Kokkos: Enabling performance portability across manycore architectures

H. Carter Edwards and Christian R. Trott
Sandia National Laboratories
PO Box 5800 / MS 1318
Albuquerque NM, 87185

Abstract—The manycore revolution in computational hardware can be characterized by increasing thread counts, decreasing memory per thread, and architecture specific performance constraints for memory access patterns. High performance computing (HPC) on emerging manycore architectures requires codes to exploit every opportunity for thread-level parallelism and satisfy conflicting performance constraints. We developed the Kokkos C++ library to provide scientific and engineering codes with a user accessible manycore performance portable programming model. The two foundational abstractions of Kokkos are (1) dispatch work to a manycore device for parallel execution and (2) manage multidimensional arrays with polymorphic layouts. The integration of these abstractions enables users’ code to satisfy multiple architecture specific memory access pattern performance constraints without having to modify their source code. In this paper we describe the Kokkos abstractions, summarize its application programmer interface (API), and present performance results for a molecular dynamics computational kernel and finite element mini-application.

I. INTRODUCTION

The Kokkos library [1]–[3] provides scientific and engineering codes with a user accessible programming model that enables performance portability across diverse manycore architectures. We define user accessibility by the degree to which we can minimize (1) users’ need to have architecture specific knowledge and (2) pollution of users’ code with parallel directives. We define performance portability by the amount of user code which can be compiled for diverse manycore architectures and obtain the same, or nearly the same, performance as an architecture specialized version of that code.

The vision for Kokkos has evolved from a hidden portability layer for sparse linear algebra kernels in Trilinos [4] to a hierarchy of broadly usable libraries. The current core library and programming model provides two fundamental capabilities: (1) thread parallel execution on manycore devices and (2) multidimensional arrays. Plans for higher level libraries include sparse linear algebra, tensor algebra, containers (e.g., unordered maps), and finite elements.

Kokkos’ thread parallel execution follows the parallel dispatch pattern used by Intel Threaded Building Blocks (TBB) [5], NVIDIA Thrust [6], C++ AMP [7], and others. Kokkos’ multidimensional arrays serve the same role as multidimensional arrays intrinsic to many programming language (e.g., FORTRAN and C), and are similar in concept to the flexible storage ordering in Boost.MultiArray [8]. By combining parallel dispatch and flexible storage ordering in a single programming model we can portably achieve the optimal data access pattern for disparate manycore devices. This concept allows Kokkos to supersede the contemporary manycore programming dilemma of array of structures (AoS) versus structure of arrays (SoA) by choosing the best storage ordering for the given device without having to modify computational kernels.

Kokkos provides a minimal overhead API that isolates user code from device specific programming models. This allows us to choose the most performant programming model for each manycore device and optimize our use of that programming model without impacting user code. Current back-ends for this API are Cuda [9], pthreads [10], and OpenMP [11]. Pthreads and OpenMP back-ends use the portable hardware locality (hwloc) library [12] for explicit thread placement. Back-ends for the Intel Xeon Phi co-processor use the self hosted mode – processes run exclusively on this “device” as opposed to using the offload model.

We evaluate user accessibility and performance portability through unit-level performance tests and miniapplications. Mantevo [13] miniFE and miniMD (finite elements and molecular dynamics) miniapplications [14] have been ported to Kokkos for comparison with architecture-specialized versions. We have found that when using portable Kokkos kernels we typically obtain within 90% of the performance of specialized implementations.

In this paper we describe the semantics of Kokkos’ programming model, summarize the API, and present performance results for the Lennard-Jones force kernel from miniMD and scaling results for miniFE.

II. 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. An HPC application executing in this environment has two levels of parallelism: (1) distributed memory parallelism typically supported through a Message Passing Interface (MPI) library and (2) thread level parallelism on the manycore device. We assume that each MPI process uses at most one manycore device so that Kokkos does not have to manage the interaction of multiple devices.

