Kokkos 3: Programming Model Extensions for the Exascale Era

Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke

Abstract—As the push towards exascale hardware has increased the diversity of system architectures, performance portability has become a critical aspect for scientific software. We describe the Kokkos Performance Portable Programming Model that allows developers to write single source applications for diverse high-performance computing architectures. Kokkos provides key abstractions for both the compute and memory hierarchy of modern hardware. We describe the novel abstractions that have been added to Kokkos version 3 such as hierarchical parallelism, containers, task graphs, and arbitrary-sized atomic operations to prepare for exascale era architectures. We demonstrate the performance of these new features with reproducible benchmarks on CPUs and GPUs.

Index Terms—Performance portability, programming models, high-performance computing, heterogeneous computing, exascale

1 Introduction

Over the last decade, the High Performance Computing (HPC) hardware landscape has diversified significantly. Currently, GPU-based systems make up six of the ten most powerful HPC systems in the world [24]. However, the top-ranked system is a CPU-only design, deploying ARM-based chips with 512-bit vector units, while other top-10 machines use custom accelerators and even simply Intel CPUs. The GPU landscape is also getting more diverse. Long dominated by NVIDIA alone, the first generation of upcoming exascale platforms will deploy AMD and Intel GPUs instead.

All of this means that it is becoming more difficult to write code which can leverage all of the HPC systems that users have access to. With the lifetime of the most important HPC applications measured in decades, and thus far exceeding the lifetime of any given machine, the demand for performance-portability solutions has exploded [4], [16], [22]. Application developers want to write their code in a way that it can leverage any of the current and future systems without major rewrites of the code itself.

The Kokkos Programming Model offers one such solution for performance portability [9]. Initially developed at Sandia National Laboratories, it is now an open source community project largely funded through the DOE Exascale Computing Project, with a core development team spanning five US National Laboratories. Since the publication of [9], the need to support more complex applications has resulted in significant extensions of the programming model, which are the focus of the current paper. These additions, developed as part of the Kokkos version 3 release cycle, are focused on exposing more parallelism, asynchronicity and advanced hardware capabilities, which are relevant to fully leverage the upcoming exascale era architectures. A guiding principle of the development of these capabilities was to leverage advanced hardware features where we can, without hurting performance on platforms that do not provide the same features set.

Furthermore, a comprehensive Kokkos EcoSystem has been developed on the foundation of the programming model. This ecosystem provides commonly needed capabilities which go beyond a pure programming model, such as linear algebra libraries, tools infrastructure, language interoperability layers and user support. Descriptions of these efforts will be provided elsewhere.

The Kokkos Programming Model provides performance-portability through a *library-based* or embedded-language approach as opposed to a *directive-based approach* (e.g., OpenMP, OpenACC) or a *language-based* approach. Among library-based approaches for performance portability, there are other performance portable libraries such as SYCL, RAJA and OCCA. There is some overlap among the features of these programming models. For example, Hammond *et al.* [12]

- *Christian R. Trott, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Dan Sunderland, and Jeremiah Wilke are with Sandia National Laboratories, Albuquerque, NM 87185 USA. E-mail: {crtrott, jciesko, vqdang, ndellin, eharvey, dshollm, daibane, jsmiles, dzpolia, ajpowel, srajama, dsunder, jjwilke}@sandia.gov.*
- *Damien Lebrun-Grandié, Daniel Arndt, and Bruno Turcksin are with Oak Ridge National Laboratory, Oak Ridge, TN 37830 USA. E-mail: {lebrungrandt, arndtd, turcksinbr}@ornl.gov.*
- *Rahulkumar Gayatri and Jonathan Madsen are with Lawrence Berkeley National Laboratory, Berkeley, CA 94720 USA. E-mail: {rgayatri, jrmadsen}@lbl.gov.*
- *Nevin Liber is with Argonne National Laboratory, Lemont, IL 60439 USA. E-mail: nliber@anl.gov.*
- *Mikael Simberg is with Swiss National Supercomputing Centre, 6900 Lugano, Switzerland. E-mail: simbergm@cscs.ch.*

Manuscript received 25 Feb. 2021; revised 12 May 2021; accepted 16 June 2021. Date of publication 14 July 2021; date of current version 15 Oct. 2021. (Corresponding author: Christian R. Trott.) Recommended for acceptance by S. Alam, L. Curfman McInnes, and K. Nakajima. Digital Object Identifier no. 10.1109/TPDS.2021.3097283

provide a short comparison of the features of Kokkos and SYCL. There are a number of functionality and performance comparisons of Kokkos with other programming models such as RAJA, CUDA, OpenMP and OpenACC [2], [7], [14]. Detailed comparison of all these approaches for productivity, performance, and portability is beyond the scope of this paper.

The primary contributions of this paper are:

- A description of the additions to the Kokkos Core programming model to support exascale systems.
- Demonstration of the flexibility and the performance of the programming model through carefully chosen benchmarks on CPU and GPU architectures.
- Unique functionality such as arbitrary-sized atomic operations in a portable manner.
- A reproducible set of benchmarks and instructions to reproduce them.

In the following, we discuss the primary capabilities of the Kokkos Core Programming Model. First, in Section 3, we provide a short overview of capabilities introduced in a previous paper [9] (for more details on these subjects, please refer to that paper). Then we discuss features added since the publication of [9]. We describe advanced reductions in Section 4. In Section 5, Kokkos' comprehensive support for atomic operations is presented. We continue the description of the Kokkos Core Programming Model with ways of exposing more parallelism in Sections 7 and 8. This is followed by a discussion of execution space instances (Sections 9) and graph constructs (Section 10), which enable users to express programs where the relationships between kernels are more complex than simple sequential dependencies. We also describe the vectorization support via SIMD types in Kokkos in Section 11, before providing a short overview of existing backend support in Section 12. Finally, we provide a small insight into practical performance portability achieved by users of Kokkos, based on a number of studies these users published.

2 Benchmark Reproducibility Information

Performance benchmarks in this paper were conducted with Kokkos release candidate version 3.4 (SHA 31ae2ea3) configured as a release build. We used a Intel Skylake based system with a NVIDIA V100, a Fujitsu ARM A64FX based system, a IBM Power9 based system and a platform with AMD MI100 GPUs and AMD CPUs. Generally CUDA 11.1, Intel 19.0.5, GCC 10.2, GCC 7.2, and ROCM 4.1 provided the compilers on these platforms respectively. Additionally we used the Clang 12 release candidate 3 for OpenMP 5 results on the V100. We utilized the Kokkos CUDA backend for the V100, the Kokkos HIP backend for the MI100 and the Kokkos OpenMP backend (which is OpenMP 3 based) on all CPUs. For OpenMP runs only a single socket of each system is used, with all hardware threads and close placement affinity. Hyperthreads were disabled for the Skylake system.

The code examples used to collect data for performance plots throughout this paper are available at https://github.com/kokkos/code-examples/ in the `papers/kokkos-core-tpds-2021` directory. The benchmarks were run 20 times, and the average was taken as the performance data presented here. Detailed configuration, build and run instructions are provided in the repository given above.

3 Fundamental Capabilities

As described in [9] Kokkos is a performance-portable programming model, implemented as a C++ library, that provides a number of abstractions to express both parallel execution and data structures. These abstractions are generic over a variety of backing hardware-specific execution and memory models.

Kokkos is built around six core abstractions: *Execution Spaces*, *Execution Patterns*, and *Execution Policies* control parallel execution; while *Memory Spaces*, *Memory Layouts*, and *Memory Traits* control data storage and access.

3.1 Execution Spaces

An *Execution Space* is a generic abstraction that represents an execution resource and an associated execution model. Every parallel operation in Kokkos is enqueued into a first-in, first-out queue in an instance of an execution space (*Execution Space Instance*). The resources that these instances encapsulate are execution model entities like threads or handles to a device. For example, each `Kokkos::Cuda` execution space instance encapsulates a CUDA stream, and each `Kokkos::Threads` instance encapsulates a thread pool [18].

