + Y \)” on arrays. The trivial constructor for initializing \( a \), \( x \), and \( Y \) data members is omitted.
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 functions 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 $2^N$ 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 (codenamed 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 \( N \times M \) problem size the MGS net bandwidth, \( B_{\text{mgs}} \), 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 that \( B \) is constant.

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

\[
B_{\text{mgs}} \approx \left( \frac{1}{B(N, M)} + \frac{2S}{5Nb} \right)^{-1}, \quad 1 \ll M. \tag{1}
\]

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 precision vectors over a range of vector lengths \( N \), as presented in Fig. 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 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 Eq. (1) to the data in Fig. 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.

![Fig. 10](image_url) 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.

![Fig. 11](image_url) 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.

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 Fig. 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 Fig. 11, loops over atoms and calculates the forces between pairs of atoms with a distance \( r_{\text{cutoff}} \) smaller than a cutoff radius. 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 Fig. 11 this list is implemented by the two dimensional neighbors array.

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,
Table 3
LJ-kernel and MiniMD test problem configuration.

<table>
<thead>
<tr>
<th>Component</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atoms (LJ-kernel test)</td>
<td>864,000</td>
</tr>
<tr>
<td>Atoms (MiniMD test)</td>
<td>2,048,000</td>
</tr>
<tr>
<td>Units</td>
<td>Lennard-Jones</td>
</tr>
<tr>
<td>Density</td>
<td>0.8442 (\sigma_0^{-1})</td>
</tr>
<tr>
<td>Initial temperature</td>
<td>1.44</td>
</tr>
<tr>
<td>Initial config</td>
<td>FCC lattice</td>
</tr>
<tr>
<td>Force cutoff</td>
<td>2.5 (\sigma_0)</td>
</tr>
<tr>
<td>Neighbor cutoff</td>
<td>2.8 (\sigma_0)</td>
</tr>
<tr>
<td>Neighbor type</td>
<td>Full</td>
</tr>
<tr>
<td>Renegoring every</td>
<td>20</td>
</tr>
<tr>
<td>Sorting every</td>
<td>20</td>
</tr>
<tr>
<td>Thermo calculation every</td>
<td>100</td>
</tr>
<tr>
<td>Threads on Xeon</td>
<td>32</td>
</tr>
<tr>
<td>Threads on Xeon Phi</td>
<td>224</td>
</tr>
<tr>
<td>Density</td>
<td>0.8442 (\sigma_0^{-1})</td>
</tr>
<tr>
<td>Initial temperature</td>
<td>1.44</td>
</tr>
<tr>
<td>Initial config</td>
<td>FCC lattice</td>
</tr>
<tr>
<td>Force cutoff</td>
<td>2.5 (\sigma_0)</td>
</tr>
<tr>
<td>Neighbor cutoff</td>
<td>2.8 (\sigma_0)</td>
</tr>
<tr>
<td>Neighbor type</td>
<td>Full</td>
</tr>
<tr>
<td>Renegoring every</td>
<td>20</td>
</tr>
<tr>
<td>Sorting every</td>
<td>20</td>
</tr>
<tr>
<td>Thermo calculation every</td>
<td>100</td>
</tr>
<tr>
<td>Threads on Xeon</td>
<td>32</td>
</tr>
<tr>
<td>Threads on Xeon Phi</td>
<td>224</td>
</tr>
</tbody>
</table>

Fig. 12: LJ-kernel performance in MiniMD on Intel Xeon CPU, Intel Xeon Phi, and a 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. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The importance of polymorphic array layouts is highlighted in reading the neighbor index array, \(\text{neighbors}(i,jj)\) 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.

The 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 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 3.3 GB 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 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 [29,25] implementation of MPI so that MPI can directly access GPU memory via CUDA GPU-Direct. Thus, explicit data copies are not necessary between the device and the CPU during the CG-solve.

Weak scaling performance of the miniFE CG-solve phase is shown in Fig. 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

\(^1\) Recipient of a 2013 R&D 100 Award, www.rdmag.com/award-winners/2013/07/2013-r-d-100-award-winners.
particular miniMD implemented only two types of models: (1) a simple Lennard-Jones system using the microcanonical ensemble and (2) an 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 code 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 breakdown of performance among components of miniMD.

The total time consumed for 1000 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.

**Fig. 15.** Total time consumed for running 1000 simulation steps of a 2,048,000 atom Lennard-Jones simulation with miniMD variants on different test-beds.

Timings from the four miniMD computations (force calculation, neighborlist construction, communication, and time integration), shown in Fig. 16, are obtained to gain additional insight into the performance aspects of miniMD.
into miniMD performance. First, the increase in total time correlated with increasing MPI ranks is almost entirely caused by the communication routines. 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[], begin(), and end() functions. Code executing in CPU space has access to begin() and end() 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 Fig. 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.

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 Fig. 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 shown in Fig. 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.