A. Processes and Devices

Our abstraction of a single MPI process is a master thread that dispatches parallel work to a manycore device. This work
dispatch (a.k.a., asynchronous callback) abstraction is common to numerous programming models; for example, function objects are dispatched to C++ Standard Template Library (STL) algorithms [15], Intel Threading Building Blocks, and Thrust. A key element in this abstraction is that the master thread executes in the CPU host space and worker threads execute in the manycore device space. Note that these spaces are the same when executing on a multicore CPU or self-hosted manycore device.

B. Execution and Memory Spaces

Threads execute in an execution space and data resides in a memory space. Execution spaces have accessibility and performance relationships with a memory spaces. For example, code executing in the Cuda space can access pinned memory in the Host space but with degraded performance compared to accessing memory in the Cuda space.

We address memory accessibility and performance concerns through (1) an explicit execution/memory space abstraction and (2) a design policy to never hide expensive memory copy operations. Each space is defined by a type (a C++ class) so that the execution space of a computation and the memory space of an array are known at compile-time. This enables Kokkos to, at compile time, prevent code executing in the host space from accessing memory in the device space, as opposed to generating a runtime memory fault. When manycore hardware and runtime systems provide virtual unified addressing across memory spaces (e.g., NVIDIA’s use of pinned host memory) we can define additional memory spaces that clearly communicate the execution-memory performance relationship.

III. Multidimensional Array

A Kokkos multidimensional array consists of a homogeneous set of values residing in a memory space, an index space defined by the Cartesian product of integer ranges, and a layout – a bijective map between the index space and the set of values.

Programming languages with multidimensional arrays (e.g., FORTRAN and C) prescribe a layout. As a consequence a computation’s memory access pattern is dictated by array declarations and loop ordering. Thus changing a memory access pattern requires modification of source code.

In contrast, Kokkos multidimensional array layouts are chosen at compile-time resulting in the associated index calculations being inserted at every array access. As such changing the memory access pattern does not require modification of users’ code as long as (1) their code does not assume a particular layout and (2) conforms to the Kokkos API. Lifting the layout from the programming language into Kokkos defines a separation of concerns between user defined index spaces and memory access patterns. A similar separation of concerns is provided by the Boost.MultiArray library.

A. Allocation and Access

Kokkos multidimensional arrays are implemented by the C++ View template class. As shown in Figure 1 the first template argument specifies the value type, number of dynamic dimension denoted by the number of '*' tokens, and static dimensions denoted by '[#]' expressions. A second template argument defines the memory space in which the values of the array are allocated.

```cpp
typedef View<double*[8][3],Device> my_type;
// parentheses operator returns 'const double *'
typedef View<const double*[8][3],Device> my_const_type;

my_array_type a("a",N,M); // Allocate an array
a[i,j,k,l] = value;
```

Fig. 1. Syntax for defining, allocating, and accessing members of a Kokkos multidimensional array.

The View parentheses operator implements the integer arithmetic of the layout and returns a reference to a member value. This operation is valid only if the memory space is accessible to the execution space in which the operator is invoked and the indices are within the index space. A mix of static and dynamic dimensions is supported so that the parentheses operator can be optimized in the presence of static dimensions. For example, the integer arithmetic associated with a static dimension and literal value index can be performed at compile-time.

B. View and Deep Copy Semantics

The class name View is selected to inform and remind users that these objects have view, or shared ownership, semantics as shown in Figure 2. In contrast to C++ standard container semantics, multiple View objects can reference the same allocated array. The allocated array is deallocated when the last view of it is destroyed or reassigned. These semantics are analogous the C++ shared pointer semantics [16].

```cpp
typedef View<double*[8][3],Device> my_type;
// parentheses operator returns 'const double *'
typedef View<const double*[8][3],Device> my_const_type;

my_array_type a("a",N,M); // Allocate an array
{
    // More views of the same array
    my_type a2 = a;
    my_const_type a3 = a2;

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

Fig. 2. View’s shared ownership semantics with last-view delete responsibility.

In view semantics an assignment operator is a shallow copy operation – only the memory reference and array dimensions are copied. A deep copy operation copies member values between two compatible arrays. The deep copy operation is typically used to copy array values between memory spaces, from host to device and vice-versa.

Deep copying between arrays with different layouts has a performance penalty of remapping data and additional penalty
of allocating a temporary array when copying between memory spaces. Since the layouts chosen by default for a GPU view and a CPU view are different this performance penalty would occur frequently. We address this problem by defining HostMirror views (Figure 3) which are in the host memory space but have the device’s layout. Figure 3 shows how an array in device memory space is most efficiently deep copied to/from an array in the host 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 host space with a layout mirroring
// 'my_array_type'. If the device != host then
// 'create_mirror_view' allocates a compatible
// array, otherwise the input view 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.
// Copy device <- host
deep_copy( host_a, a ); // Copy host <- device
```