Kokkos creates a default instance for each *Execution Space* type, which is used when no specific instance is associated with an operation. Kokkos also allows users to provide additional information if two kernels should be executed by separate instances, but in the absence of that information, it falls back to the assumption that all kernels are sequenced in program order. The latter of these is always safe for a programming model that makes no concurrent forward progress guarantees, but it may lead to less efficient scheduling. Some kernels can use the default instance and others can use a custom one allowing incremental migration.

3.2 Execution Patterns

Parallel kernels in Kokkos are expressed through *Execution Patterns*. They express the relationship between independently executable *work items* that must not contain internal inter-dependencies. The original patterns described in [9] are:

- `parallel_for`: each work item is executed once.
- `parallel_reduce`: each work item is executed once with a provided output object. These outputs are combined according to semantics specified by a reducer object.
- `parallel_scan`: calculates the partial reduction of the output arguments of the work items [15]. Each work item is potentially executed more than once. During final execution each work item is provided its partial reduction result.

The patterns take as arguments function objects, whose call operator defines the *work item* for a given kernel index. C++ lambdas are a compact, user-friendly way to provide such a function object, but Kokkos execution patterns can handle any instance of a callable type that meets the requirements above.

It is important to note that these patterns form a fairly restricted programming model, particularly with respect

to the forward progress relationships between work items. This is by design, since it allows for maximum flexibility in the mapping of these patterns to the underlying execution model. Requiring work items to be independent allows any execution model mapping, including threading, SIMD vector units, and even pipelined execution of work items.

In contrast to Kokkos, other programming models such as OpenMP, CUDA, and HIP often provide detailed guarantees about how work is mapped to hardware resources—typically leading to over-constrained application semantics that lack the information to determine which of these guarantees are actually needed when mapping to diverse execution models [17], [18].

3.3 Execution Policies

Broadly speaking, *Execution Policies* express the *how* of the execution for the pattern. Most importantly, they specify the set of *work item* indices and the *execution space* associated with the kernel. *Execution Policies* can also convey auxiliary information such as the preferred strategy for scheduling work items, hints about the ideal chunk size to use when scheduling work items, and what type to use for work item indices. In [9], only one execution policy was described: the `RangePolicy` used to express contiguous, one-dimensional ranges of work item indices. Here, we expand upon that with execution policies that support multidimensional ranges and hierarchical parallelism.

3.4 Execution Abstraction Example Usage

Using these three execution abstractions, users can describe performance-portable, minimally constrained parallel work. For brevity, this example omits some details that should be included in typical application usage, such as labels used by the Kokkos Tools interface (described elsewhere).

```
// Simple kernel with work items in [0, n)
parallel_for(RangePolicy<Cuda>(0, n),
  KOKKOS_LAMBDA(int i) {/* ... work item i ... */});

// A count of work items is a short-hand for:
//   RangePolicy<DefaultExecutionSpace>(0, n)
parallel_for(n, KOKKOS_LAMBDA(int i) {/*...*/});

// Sum Reduction using a lambda using OpenMP backend
// and an explicit reducer
double result = 0.0;
parallel_reduce(RangePolicy<OpenMP>,
  KOKKOS_LAMBDA(int i, double& s)
  { s += 3.14 * i; }, Sum{result});

// The same thing, but using the output location as
// a shorthand for the reducer, since Sum is the
// default reducer for values of type double:
parallel_reduce(n, KOKKOS_LAMBDA(int i, float& s)
  { s += 3.14 * i; }, result);

// compute the inclusive scan values of the first n
// even numbers
parallel_scan(n, KOKKOS_LAMBDA(int i,
      int64_t& partial_sum, bool final) {
  partial_sum += 2 * i;
  // array is a Kokkos::View, introduced below
  if(final) array(i) = partial_sum;
});
```

3.5 Memory Spaces

Memory Spaces represent abstract memory resources, abstractions that are analogous to those for *Execution Spaces*. Fundamentally, these abstractions generically encapsulate a mechanism for allocating and managing memory on a particular hardware resource. *Memory Spaces* also abstract away the mechanism for moving data between memory resources, accessed by the user through the `deep_copy` function, and provide information on the set of *Execution Spaces* that can access the resource they represent (accessed by the user through the `SpaceAccessibility` type trait).

Since *Memory Spaces* abstract both a memory resource *and* an allocation mechanism, multiple Kokkos *Memory Spaces* can represent different modes of access to the same physical resource. For example when compiling for NVIDIA hardware, Kokkos provides `CudaSpace`, `CudaUVMSpace`, `CudaHostPinnedSpace` and `HostSpace` as implementations of the memory space abstraction. The latter two actually represent the same physical memory, but `CudaHostPinnedSpace` allows GPU access, while `HostSpace` does not.

3.6 Memory Layouts

Memory Layouts define the mapping of multidimensional array indices to storage location. Because memory layouts are a first-class abstraction in Kokkos, domain experts can write algorithms that are generic over data layout, allowing the backend to control data access patterns without complicating the algorithmic logic. On GPUs, workers behave more like vector lanes—generally performing better when data accesses are coalesced. On architectures where each worker has its own cache, however, data accesses need to be grouped by worker to maximize cache reuse. Sometimes which data layout is optimal may even change from one generation of a hardware to the next due to changes in cache sizes, and thread behavior. The primary data layouts provided by Kokkos are `LayoutLeft` (i.e., Fortran style Layout), `LayoutRight` (i.e., C-Style Layout) and `LayoutStride` (i.e., arbitrary regular strides).

3.7 Memory Traits

Another abstraction are *Memory Traits* which allow one to convey additional information about desired memory access behavior. This abstraction can cover semantic information—such as whether an access to a data structure should be atomic or is guaranteed to not be aliased through accesses in another data structure. Additionally, performance-critical, non-semantic information—such as that accesses through a specific handle are expected to hit random memory locations—can be conveyed. Availability of this information allows Kokkos to leverage special data access paths in hardware such as texture fetches, atomic units, or non-temporal load instructions when available.

3.8 View

The primary data structure in Kokkos to bring the three data abstractions together is `View`. It is intended as the

lowest level fundamental data structure replacing pointers and runtime-sized arrays. At its heart, `View` behaves like a multi-dimensional shared pointer; i.e., it algorithmically represents a multi-dimensional array and has shared ownership semantics with reference counting and automatic deallocation. `Kokkos::View` is parameterized on *Memory Space*, *Memory Layout* and *Memory Traits*, allowing generic code to reason about the properties of the fundamental data objects. The primary template parameter is the *data type* expressing both the scalar values and the dimensionality, or `rank` of the `View`. Both runtime and compile-time extents for dimensions are supported.

```
// a View in the default memory space with
// 2 runtime dimensions
using my_matrix = View<double**>;
auto x = my_matrix{"X", N, M};
// a constant slice of just one column of a
auto x_5 = subview(my_matrix::const_view_type{x},
    ALL, 5);
// 3D View in CudaUVMSpace with a compile time size
auto b = View<int**[5], CudaUVMSpace>{"B", N, M};
// Atomic access to b:
auto b_atomic = View<int**[5], CudaUVMSpace,
                     MemoryTraits<Atomic>>{b};
```

4 Advanced Reductions

In Edwards *et al*. [9], the ability to express custom reduction semantics operated via the use of `join` and `init` member functions of the user-provided work item functor. This approach poorly orthogonalizes the expression of work items and the means of combining results. Thus, Kokkos later introduced the concept of *Reducers*. A *Reducer* is an instance of a (potentially user-defined) type that holds a reference to the output location, and defines how to initialize and combine reduction results. Kokkos provides several implementations of common `Reducers`, similar to the reduction operations defined in MPI. In Kokkos, we provide the following reducers: `Sum`, `Prod`, `Min`, `Max`, `MinMax`, `MinLoc` (minimum with location), `MaxLoc`, `MinMaxLoc`, `LAnd` (logical and), `LOr`, `BAnd` (bit-wise and), and `BOr`. A simple example that uses *Reducers* to determine the maximum value of a `View` a:

```
int max_of_a = 0; // initial value unused
parallel_reduce(n,
  KOKKOS_LAMBDA(int i, int& my_max) {
    if(a(i) > my_max) my_max = a(i);
  }, Max<int>{max_of_a});
```

Kokkos also allows users to give multiple reductions at the same time. For example, the following snippet computes both the maximum value and the sum of the entries in a:

```
int max_of_a = 0;
int64_t sum_of_a = 0;
parallel_reduce(n, KOKKOS_LAMBDA(int i,
    int& my_max, int64_t& my_sum) {
  if(a(i) > my_max) my_max = a(i);
  my_sum += a(i);
}, Max<int>{max_of_a}, Sum<int64_t>{sum_of_a});
```

The *Reducer* concept is flexible enough that the variadic reduction overload implementation can simply delegate to the single reducer overload by creating a combined reducer and a wrapper for the work item functor. Table 1 demonstrates the performance benefit of performing multiple reductions in a single parallel operation, as opposed to doing each reduction individually. We also compare to equivalent implementations using OpenMP.