“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.
Fig. 17. C++ type definitions for some MiniMD-Kokkos array data structures.

```cpp
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>
  t_x_array;
typedef View< const X_Float >=3 , LayoutRight>
  t_x_array const;
typedef t_x_array: HostMirror t_x_array host;
```

Fig. 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.

```cpp
// Original array variables
double ** x, **v, **f;
// Kokkos array variables
t_x_array dx, x array host h_x;
// Allocate on the device
t_x_array dx = t_x_array("X", natoms);
// View or allocate a CPU copy
t_x_array host h_x = create_mirror_view(dx);
// Temporarily wrap old data structure:
double ** x = new double*[natoms];
for (int i = 0; i<natoms; i++)
x[i][i] = h_x[i][i];
```

Fig. 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.

```cpp
// 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;
      _functor<DefaultDevice>() func(*this);
    }
    forward_for(N,funct);
  }
  void f_item(int i) const {
    // loop body
  }
};
```

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 Fig. 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 Fig. 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 (_functor in Fig. 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-wrapping migration strategy is similar, as shown in Fig. 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 Fig. 21. In its
even when executing on a single thread. Modifying kernels to be
update functions introduce a noticeable performance overhead,
may not result in the most performant implementation. Atomic
approach to make serial code thread-safe. However, this approach
force updates with calls to
write conflicts, as identified in
Accumulation of forces in the half-neighbor algorithm has several
the neighborlist of a particle only needs to contain half of its
on atom
once and then adds the opposite force to both atoms (compute
Newton’s third law: it computes the force between a pair of atoms
original non-thread parallel form, the LJ-kernel takes advantage of
Regularized Newton’s third law results in:

\[
\mathbf{f}_{ij} = \mathbf{r}_{ij} \mathbf{f}_{ij} = \mathbf{r}_{ij} \frac{\partial U}{\partial \mathbf{r}_{ij}}
\]

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

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 Fig. 21.

The original kernel can be made thread-safe by replacing the
force updates with calls to $\text{atomic\_fetch\_add()}$.

Atomic functions can be a straightforward and expedient ap-
proach 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 involves redesigning their algo-
rithms with strategies such as coloring, redundant computation,
or redundant storage with subsequent reductions. Such algorithm-
redesign should be delayed until it is known that a kernel’s
performance has a significant negative impact on the overall per-
formance 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 cur-
cently maintained on the CPU in View::HostMirror variables.
For the miniFE CG-Solve phase, all arrays associated with the lin-
ear 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 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 s 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 s. 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 s. 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 Fig. 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
both cases, the write conflict free full neighborlist approach has better
performance.

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 im-
proved thread scalability are: redundant computations, redundant
storage followed by thread-safe reductions, and coloring. Redund-
ant computations were used for the LJ-kernel because it has only
a 2× redundancy, is more scalable and memory efficient than re-
dundant storage, and is much simpler to implement than strategy

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

FIG. 21. The original Lennard-Jones kernel in miniMD is not thread-safe due to
multiple statements with potential write conflicts.
Fig. 22. Total time consumed for running 1000 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.

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

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.

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. Fig. 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 XeonPhi 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 Fig. 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 [4]. 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.

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 a 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

```c
#define MINI_THREADSPerRow 16

template<int TRow> // threads per row
void operator()(const int threadId) const {
    // For OpenMP or pthreads back-end, compiler will
    // optimize reduction away since TRow=1
    if (TRow == 1) sum += shift_down<1, sum, 1, TRow>();
    if (TRow > 2) sum += shift_down<2, sum, 2, TRow>();
    if (TRow > 4) sum += shift_down<4, sum, 4, TRow>();
    if (TRow > 8) sum += shift_down<8, sum, 8, TRow>();
    if (TRow > 16) sum += shift_down<16, sum, TRow>();
}

// On GPUs only one thread writes result back
if (lane==0) {
    if (doAlpha == 1) m_o(lRow) = sum ;
    else if (doBeta == 1) m_w(lRow) = sum ;
    else if (doBeta == -1) m_w(lRow) = -m_o(lRow) + sum ;
    else m_o(lRow) = beta(0) * m_w(lRow) + sum ;
}
```
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 [40] to use Kokkos for thread level parallelism. MiniMD and miniFE are available through the Mantevo repository and is being used to migrate the Trilinos suite of libraries to manycore architectures. An effort has begun to refactor LAMMPS [40] to use Kokkos for thread level parallelism. MiniMD and miniFE are available through the Mantevo repository at www.mantevo.org.

Acknowledgments

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

References


H. Carter Edwards has over three decades of experience developing software for simulations of a variety of engineering domains. He is an expert in high performance computing (HPC) and is currently focusing on thread-scalable algorithms and data structures for heterogeneous manycore architectures such as NVIDIA GPU and Intel Xeon Phi.

He has a B.S. and M.S. in aerospace engineering from the University of Texas at Austin, and worked for ten years at the Johnson Space Center in the domain of spacecraft guidance, navigation, and control. He has a Ph.D. in computational and applied mathematics, also from the University of Texas at Austin. He has been researching and developing software for HPC algorithms and data structures for the past sixteen years at Sandia National Laboratories.

Christian R. Trott is a high performance computing expert with experience in designing and implementing software for GPU and MIC compute-clusters. He earned a Dr. rer. nat. from the University of Technology Ilmenau in theoretical physics. Prior scientific work focused on computational material research using Ab-initio calculations, molecular dynamic simulations and Monte Carlo methods for investigations of ion-conducting glass materials. As of 2012 Christian is a postdoctoral appointee at the Sandia National Laboratories and is working on developing scientific codes for future manycore architectures.

Daniel Sunderland is an expert in high performance computing who specializes in designing scalable data structures and algorithms for manycore architectures. He earned his master’s from Utah State University in computer science. Since 2009 he has been employed by Sandia National Laboratories developing and maintaining multi-physics engineering codes for current and future HPC architectures.