![Fig. 3. Deep copy performance penalties associated with remapping array layouts are avoided by using HostMirror views that have the same layout as a device view but with member values residing in the host space.](image-url)

### C. Advanced Performance Tuning Features

In the previous sections we described fundamental capabilities for device-aware multidimensional arrays. Kokkos supports additional data access performance tuning features through an optional advanced API. These features leverage extension points in Kokkos’ software design.

Multidimensional arrays have a default layout chosen for each type of device. A user may override this default layout by instantiating a View template with a specified layout. In addition an advanced user may develop and specialize their arrays with their own layouts.

A device may provide special hardware or runtime features to tune memory access performance. For example, NVIDIA devices have texture cache which can improve performance of random read access of an array. We define traits for the functional characteristic of the memory access pattern and then specialize the View class to use appropriate hardware / runtime features when available.

In Figure 4 arrays are declared with both a specified layout and a memory access trait. In this example the `read_x` view has `const` members and `RandomRead` traits. If the device is Cuda then the implementation of `read_x` is specialized to use NVIDIA texture cache to speed up random memory read access. Otherwise the default implementation will enforce the `const` condition but will not use special hardware.

```cpp
my_multivector x("x",N,M);

// Specify the layout and allocate an array:
typed View<
    double**,
    LayoutRight,
    Device > my_multivector;

// Define a View which is optimized for
// random read operations.
// Perform a shallow-copy to that view.
typed View<
    const double**,
    LayoutRight,
    Device

    RandomRead > read_x = x ;

// If Device == Cuda then the access operator
// uses NVIDIA texture cache functionality.
value = read_x(i,j));
```

![Fig. 4. A View with special memory access traits will use available device hardware or runtime features to optimize access for that trait.](image-url)

### IV. Parallel Execution

Parallel execution patterns [17] 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 via parallel_for and parallel_reduce operations. We plan to enhance Kokkos to include parallel_scan operation and hierarchical task-data parallelism where interdependent data parallel tasks are scheduled to execute on the manycore device.

A data parallel operation maps `NWork` independent units of work onto threads for execution on the manycore device. Units of work are completely independent if they do not depend upon data updated by a different unit of work and do not update 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` update the same data via global or local reduction operations. In Kokkos global reductions (e.g., an inner product) are supported by the parallel_reduce operation. Local reductions (e.g., a map reduce) are supported through atomic updates.

#### A. Parallel For Functor API

In C++ a functor is an instance of a C++ class that contains a callback function, shared parameters, and references to data upon which the callback function operates. A parallel_for functor has a work callback, shared input parameters, and views to arrays that are operated on. A data parallel work functor 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 leading (left-most) index of an array is the parallel work index.

The C++ implementation of a parallel_for functor must conform to two simple requirements illustrated in Figure 5: (1) identify the execution space of the functor and (2) provide a work callback. It is recommended that the functor has the execution space as a template parameter for portability.

#### B. Parallel Reduce Functor API

A parallel_reduce functor has a work callback, a reduction callback, shared input parameters, views that are operated on, 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.
The parallel_reduce functor API is designed so that Kokkos can provide scalable and deterministic global reductions. For large thread counts the global reduction follows a traditional fan-in algorithm \((NT)\) fan-in algorithm \((NT = \text{number of threads})\). The fan-in algorithm requires thread-locale copies of the reduction parameters which are reduced to a single global value through an ordered sequence of concurrent pair-wise reductions. This ordered sequence is derived from the number of threads \(NT\) and number of work items \(NWork\), and guarantees a deterministic result given the same \(NT\) and \(NWork\).