TABLE 1
Time in us for Performing a *min* and a *sum* Reduction as Two `parallel_reduce` Calls (*min+sum*) and as a Single `parallel_reduce` With Two *Reducer* Arguments (*min/sum*)

	V100		SKX		A64FX		P9		MI100	
	KK	OMP	KK	OMP	KK	OMP	KK	OMP	KK	OMP
min	37	141	37	54	66	58	73	87	86	443
sum	33	135	82	10	140	140	84	78	83	248
min+sum	70	276	119	64	205	206	157	165	169	691
min/sum	51	155	83	23	145	145	169	118	98	616

The array is of length 1,000,000 with a scalar type of `double`.

Another extension of the reduction interface allows users to provide a `View` of data as the result argument, either directly or via a reducer. If a `View` is provided instead of a scalar value, the reduction operation will execute asynchronously and a `fence` is required to guarantee that the operation completed.

```
auto result = View<int>{"result"}; // a scalar View
// Asynchronous invocation; result may not contain
// the output after this call
parallel_reduce(n, KOKKOS_LAMBDA(int i, int& my_sum)
  { my_sum += i; }, result);
// Requires fence to guarantee that result contains
// the output of the reduction:
fence();
```

5 Generic Atomics

Edwards *et al*. [9] discussed only basic atomic support. Even at that point, Kokkos already abstracted over machine-specific atomic operations. We have since also implemented arbitrary-sized atomic operations. Previously limited to types where arbitrary atomic operations could be implemented using atomic compare-and-swap (CAS) operations, Kokkos now supports atomic operations on distinct objects of arbitrary size. Depending on the target hardware and the size of the operands, these operations either map directly to atomic instructions, are implemented via CAS loops, or use a sharded lock table.

The sharded lock table takes the address of the atomically-accessed memory location, and then uses a hash to compute an index into a globally accessible lock table. Particular care must be taken to avoid deadlocks when implementing this approach on GPUs without a strong forward progress guarantee for divergent warp lanes (e.g., NVIDIA GPUs before Volta and AMD GPUs). Specifically, all active lanes in a warp must participate in the spin lock until each lane has completed the associated atomic operation. In this way, no additional warp divergence is introduced by the lock acquisition—i.e., the successful acquisition of the lock does not have an `else` branch—and thus forward progress is still guaranteed.

TABLE 2
Performance for a Scatter-Add Algorithm Using Atomic Operations With Different Scalar Types

	V100		SKX		A64FX		P9		MI100	
	KK	CUDA	KK	OMP	KK	OMP	KK	OMP	KK	HIP
int	28	25	0.31	0.30	0.165	0.28	0.86	1.07	26	23
double	27	24	0.29	0.29	0.11	0.11	0.68	0.74	4.7	3.7
c<float>	3.9	X	0.26	X	0.11	X	0.5	X	4.8	X
c<double>	2.2	X	0.09	X	0.1	X	0.35	X	3.5	X

10e6 atomic additions are performed on random cells in a 20x20x20 grid. Numbers reported are Giga-Updates per second. c<T> stands for complex<T>.

```
T atomic_fetch_oper(T* ptr, T val, auto apply) {
  unsigned mask = /* get a mask with 1 bits for */
                  /* lanes that reach this point */;
  unsigned nactive = /* number of 1 bits in mask */;
  unsigned ndone = 0;
  bool this_lane_done = false;
  T old_val = {};
  // Loop until every active lane is done
  while(ndone != nactive) {
    // If this lane is not done, try to acquire lock
    if(!this_lane_done) {
      // Try to acquire lock
      if(lock_address(ptr)) {
        memory_fence(memory_order_acquire);
        // load the value and do the operation
        old_val = *ptr;
        *dest = apply(old_val, val);
        memory_fence(memory_order_release);
        // unlock
        unlock_address(ptr);
        this_lane_done = true;
      }
    }
    ndone = BALLOT(mask, unsigned(this_lane_done));
  }
  return old_val;
}
```

In summary, the introduction of sharded lock tables for arbitrary-sized atomics enables users to write less constrained generic code that uses atomic semantics. This is particularly useful for applications that need to use complex<double> on platforms that do not have 128-bit atomic operations. To the best of our knowledge, ours is the first implementation of atomic operations for types of arbitrary sizes on GPUs.

In Table 2 the performance of a scatter-add algorithm [23] with atomic operations is shown for int, double, complex<float> and complex<double> operands. The algorithm mimics a discretization algorithm, where randomly distributed particles contribute some value to a contiguous field. For the experiment, 10 million particles contribute to the grid cell corresponding to their position in a 20x20x20 grid. On GPUs the int (and on V100 also double) atomic operations are supported with special hardware units which provide significant higher throughput than the CPUs. However, even for atomics of complex the GPUs provide about an order of magnitude higher performance than CPUs. Of particular note though is the fact that the lock based approach for complex<double> generally still achieves about half of the throughput of compare-and-swap based atomic operations. There are no performance numbers for the native models provided with complex scalar types, since none of CUDA, HIP and OpenMP support such atomic operations.

6 Containers

View is the only data structure introduced in [9]. However, applications often develop higher level data structures based on View. To avoid duplicated effort, the Kokkos Core Programming Model provides the *containers* library that collects optimized implementations of several common higher-level data structures. As such, the role of the *containers* library is analogous to the role that the C++ standard library fulfills in the C++ standard. In the following, we introduce only the most commonly used data structure, ScatterView, in detail.

6.1 ScatterView

ScatterView facilitates implementing performance portable *Scatter Contribute* algorithms. A *Scatter Contribute* algorithm is an algorithm where one entity contributes to values owned by multiple other entities [23]. In the presence of concurrent execution, this may result in write conflicts. Examples are particle codes where each particle contributes force to every particle within a certain distance, matrix assembly routines in finite element codes where multiple elements contribute to the same entries in the matrix, or generally stencil operations where each cell contributes something to every cell within its stencil.

When parallelizing such algorithms with CPU threads, it is common practice to resolve the write conflicts through data replication strategies. However, the data replication approach is not scalable - on GPUs, where thread counts exceed 100,000, it would exhaust memory capacities. For that reason, atomic operations are generally used to resolve the write conflicts on GPU architectures [1]. While using atomic operations would work on CPUs as well, they often result in slower algorithms than the data replication approach. The primary reason is that on GPUs, atomic operations are resolved in special function units attached to the common cache, while on CPU architectures atomic operations require cache lines to move between cores.

ScatterView abstracts over the two approaches. Within the computational algorithm one contributes to ScatterView as if contributing sequentially to an array. Internally, these contributions will then either map to a per-thread copy of the data, or use atomic operations. After the computational kernel is finished, the user can then request that all values are contributed back into the root copy. When ScatterView uses atomic operations that final step is a no-op.