Requirements for a reduction callback API are given by example in Figure 6. First, the reduction parameters must be defined through a C++ type satisfying the plain old data type conditions; e.g., a bit-wise copy of values will yield the correct result. This is typically a simple intrinsic type such as ‘double’. Second, the reduction callback consists of two functions with interface requirements to support correct inter-thread communication. These requirements are defined by example in Figure 6.

### C. Local Parallel Reductions via Atomics

Kokkos supports local parallel reductions through atomic reduction operations; e.g., atomic addition. An atomic operation serializes concurrent updates to a values but does not guarantee the ordering of these updates among threads. Thus a non-associative reduction operation (e.g., floating point addition) can yield non-deterministic results for local parallel reductions.

Atomic operations’ serialization can introduce scalability bottlenecks. Typically atomic operations should only be used when the number of atomic updates to a particular variable 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.

### V. Performance Evaluation

We use the Lennard Jones force calculation (LJ-kernel) extracted from our molecular dynamics mini-application MiniMD and the finite element proxy code miniFE for two performance evaluation tests. All tests are carried out on our Compton and Shannon testbed clusters, with details of their respective configurations given in Table I. Compton is used for Xeon and Xeon Phi tests and Shannon is used for Kepler GPU tests.

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.
A. Molecular Dynamics Force Kernel

The LJ-kernel shown in Figure 7 loops over atoms and calculates the forces between neighboring pairs of atoms with a distance $d_{ij}$ being smaller than a cutoff $r_{cut}$. To that end a list of neighbors $j$ for each atom $i$ is precomputed and then used in the kernel.

```c
// 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;
    // Iterating 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[3] = f_i[0..2];
}
```

Fig. 7. Pseudo code for the thread safe Lennard Jones molecular dynamics kernel (LJ-kernel) in MiniMD.

We ran miniMD’s default performance test problem with 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 computation neighbors($i$, $j$) has a regular memory access pattern and $x(j, 0-2)$ has a random memory access pattern. Even so, when atoms are ordered in memory according to spatial location, it is possible to achieve a very high cache reuse for $x(j, 0-2)$. This reuse drastically limits the actual number of loads from main memory.

The importance of layouts in Kokkos is highlighted in the load of the neighbor index $j = \text{neighbors}(i,jj)$ on Xeon (and Xeon Phi) this should be a LayoutRight (row major ordering) so that the loaded cache line contains the values for the next iteration step of the inner loop over $jj$. On a Kepler GPU LayoutLeft (column major ordering) should be used so that the load is contiguous for threads working on different atoms $i$. Furthermore on Kepler GPUs it is important to use texture fetches (via RandomRead attribute) for $x(j, 0-2)$ random accesses.

Figure 8 shows performance in GFlop/s for the kernel on our Compton and Shannon testbeds using a single node for both the optimal code and when using the wrong memory layout on each hardware. The latter causes a performance drop of 1.9x, 3.4x and 6.6x on the Xeon, Xeon Phi, and Kepler GPU testbeds. Also, using the correct layout but not using texture fetches results in a 3.6x slowdown for the Kepler GPU.

<table>
<thead>
<tr>
<th>Name</th>
<th>Compton</th>
<th>Shannon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>CPU</td>
<td>2x Intel E5-2670 HT-on</td>
<td>2x Intel E5-2670 HT-off</td>
</tr>
<tr>
<td>Co-Processor</td>
<td>2x Intel Xeon Phi 57c 1.1GHz</td>
<td>2x K20x</td>
</tr>
<tr>
<td>Memory</td>
<td>64 GB</td>
<td>128 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>QDR IB</td>
<td>QDR IB</td>
</tr>
<tr>
<td>OS</td>
<td>RedHat 6.1</td>
<td>RedHat 6.2</td>
</tr>
<tr>
<td>Compiler</td>
<td>ICC 13.1.2</td>
<td>GCC 4.4.6 + CUDA 5.5 RC</td>
</tr>
<tr>
<td>MPI</td>
<td>IMPI 4.1.036</td>
<td>MVAPICH2 1.9</td>
</tr>
</tbody>
</table>