To help amortize the cost of mapping to the correct data replication, ScatterView requires users to get an accessor inside the kernel. That accessor is then either the equivalent of a subview with the Atomic memory trait or a subview to the thread-private copy of the data.

TABLE 3
Performance for a Scatter-Add Algorithm Using `ScatterView` With Different Scalar Types

	V100		SKX		A64FX		P9		MI100	
	KK	CUDA	KK	OMP	KK	OMP	KK	OMP	KK	HIP
`int`	27	25	3.7	3.6	0.65	0.65	2	2.4	22	23
`double`	26	24	3.3	3.2	0.65	0.65	1.8	2.3	4.5	3.7
`c<float>`	3.9	X	3.3	3.2	0.65	0.65	1.8	2	4.9	X
`c<double>`	2.2	X	2.7	2.7	0.64	0.64	1.8	1.9	3.5	X

10e6 updates are performed on random cells in a 20x20x20 grid. Numbers reported are Giga-Updates per second. `c<T>` stands for `complex<T>`.

```
// wrap a view of doubles into a scatter view
ScatterView<double*> sv(data);
parallel_for(N, KOKKOS_LAMBDA(int i) {
  auto accessor = sv.access();
  for(int j = 0; j < neighbors(i); j++) {
    // Compute something
    // Contribute to the correct neighbor
    accessor(index(i, j))+=value;
  }
});
sv.contribute(data);
```

Similar to reductions, ScatterView can take a template argument to allow different types of contributions, such as minimum or maximum. Furthermore, users can explicitly overwrite whether to use atomic operations or data replication. The defaults are chosen based on the targeted *Execution Space* associated with the *Memory Space* of the `ScatterView`.

In Table 3, the performance of a scatter-add algorithm using `ScatterView` is shown for a `int`, `double`, `complex<float>`, and `complex<double>`. This is essentially the same benchmark as in Section 5, except it uses a `ScatterView` for the accumulation instead of atomic operations. On the V100, the performance is very close to the atomic operations since `ScatterView` uses its atomic implementation path by default for GPUs. On CPUs, performance is significant increased, compared to using atomic operations.

7 MDRangePolicy

A common use case not covered by the initial Kokkos release is iterating over a multi-dimensional space.

A simple example is the initialization of multi-dimensional `Kokkos::View`. Simply parallelizing over one dimension and then iterating serially over the others often does not expose enough parallelism to leverage all architectures [8], [19]. Consider a 3D View of extents $200 \times 200 \times 200$. Parallelizing over a single dimension would only provide work for 200 threads, which is a small fraction of a GPU's 10,000s of threads. A better implementation would recompute the individual indices from a global index of a flattened iteration space, and thus expose 8 million-way parallelism:

```
View<int***> my_view("A", N0, N1, N2);
parallel_for(N0*N1*N2, KOKKOS_LAMBDA(int ii) {
  int i0 = ii / (N1*N2);
  int i1 = ii % (N1*N2) / N2;
  int i2 = ii % N2;
  my_view(i0, i1, i2) = ii;
});
```

However, this is both non-intuitive and hard-codes a specific mapping of 1D indices to 3D space that is hidden from Kokkos. Depending on the *MemoryLayout* of `my_view` this mapping order could be good or bad. The particular order hard-coded here corresponds to a `LayoutRight` iteration order, and thus would lead to good performance on CPU-like architectures, and bad performance on GPUs.

TABLE 4
Performance Comparison of a 200x200x200 Tensor Add Operation With Three Different Implementations: Parallelizing Over a Single Dimension (Outer), Flattening the Index Space (Flattened), and Using an `MDRangePolicy` (MDRange)

	V100		SKX		A64FX		P9		MI100	
	KK	CUDA	KK	OMP	KK	OMP	KK	OMP	KK	HIP
Outer	11	11	107	108	360	680	195	242	5.5	5.5
Flattened	43	45	54	60	43	44	60	60	87	87
MDRange	692	678	107	23	411	293	255	277	468	469

Numbers reported correspond to the bandwidth achieved in GB/s.

To resolve this issue Kokkos introduces the `MDRangePolicy`. It provides the capability of describing a multi dimensional iteration space, and incorporates support for different iteration schemes. Furthermore, `MDRangePolicy` directly supports tiling strategies to optimize data accesses for stencil operations [3].

`MDRangePolicy` takes a template parameter `Rank`, which is a struct that is templated on the rank, iteration order over tiles, and the iteration order within a tile.

```
MDRangePolicy<Rank<3,Iterate::Left,Iterate::Right>>
```

This policy will iterate over a 3D space; *within the tile*, it will iterate fastest over the left-most index, but it will iterate *over tiles* fastest on the right-most index. The iteration order can be omitted in which case they are chosen based on the architecture. The default values match the default Layouts for `View`. The iteration space is provided via constant size arrays as begin and end values. Optionally, tile sizes can be provided. Rewriting the above example with `MDRangePolicy` leads to the following code, which avoids hard-coding an iteration order and thus will perform well when compiled for either CPU or GPU like architectures.

```
auto my_view = View<int***>("A", N0, N1, N2);
auto p = MDRangePolicy<Rank<3>>(
  {0,0,0}, {N0,N1,N2}, {4,8,4});
parallel_for(p,
  KOKKOS_LAMBDA(int i0, int i1, int i2) {
  my_view(i0, i1, i2) = i0*N1*N2 + i1*N2 + i2;
});
```

In Table 4 the impact of the `MDRangePolicy` is demonstrated for a 3D tensor add (i.e., $A = A + B$). Using an `MDRangePolicy` is in this case strictly faster than parallelizing just the outer loop or flattening the index space irrespective of architecture. The tensors are $200 \times 200 \times 200$ large, and thus have a total memory size of 128 MB - well outside the cache limit. On the GPU, parallelizing only over the outer index range is over 60 times slower than using an `MDRangePolicy`, since not enough parallelism is exposed. The flattened index space approach is still significantly slower due to inefficient data access and the use of costly integer division and modulo operations. On CPUs, it is sufficient to parallelize over the outer dimension, since there are only a few tenths of threads. The

flattened space is again slower, likely due to the indexing scheme obscuring the data access pattern for the prefetcher. Generally the native model implementations provide performance similar to the Kokkos variant. The two exceptions are the native OpenMP MDRange equivalent using the collapse clause, which provides poor performance with the Intel compiler, and the outer loop parallization strategy on ARM which appears to provide significantly higher performance.

8 Hierarchical Parallelism

While `MDRangePolicy` can help in many cases, it only works for *tightly nested loops*, i.e., loops that do not have code in between the loop statements.

One simple example of a non-tightly nested loop is a (dense) matrix-vector product.

```
for(int i = 0; i < n; i++) {
  double sum_i = 0.;
  for(int j = 0; j < m; j++) {
    sum_i += A(i, j) * x(j);
  }
  y(i) = sum_i;
}
```

One could rewrite the above code to have two tightly nested loops by directly summing into `y(i)` in the inner loop, but parallelizing that algorithm with an MDRangePolicy would result in a race condition when two threads working on the same index `i` but different index `j` access `y(i)`.

8.1 Basic Thread Teams

Kokkos addresses this problem by enabling parallelization of nested loops through *hierarchical* parallelism [8], [19]. To that end, Kokkos introduces two new execution policies: `TeamPolicy` and `TeamThreadRange`. `TeamPolicy` is used at the outer parallelization level. When using a `TeamPolicy`, instead of a single thread handling an iteration, each iteration will be assigned to a team of threads, which are then able to parallelize nested loops. In this case, the call operator of the lambda or functor is passed a handle to the team rather than an index.

```
using team_t = typename TeamPolicy<>::member_type;
parallel_for("MatVec", TeamPolicy<>(n, AUTO),
  KOKKOS_LAMBDA (const team_t& team_h) {
  double sum_i = 0.;
  int i = team_h.league_rank();
  parallel_reduce(TeamThreadRange(team_h, m),
    [&] (int j, double& my_sum) {
    my_sum += A(i, j) * x(j);
  }, sum_i);
  y(i) = sum_i;
}
```

This team handle provides access to team level functions, such as team synchronization and obtaining the index of the team, the rank of a thread within the team, and the team size.

The `TeamThreadRange` is used as the policy for execution patterns to parallelize nested loops. The interface for nested parallel patterns is otherwise the same as those already introduced, except that labels are not supported.

For the above example of a matrix vector multiply, the natural mapping is to assign a team to each row of the matrix, which then computes the nested reduction:

The `TeamPolicy` takes two arguments: the number of teams to be launched, and the team size. The number of teams to be launched represents an iteration range, and is not limited by the actual concurrency of the targeted hardware.

The team size, on the other hand, is limited by implementation details. In the above example, the `AUTO` parameter indicates that the choice of team size is left to the Kokkos runtime. The actual team size is determined either through heuristics, or potentially even through auto tuning tools. The tools integration into Kokkos will be described in a different paper. The current heuristics in Kokkos can take various properties into account, such as register utilization, shared memory usage, number of teams requested, and properties of the hardware. On GPUs the heuristic tries to maximize the number of concurrently active threads on the GPU (i.e., the occupancy). If multiple team sizes yield the same occupancy, the smallest is chosen in order to maximize flexibility for the GPU scheduler, and provide more opportunities for load balancing.

8.2 Team Synchronization Semantics

It is useful to think of a team as a collection of threads which share a common cache. In practice, that means that a team in Kokkos maps on GPUs to a CUDA block or a HIP group, and to a set of threads running on the same core or socket on CPUs. We define teams as such to enable faster coordination within a subset of the larger threadpool. One example that requires such coordination is an algorithm in which iterations of a nested loop depend on the completion of a prior nested loop, necessitating a barrier between the loops.

```
parallel_for("MatVec", TeamPolicy<>(N, AUTO),
  KOKKOS_LAMBDA ( const team_t& team_h ) {
  parallel_for(TeamThreadRange(team_h, K),
    [&] (int i) {
    // Fill some buffer
  });
  // Wait for every thread in the team to be done
  team_h.barrier();
  double sum;


  parallel_reduce(TeamThreadRange(team_h, M),
    [&] (int j, double& lsum) {
    // Access the buffer randomly
  }, sum);
  // Do more loops and other things
});
```

It is important to recognize that the nested loops are not actually fork-join loops. All threads in the team are active upon entry into the outer parallel pattern. Nested loops simply split the iteration range in some implementation-defined way. This implies that a thread can enter and exit the nested loop independent of other threads in the team. Consequently, without the `barrier`, some threads could execute the second loop, while others are still working on the first.

Another important semantic of Kokkos' thread teams is that Kokkos does NOT give a forward progress guarantee. In fact Kokkos does not even guarantee that threads execute at the same time. A legal implementation strategy for Kokkos' hierarchical parallelism is to split the loop based on synchronization points, and then express threads through pipelining each stage. These semantics provide maximum flexibility for Kokkos to

map hierarchical parallelism to various architectures. However, it also means that users can only use Kokkos' synchronization mechanisms. They can not legally write their own. Specifically one can not write spin wait loops using Kokkos.

8.3 Vector Level Parallelism

Kokkos Hierarchical Parallelism exposes a *vector* level of parallelism, in addition to *teams* and *threads*. This level of parallelism can be accessed using the `ThreadVectorRange` policy with nested patterns. To request vector level parallelism, a third argument is given to the `TeamPolicy`, that can be an integer or `AUTO`. The vector length is bound by hardware resources, and the maximum value can be queried in the TeamPolicy. On GPUs, this level of parallelism is mapped to threads within a warp/wavefront. When the vector length is less than the native warp/wavefront size, multiple Kokkos *threads* are mapped to a single warp/wavefront. Parallel patterns with a `ThreadVectorRange` can be nested inside execution patterns using the `TeamThreadRange`.

A third policy for nested patterns is `TeamVectorRange`. Patterns using this policy will leverage both thread and vector parallelism.

Nested patterns can be `parallel_for`, `parallel_reduce` and `parallel_scan`. Kokkos does not impose any limit on the number of loops that can be parallelized with nested patterns.

```
parallel_reduce("Label", TeamPolicy<>(N, AUTO, 8),
 KOKKOS_LAMBDA(const team_t& team_h, int& team_sum){
 int i = team_h.league_rank();
 parallel_for(TeamVectorRange(team_h, K),
  [&] (int j) {
    // fill buffer
 });
 team_h.barrier();
 // do stuff here
 parallel_scan(TeamThreadRange(team_h, M),
  [&] (int j, double& scanval, bool final) {
  // more stuff
  parallel_reduce(ThreadVectorRange(team_h, K),
    [&] (int k, double& lsum) {
    // even more stuff

  });
 });
 team_h.barrier();
 int j = team_h.team_rank();
 parallel_for(ThreadVectorRange(team_h, count(j)),
  [&] (int k) {
   // do something for i,j,k
 });
});
```

8.4 Team Scratch Memory

The `TeamPolicy` also allows the acquisition of scratch memory. It can be used to leverage explicit on-die scratch pads, such as the *shared memory* on GPUs. Scratch memory has to be requested at dispatch time, as is the case for HIP and CUDA shared memory. Kokkos allows scratch memory to be requested on a per-thread or per-team basis. This contrasts with the HIP and CUDA requirements, where such requests must happen on a per-team basis

Kokkos provides two levels of scratch pad. Level 0 maps to on-die cache, which is generally not more than a few tens of kilobytes. Level 1 maps to general memory, and thus allows much larger scratch allocations.

While Kokkos allows users to utilize scratch memory via raw pointers, generally scratch memory is used in conjunction with `Views`. To that end, each *ExecutionSpace* has an associated `scratch_space`, used as the memory space argument for scratch views. The amount of scratch memory necessary to hold a `View` can be queried through a static `View` member function. That function takes alignment requirements for the data into account. Users can mix and match both levels of scratch memory, as well as per-team and per-thread memory. Scratch `Views` are unmanaged. Their constructor takes a scratch handle argument obtained with the `scratch_space` member functions `team_scratch` and `thread_scratch`. The number of `Views` a user can create is not limited.

```
using scr_t = View<int*, typename ExecS::
    scratch_space>;
int per_team_bytes = scr_t::shmem_size(K);
int per_thr_bytes = scr_t::shmem_size(V);

// Create the policy and set the scratch sizes
TeamPolicy<ExecS> policy(N, AUTO, 8);
policy.set_scratch_size(0,PerTeam(per_team_bytes);
policy.set_scratch_size(1,PerThread(per_thr_bytes);

parallel_for("Label", policy,
  KOKKOS_LAMBDA(const team_t& team_h) {
  // Create a scratch View shared by all threads
  scr_t team_scr(team_h.team_scratch(0),K);
  // Create scratch views, with separate allocation
  // for each thread
  scr_t team_scr(team_h.thread_scratch(1),V);
  // ...
});
```

Actual memory allocations and deallocations, even through a memory pool, would be too costly within a parallel kernel. Instead, two functions `team_scratch` and `thread_scratch`, are used with a special `View` constructor. These functions strictly advance an internal pointer into the team's scratch allocation every time they are used to create a new scratch `View`. If a `View` creation would advance the scratch pointer beyond the preallocated size, the allocation - and with it the kernel execution - will fail. However, creating a scratch view requires only a few integer operations with this approach. The following code illustrates the failure due to advancing the scratch pointer too far. In the first iteration the scratch pointer already reached the end, in the second it will fail.

```
using scr_t = View<int*, typename ExecS::
    scratch_space>;
int per_team_bytes = scr_t::shmem_size(K);

// Create the policy and set the scratch sizes
TeamPolicy<ExecS> policy(N, AUTO, 8);
policy.set_scratch_size(0, PerTeam(per_team_bytes);

parallel_for(policy,
  KOKKOS_LAMBDA(const team_t& team_h) {
  for(int r=0; r<10; r++) {
    // This will fail in iteration r==1
    scr_t team_scr(team_h.team_scratch(0),K);
    // Do stuff
  }
});
```

TABLE 5
Bandwidth Comparison of a CG Solve of 100x100x100 Heat Conduction Problem Matrix

V100		SKX		A64FX	P9	MI100	
KK	CuSparse	KK	MKL	KK	KK	KK	rocSparse
965	835	161	152	145	148	805	758

For the TPL versions, the SPMV is executed via CuSparse and MKL respectively. The native Kokkos SPMV algorithm uses 3-level hierarchical parallelism. Numbers reported are in GB/s.