**TABLE 1. CONFIGURATIONS OF TESTBED CLUSTERS.**

We achieve similar fractions of the theoretical peak flop rate on both Xeon (14%) and Kepler (17%) testbeds; however, on the Xeon Phi we achieve only 5% of peak. This small percentage of peak on the Xeon Phi is obtained even with a well optimized OpenMP-based version of miniMD.

B. MiniFE

MiniFE is a hybrid parallel (MPI+X) finite element mini-application that constructs a linear system of equations for a 3D heat diffusion problem and performs 200 iterations of a conjugate gradient (CG) solver on that linear system. It is designed to capture a number of important characteristics of implicit parallel finite element codes. MiniFE has been implemented in various programming models some of which are available at mantavez.org.

We compare the performance of miniFE-Kokkos (portable variant) with miniFE-OpenMP running on the Xeon and Xeon Phi testbeds, and with miniFE-Cuda running on the Kepler GPU testbed. The miniFE-Kokkos back-end for Xeon and Xeon Phi is OpenMP and the back-end for 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. Since the majority of miniFE optimization efforts have concentrated on the CG-solver, we focus on this phase of miniFE performance.

Our miniFE test case is a weak scaling problem with 8M elements per compute node or device and requires 3.3GB of main memory per node. Tests are run with a single MPI process per device, except for Xeon tests with miniFE-OpenMP which runs with one MPI process per NUMA region. We make this exception because when miniFE-OpenMP is run with one MPI process per node the execution time more than doubles. This slowdown is due to the problem construction phase performing an implicit NUMA first touch on the linear...
system that is incompatible with the access pattern of the CG-solve phase. Consequently threads regularly access memory in the wrong NUMA domain with the associated bandwidth penalty. Kokkos transparently handles this NUMA issue by running a parallel for first touch initialization of each new data allocation that is compatible with typical data parallel kernels. For Kepler tests we use the MVAPICH2 1.9 [18], [19] implementation of MPI to enable GPU-Direct capabilities; i.e., MPI can directly access GPU memory. Thus no explicit data copies are necessary between device and host during the CG-solve.

In Fig. 9 timings for the CG-solve phase are shown from a weak-scaling study on our Xeon, Xeon Phi, and Kepler GPU testbeds (Table I). For each data point the best time out of 12 runs was used. Overall, Kokkos delivers similar performance as the native implementations. 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. Both on Xeon and Kepler GPU testbeds excellent scaling is observed, with miniFE-Kokkos having about 95% parallel efficiency with 32 MPI ranks. MiniFE-OpenMP shows slightly worse scaling efficiency, which is likely due to using twice as many MPI ranks. 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 soon with a new runtime software stack, so that Xeon Phi based systems should see similar scaling behavior as the Kepler GPU based system. Another problem with the Xeon Phi runs were occasional outliers in the performance. Some of the tests produced timings which were twice as large as the one from the fastest run. The cause for this behavior is currently under investigation.

VI. 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 optimal memory access patterns. We have described these abstractions in detail, summarized the API, and presented performance portability results for a molecular dynamics computational kernels and finite element mini-application. Our test cases achieve at least 90% of the performance of architecture specific, optimized variants of those test cases.

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 to incorporate new manycore capabilities, array layouts, aggregate “scalar” data types, parallel operations, and higher level libraries of data structures and kernels. For example, tiled layouts can be transparently introduced into dense matrices without modification of user code. Similarly, automatic differentiation or stochastic variable types are transparently incorporated into array layouts. Plans for new parallel operations include parallel scan and hierarchical task-data parallelism. Finally, development has begun for higher level libraries such as sparse linear algebra and array-based containers (e.g., hashmaps).

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. 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 SAND2013-5221C.

REFERENCES