8.5 CGSolve Benchmark

An example demonstrating the versatility of Kokkos's hierarchical parallelism is a Sparse Matrix Vector Multiplication (SPMV) operation. SPMV is a critical component of sparse linear solvers. A commonly used Kokkos SPMV implementation assigns a workset of rows to each team and distributes the rows of the workset across the threads of that team. Vector-level parallelism is used to perform the nested reduction, which returns the dot product of a matrix row with the vector.

```
parallel_for(TeamPolicy{nworksets, team_size,
    vector_length},
 KOKKOS_LAMBDA(team_h const& team) {
  int first_row = team.league_rank()*rows_per_team;
  int last_row  = first_row + rows_per_team < nrows
                ? first_row + rows_per_team : nrows;
  parallel_for(TeamThreadRange(team,
               first_row, last_row), [&](int row) {
    int row_start = A.row_ptr(row);
    int row_length = A.row_ptr(row + 1) - row_start;
    double y_row;
    parallel_reduce(ThreadVectorRange(team,
              row_length), [=](int i, double& sum) {
      sum += A.values(i + row_start) *
             x(A.col_idx(i + row_start));
    }, y_row);
    y(row) = y_row;
  });
});
```

This kernel has three tuning parameters: `rows_per_team`, `team_size` and `vector_length`. These three tuning parameters allow the kernel to adapt to a range of different sparse matrices and hardware platforms. In Table 5 the performance of a Kokkos-based Conjugate Gradient Solver, as used in miniFE, is shown. One version uses the above SPMV kernel. The TPL versions differ only by calling an optimized vendor library for the SPMV kernel. For this particular matrix, the Kokkos SPMV kernel with optimized values for the three tuning parameters outperforms the vendor libraries, resulting in higher aggregate performance. This Kokkos SPMV algorithm might not perform as well for matrices with different sparsity structure. There is no expectation that this simple Kokkos algorithm generally outperforms the vendor libraries. It is worth noting that the aggregate achieved bandwidth can exceed the main memory bandwidth due to caching effects.

9 Execution Space Instances

While both `MDRangePolicy` and `TeamPolicy` are helping developers to expose more *fine grained* parallelism, sometimes the need arises to express parallelism on coarser levels. Often that parallelism is *functional* parallelism. In CUDA, HIP, OpenCL and SYCL this type of parallelism can be exploited by using multiple *streams* or *command queues* [18]. For Kokkos, the corresponding concept is called *Execution Space Instances*.

```
ExecutionSpace exec_1(...), exec_2(...);
// Two kernels running concurrently
parallel_for("Kernel1", Policy(exec_1, args...),
    Functor1);
parallel_for("Kernel2", Policy(exec_2, args...),
    Functor2);

// Wait for Kernel1
exec_1.fence();
// ...
// Wait for Kernel2
exec_2.fence();
```

When parallel patterns are dispatched in Kokkos, they are enqueued into the execution queue of an *Execution Space Instance*. Kokkos provides default instances via singletons. If no specific *Execution Space Instance* is passed to the *Execution Policy*, an operation is enqueued into the default instance of the Execution Space *type* the kernel is compiled for. *Execution Space Instances* have *reference* semantics; internal resources, such as scratch buffers, are not duplicated and are released after the last reference goes out of scope. To make sure work dispatched to an instance is finished, a user can call its `fence` member function. The global non-member function `fence` will block on *all* outstanding work on *all* active *Execution Space Instances*.

As with streams in CUDA and HIP, work dispatched to two distinct *Execution Space Instances* have parallel forward progress semantics; e.g., kernels submitted to two different instances may overlap in execution. This allows application developers to expose functional parallelism, such as computing long range and short range forces in parallel in Molecular Dynamics applications [6].

Note that the *Execution Space Instance* does not express any dependencies between kernels, other than those the user defines explicitly through the use of `fence`.

10 Kokkos Graphs

Once other factors like kernel implementation and data movement have been optimized, kernel latency is often a remaining bottleneck. Particularly, in scenarios where strong scaling of computational cost with respect to available resources is desired, latency costs can become dominant. With less work per parallel region, the latency associated with kernel submission—which includes costs like driver overhead and on-device work scheduling—becomes signifcant. Other factors, such as software complexity, modularity, and increased kernel diversity can also lead to decreased kernel size and thus an increase in the contribution of latency to the computational cost.

To estimate the potential scope of computational challenges around increasingly complex physics, consider a simple sparse conjugate gradient (CG) solver. It consists of three kernels: sparse matrix vector multiply (*spmv*), dot product (*dot*) and vector add (*axpby*). It is an iterative solver; iterative applications are an excellent example of a typical case where the need for strong scaling can be particularly acute.

```
function x = conjgrad(A, b, x)
    spmv(r, A, x)
    axpby(r, 1, b, -1, r);
    p = r;
    rsold = dot(r, r);

    while(true) do
        spmv(Ap, A, p);
        alpha = rsold / dot(p, Ap);
        axpby(x, 1, x, alpha, p);
        axpby(r, 1, r, -alpha, Ap);
        rsnew = dot(r,r);
        if sqrt(rsnew) < threshold
              break
        end
        axpy(p, 1, r, (rsnew / rsold), p)
        rsold = rsnew;
    end
end
```

Each iteration calls *axpby* three times, *dot* twice, and *spmv* a single time. Now consider the performance characteristics of this solver for a matrix derived from a regular 3D heat conduction problem, such as the one used for miniFE [13]. Since the problem in miniFE is derived from a regular grid, the matrix A contains 27 entries per row, with the number of rows in A and the vector length being equal to the number of grid cells. Using 204,800 grid cells, the operations have the following memory accesses requirements:

- AXPBY: 204,800 * 3 * sizeof(double) = 4,800kB
- DOT: 204,800 * 2 * sizeof(double) = 3,200kB
- SPMV: 204,800 * 1 * 27 * (sizeof(double) + sizeof(int)) = 64,800kB

Given the specifications of the latest NVIDIA A100 GPU, which provides 2TB/s memory bandwidth in the 80GB capacity version, these kernels should take 2.4μs, 1.6μs and 32.4μs respectively. Thus, excluding launch latencies, a single iteration of the solver should take less than 43μs at peak bandwidth. However, on X86 systems with NVIDIA GPUs, our experience has been that kernel launch latencies are about 3.5μs. With AMD GPUs, we have observed kernel launch latencies on the order of 8μs. With six kernel launches in each iteration of CG Solve, the kernel launch latency will add around 50 to 100 percent to the runtime.

To address latency issues in repeated sets of dependent kernels like the CG solver, CUDA introduced a feature called CUDA Graphs [18]. CUDA Graphs allow the lazy expression of a set of dependent kernels. While the creation of the graph has an upfront cost, the graph can then be dispatched repeatedly through a single operation. Even when dispatching a kernel with a CUDA graph, both dispatch and device latencies persist, however the overall average latency will be reduced, relative to launching each kernel individually.

Kokkos Graphs was designed in part to surface this functionality in a performance portable way. As with CUDA, an object is created lazily that represents a directed acyclic graph of kernels. Once constructed, the graph can be resubmitted any number of times with reduced latency per kernel. Kokkos Graphs provides an explicit creation phase, delimited by the scope of an immediately evaluated function that clearly indicates the portion of the program where the graph is modifiable but not executable. Graphs are created through a `create_graph` function with a function object argument, which is passed the root node of the graph. Graph nodes are then connected with a typical async-future model, using member functions like `then_parallel_for`, `then_parallel_reduce`, and `then_parallel_scan` which generate a new node with a dependency on the preceding node. Each of these calls has the same function signature as the normal, eager dispatch versions of Kokkos's parallel patterns. Another new API function is `when_all`, which creates a join point across multiple kernels that can serve as a dependency for subsequent graph kernels.

The following snippet uses Kokkos Graphs to express a diamond shaped dependency between kernels and execute it repeatedly:

```
auto graph = create_graph(exec, [=](auto root) {
 auto n1 = root.then_parallel_scan(N, ScanFunc);
 auto n2a = n1.then_parallel_for(M, Func2A);
 auto n2b = n1.then_parallel_for(L, Func2B);
 auto n2 = when_all(n2a, n2b);
 n2.then_parallel_reduce(K, ReduceFunc, result);
});
while(result() < threshold) {
  graph.submit(); fence();
}
```

11 SIMD Support

While Kokkos's parallel patterns are semantically vectorizable, and the `ThreadVectorRange`-based loops in *Hierarchical Parallelism* express additional parallelism to the compiler, there are often situations where Kokkos is unable to sufficiently convey the vectorizability of the work items to the compiler, and an explicit expression of SIMD vector semantics is required. One such scenario is algorithms where vectorization of the innermost loop is inefficient, and in the absence to this vectorization, results in utilization of scalar instructions by the compiler. Generally, this situation is referred to as outer loop vectorization.

```
parallel_for(N, KOKKOS_LAMBDA(int i) {
  // for small K would be better to vectorize over i
  for(int j = 0; j < K; j++) { a(i, j) += b(i, j); }
});
```

In these scenarios, one commonly used solution is to use explicit *SIMD* types. In [21], a more detailed explanation for the desired capabilities is given. Here, we introduce a capability incorporating those design ideas into the an API based on what is proposed for the C++ standard in the ISO C++ Parallelism TS v2.

The main distinguishing feature of the C++ standard proposal is the use of an *ABI* template parameter, which determines the actual implementation of a SIMD type. We utilize this template parameter to extend the interface to provide an associated *storage* ABI type for each *compute* ABI type. For CPUs the *storage* and the *compute* SIMD types are actually the same. But on GPU systems the *storage* needs to happen as dense arrays, while the *compute* types create only a single scalar variable per GPU warp lane in order to use warp level parallelism to do the SIMD operations.

The following code example illustrates this concept of using distinct *storage* and *compute* SIMD types:

```
using simd_abi = // ..., e.g. cuda_warp<32>, native,
     packed<16>
using simd_t = simd<double, simd_abi>;
using simd_storage_t = simd_t::storage_type;

auto data = View<simd_storage_t**>{"D", N, M};
auto results = View<simd_storage_t*>{"R", N};

parallel_for(p, TeamPolicy{N, 1, simd_t::size()},
  KOKKOS_LAMBDA(team_t const& team) {
    simd_t tmp = 0.0;
    double b = a;
    int i = team.league_rank();
    for(int j = 0; j < data.extent(1); j++) {
      tmp += b * simd_t(data(i, j));
      b += a + (j + 1);
    }
    results(i) = tmp;
  });
```

Besides the usual arithmetic and math functions provided for the SIMD types, the Kokkos implementation also provides mask types for masked operations.

12 Backends

Kokkos currently features the `Serial`, `OpenMP`, `Cuda`, `HIP`, `OpenMPTarget`, `HPX`, `Threads` and `SYCL` backends. These backends provide coverage for all major existing super computer architectures, including the announced exascale machines. As of writing the new `SYCL`, `HIP` and `OpenMPTarget` backends are almost feature complete, and are missing primarily the *Tasking* support, which currently is not widely used in applications. We expect that these backends will be optimized and fully ready for the upcoming ExaScale machines by the end of 2021.

Generally, the Kokkos Programming Model is easily mappable to any of the native models mentioned above. That said, not all of them can exploit the full semantics of Kokkos. For example, with the CPU `OpenMP` backend, every parallel operation we dispatch is synchronous, since *OpenMP* parallel regions are synchronous. Similarly `HIP` doesn't have the equivalent to *CUDA Graphs* yet, which means that *Kokkos Graphs* will simply dispatch kernels one by one, and will not provide the latency benefits seen with the `Cuda` backend. Our philosophy is to generally expose advanced capabilities of native programming models, and then providing fallback implementations for others. In particular, that means that Kokkos strives to find the common subset of models, and aims to expose powerful capabilities wherever possible.

13 Performance Portability in Practice

While a comprehensive study of practical performance portability in applications is out of scope for this paper, many such studies have been conducted in the past, and we will summarize some results here. These studies generally fall into one of three categories (i) comprehensive comparison of programming models with mini-apps, (ii) comparison of a real application ported to Kokkos with its non-Kokkos version, and (iii) comparison of Kokkos-based application on a range of hardware. Deakin *et al.* for example used the performance portability metric developed by Pennycook *et al.* [20] to compare OpenMP, Kokkos, CUDA, OpenACC and OpenCL [7]. Kokkos achieved the highest score with 68 percent while OpenMP got 28 percent. No overall scores for the other models were produced, since they didn't support enough systems. It is worthwhile to note that Kokkos achieves that higher performance portability score with less lines of code than OpenMP in the study. Grete *et al.* report on the astrophysics code K-Athena and compare it to the legacy non-Kokkos Athena++ as well as another similar code called GAMER [10]. GAMER supports CPUs as well as NVIDIA GPUs via CUDA. On CPUs K-Athena achieves 93 percent of the Athena++ performance and is 1.5 times faster than GAMER. K-Athena is also 1.3 times faster than GAMER on a GPU. Bertagna *et al.* report on performance results of porting the climate code HOMME to Kokkos, now called HOMMEXX [5]. They first demonstrate that in large scale runs the new Kokkos code is actually up to 60 percent faster on CPU systems, before comparing seven different node architectures. Scaling the maximum achieved performance of each node type by its theoretical peak bandwidth normalized to the Skylake result gives values of 0.56 to 1, roughly in line with the range observed by Deakin *et al.* Halver *et al.* implemented a long range coulomb solver and reported fraction of achieved peak performance [11]. With the exception of the Intel Haswell node, which achieved 40 percent, every other architecture fell into a narrow band of 28 to 33 percent. Results such as these demonstrate that Kokkos based code can achieve practical performance portability in real world settings and thus provides an excellent basis to prepare for the upcoming exascale era platforms.

Acknowledgments

The authors would like to thank the contributions of one particular person, H. Carter Edwards. He led the Kokkos project through its youth phase, and laid the foundation for what it has become. Without him, Kokkos wouldn't be what it is today. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Grant DE-NA-0003525. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Grant DE-AC02-06CH11357. This manuscript has been authored by UT-Battelle, LLC, under Grant DE-AC05-00OR22725 with the U.S. Department of Energy (DOE). This work was supported by Exascale Computing Project 17-SC-20-SC, a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

References

[1] V. Alessandrini, "Atomic types and operations," in *Shared Memory Application Program. Concepts and Strategies in Multicore Application Program.*, V. Alessandrini, ed. Waltham, MA, USA: Morgan Kaufmann, 2016, ch. 8, pp. 167–190.

[2] Y. Asahi, G. Latu, V. Grandgirard, and J. Bigot, "Performance portable implementation of a kinetic plasma simulation mini-app," in *Proc. Int. Workshop Accel. Prog. Using Directives*, 2019, pp. 117–139.

[3] B. Bastem and D. Unat, "Tiling-based programming model for structured grids on GPU clusters," in *Proc. Int. Conf. High Perform. Comput. Asia-Pac. Region*, 2020, pp. 43–51.

[4] D. A. Beckingsale *et al.*, "RAJA: Portable performance for large-scale scientific applications," in *Proc. IEEE/ACM Int. Workshop Perform., Portability Productiv. HPC*, 2019, pp. 71–81.
[5] L. Bertagna *et al.*, "HOMMEXX 1.0: A performance-portable atmospheric dynamical core for the energy exascale earth system model," *Geoscientific Model Develop.*, vol. 12, no. 4, pp. 1423–1441, 2019.
[6] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers-Particle-particle particle-mesh," *Comput. Phys. Commun.*, vol. 183, no. 3, pp. 449–459, 2012.
[7] T. Deakin *et al.*, "Performance portability across diverse computer architectures," in *Proc. IEEE/ACM Int. Workshop Perform., Portability Productiv. HPC*, 2019, pp. 1–13.
[8] V. Devadas and M. Curtis-Maury, "Scalable coordination of hierarchical parallelism," in *Proc. 49th Int. Conf. Parallel Process.*, 2020, pp. 1–11.
[9] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, pp. 3202–3216, 2014.
[10] P. Grete, F. W. Glines, and B. W. O'Shea, "K-Athena: A performance portable structured grid finite volume magnetohydrodynamics code," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 85–97, Jan. 2021.
[11] R. Halver, J. H. Meinke, and G. Sutmann, "Kokkos implementation of an Ewald Coulomb solver and analysis of performance portability," *J. Parallel Distrib. Comput.*, vol. 138, pp. 48–54, 2020.
[12] J. R. Hammond, M. Kinsner, and J. Brodman, "A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications," in *Proc. Int. Workshop OpenCL*, 2019, pp. 1–2.
[13] Mantevo, "minife," Accessed: Feb. 24, 2021. [Online]. Available: https://github.com/Mantevo/miniFE
[14] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the performance portability of modern parallel programming models using TeaLeaf," *Concurrency Comput.: Pract. Experience*, vol. 29, no. 15, 2017, Art. no. e4117.
[15] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Program. Patterns for Efficient Computation*, 1st ed. Waltham, MA, USA: Morgan Kaufmann Publishers Inc., 2012.
[16] D. S. Medina, A. St-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," 2014, *arXiv:1403.0968*.
[17] R. Menon and L. Dagum, "OpenMP: An industry-standard API for shared-memory programming," *Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan.–Mar. 1998.
[18] NVIDIA, "Cuda runtime API," Accessed: Feb. 23, 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html
[19] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, and H. Kasahara, "Hierarchical parallelism control for multigrain parallel processing," in *Proc. Int. Workshop Lang. Compilers Parallel Comput.*, 2005, pp. 31–44.
[20] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A metric for performance portability," in *Proc. Program. Models Benchmarking Simul. Workshop SC*, 2016, pp. 1–7.
[21] D. Sahasrabudhe, E. T. Phipps, S. Rajamanickam, and M. Berzins, "A portable simd primitive using kokkos for heterogeneous architectures," in *Proc. Int. Workshop Accel. Program. Using Directives*, 2020, pp. 140–163.
[22] C. Tanis, K. Sreenivas, J. C. Newman, and R. Webster, "Performance portability of a multiphysics finite element code," in *Proc. Aviation Technol., Integration, Operations Conf.*, 2018.
[23] J. L. Traff, "Hierarchical gather/scatter algorithms with graceful degradation," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004.
[24] P. P. M. T. and Services GmbH, "Top500," Accessed: Feb. 18, 2021. [Online]. Available: https://www.top500.org/

Christian R. Trott received the PhD degree in theoretical physics from TU Ilmenau, Germany. He is currently a principal member of Technical Staff and Sandia National Laboratories, where he has been working since acquiring his PhD degree. He leads the Kokkos Core Project and represents Sandia at the ISO C++ committee.

Damien Lebrun-Grandié is currently a computational scientist with Oak Ridge National Laboratory. He coleads the Kokkos Core Project and represents ORNL at the ISO C++ standards committee.

Daniel Arndt is currently a computational scientist with Oak Ridge National Laboratory, working on various ECP projects. He is also a mathematician by training specializing on finite element simulations. His research focuses on supporting new backends in Kokkos.

Jan Ciesko received the PhD degree in computer science from the Universitat Politècnica de Catalunya, Spain. He is currently a postdoctoral researcher with Sandia National Laboratories. His research interests include user-level threading in Open MPI and PGAS support in the Kokkos programming model.

Vinh Dang received the PhD degree in electrical engineering from the Catholic University of America in 2015. He is currently a senior member of Technical Staff with Sandia National Laboratories. His research interests include high-performance computing and parallel dense linear algebra or solvers.

Nathan Ellingwood received the PhD degree in applied mathematics and computational sciences from the University of Iowa. He is currently a senior member of Technical Staff with Sandia National Laboratories, where he contributes to the Kokkos Core and Kokkos Kernels projects, with a focus on testing and release infrastructure.

Rahulkumar Gayatri received the PhD degree in computer science from the Universitat Politècnica de Catalunya, Spain. He is currently an application performance specialist with NERSC, Lawrence Berkeley National Laboratory.

Evan Harvey received the BS degree in computer science from the New Mexico Institute of Mining and Technology in 2016. He is currently a limited term employee with Sandia National Laboratories. He contributes to the Kokkos Core and Kokkos Kernels Projects, with a focus on parallel dense linear algebra, software engineering, and continuous integration testing.

Daisy S. Hollman received the PhD degree in quantum chemistry from the University of Georgia. She is currently researching programming model design and programming languages with Sandia National Labs, where she has been working since acquiring her PhD degree. She represents Sandia on the ISO C++ standards committee, where she is involved in proposals ranging from multidimensional arrays to generic execution abstractions.

Dan Ibanez received the BS and PhD degrees in computer science from Rensselear Polytechnic Institute. Since 2016, he has been working with Sandia National Laboratories. His research interests include computational hydrodynamics, adaptive mesh algorithms, and contributions to Kokkos, such as ScatterView and SIMD vectorization.

Nevin Liber is currently a computer scientist with Argonne National Laboratory, working on the SYCL backend for Kokkos. He is also a C++ committee officer (Vice Chair, Library Evolution Working Group Incubator), and represents Argonne in SYCL related standardization efforts.

Jonathan Madsen is currently an application performance specialist with the National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory. He is a developer of the Kokkos Core project, leads the development of a modular toolkit for software monitoring at LBNL (timemory), and represents LBNL at the ISO C++ standards committee.

Jeff Miles received the PhD degree in mechanical engineering from North Carolina State University, after developing software at Toshiba in an industrial setting. He then joined Sandia National Laboratories, where he worked on Kokkos and helped application with its adoption.

David Poliakoff currently leads the Kokkos Tools effort. He has spent seven years with various DOE National Laboratories, working on tools in multiphysics applications.

Amy Powell is currently a senior member of the Technical Staff, Sandia National Laboratories. She is also a Kokkos and Kokkos Kernels developer, focusing on performance testing. She is trained as an evolutionary biologist and active in climate R&D.

Sivasankaran Rajamanickam received the PhD degree in computer science and engineering from the University of Florida in 2009. He is currently a principal member of Technical Staff, Sandia National Laboratories. His research interests include high-performance computing and sparse linear algebra or solvers.

Mikael Simberg received the master's degree in operations research and computer science from Aalto University, Finland. He is currently a scientific software developer with Swiss National Supercomputing Centre. His research interests include contributions to parallel programming models, in particular HPX.

Dan Sunderland is currently an expert in C++ and high-performance computing. He worked on programming models, including Kokkos and OpenMP at Sandia and helped represent the laboratory at the ISO C++ committee.

Bruno Turcksin received the PhD degree in nuclear engineering from Texas A&M University. He is currently a computational scientist with Oak Ridge National Laboratory. He worked on multiple ECP projects, including Kokkos where he focuses on the development of HIP backend.

Jeremiah Wilke received the PhD degree in computational chemistry from the University of Georgia. He is currently a scientist with Sandia National Laboratories. His research interests include high-performance computing, which include distributed computing and network simulations.