Clarifying and Specifying Concepts Used By Kokkos

Introduction

Kokkos has been using concept-driven design essentially since its inception. Until recently, there has been no readily-available mechanism for expressing these concepts in code. (In fact, there hasn't really even been a generally agreed-upon approach to *documenting* concepts.) With the merging of C++ Concepts as a language feature into the working draft, their implementation in multiple compilers, and the acceptance of the first concept-driven library feature, ranges, into the standard, it is time to consider formally specifying and documenting the concepts Kokkos uses in generic code. Beyond this, the recent rapid expansion in (and corresponding demand for) new Kokkos backends makes the lack of a hardened, formal specification for basic Kokkos concepts more acute than ever. With several new backend projects on the near horizon—SYCL, resilience, and HPX, just to name a few—now is the best time to formalize the core concepts in Kokkos so that we can begin to iteratively harden the concept set that Kokkos uses in generic code. Additionally, since the promotion to Kokkos 3.0 includes a plan to deprecate many non-general features of certain execution spaces, now is a good time to take a more holistic approach to the execution space concept and the concepts it interacts with.

Approach

In addition to the experience we have garnered over the past several years of participation in the ISO-C++ executor specification effort, we should use the approach in "Design of Concept Libraries for C++" (2011) by Sutton and Stroustrup as a guide. In particular, we should design based on the *concepts = constraints + axioms* philosophy, which focuses on balancing flexibility of use with ease of learning. This is not far from the design philosophy we already take—for instance, we don't really have separate "concepts" for the range policies given to `parallel_for` and `parallel_reduce`, even though a pure constraints-based approach may not see a need to make these two the same (the algorithms do very slightly different things with them, after all). By combining constraints sets that are "similar enough" (as we always have done in the past), we can maintain the flexibility we need while minimizing cognitive load

on users.

Overview

When it comes to cognitive load, perhaps even more important than limiting the total number of concepts is limiting the number of *subsumption hierarchies* of concepts. Experience with C++ ranges has also shown that limiting the branching width of these hierarchies increases ease of learning. Roughly speaking and from a high-level perspective, the major user-visible concept hierarchies that Kokkos currently uses are:

- `ExecutionSpace`
- `MemorySpace`
- `ExecutionPolicy` (includes, for instance, `RangePolicy`)
- `TeamMember`
- `Functor`

Some minor hierarchies include:

- `MemoryLayout`
- `Reducer`
- `TaskScheduler`

Some things currently being treated as concepts (according to `Kokkos_Concepts.hpp`) that are really more like enumerations (or something similar) are:

- `MemoryTraits`
- Several things used only with execution policies:
    - `Schedule`
    - The `IndexType<>` tag
    - The `LaunchBounds<>` tag
    - `IterationPattern` (a.k.a. `Kokkos::Iterate`)

There is also some question as to whether `Kokkos::View` (and friends) should be presented as a concept rather than just a class template, given the existence of act-alike class templates such as `DualView` and `OffsetView` external to Kokkos.

The `ExecutionSpace` Concept

Working off the functionality currently common to `Serial`, `Cuda`, `OpenMP`, `Threads`, `ROCm`, and `OpenMPTarget`, the current state of the Kokkos `ExeuctionSpace` concept looks something like:

```
template <typename Ex>
concept ExecutionSpace =
  CopyConstructible<Ex> &&
  DefaultConstructible<Ex> &&
  Destructible<Ex> &&
  // Member type requirements:
  requires() {
    typename Ex::execution_space;
    std::is_same_v<Ex, typename Ex::execution_space>;
    typename Ex::memory_space;
    is_memory_space<typename Ex::memory_space>::value;
    typename Ex::size_type;
    std::is_integral_v<typename Ex::size_type>;
    typename Ex::array_layout;
    is_array_layout<typename Ex::array_layout>::value;
    typename Ex::scratch_memory_space;
    is_memory_space<typename Ex::scratch_memory_space>::value;
    typename Ex::device_type;
  } &&
  // Required methods:
  requires(Ex ex, std::ostream& ostr, bool detail) {
    { ex.in_parallel() } -> bool;
    { ex.fence() };
    { ex.name() } -> const char*;
    { ex.print_configuration(ostr) };
    { ex.print_configuration(ostr, detail) };
  } &&
```

Where we've extrapolated from the recent progress on execution space instances that many methods currently implemented as static methods eventually need only be instance methods in the general case.

Implementation Requirements

Further requirements cannot be expressed without additional types constrained by additional concepts (this is a well-known limitation of the concepts mechanism in C++, and is necessary

to preserve decidability of the type system). Though some argue for using an archetype pattern to get around this (whereby an archetype with an implementation-private name designed to meet the requirements of the extra concept is used in the definition of constraints), the state of practice appears to be converging on a strategy that involves creating an additional named concept templated on all relevant types and constraining them together, which can then be used at relevant call site. Most argue that this is a necessary artifact of the language feature, but that constraining concepts together in this way does not count as an "extra" concept for the purposes of cognitive load assessment. Applying this approach and assuming the intention is for things like `Kokkos::parallel_for` to remain as algorithms rather than customization points, we get some further requirements from the `Kokkos::Impl` namespace:

```
template <typename Ex, typename ExPol, typename F, typename ResultType = int>
concept ExecutionSpaceOf =
  ExecutionSpace<Ex> &&
  ExecutionPolicyOf<ExPol, Ex> && // defined below
  Functor<F> && // defined below
  // Requirements imposed by Kokkos_Parallel.hpp
  requires(Ex ex, ExPol const& policy, F f, ResultType& total) {
    // This is technically not exactly correct, since an rvalue reference qualified
    // execute() method would meet these requirements and wouldn't work with Kokkos,
    // but for brevity:
    { Impl::ParallelFor<F, ExPol, Ex>(f, policy).execute(); }
    { Impl::ParallelScan<F, ExPol, Ex>(f, policy).execute(); }
    { Impl::ParallelScanWithTotal<F, ExPol, Ex>(f, policy, total).execute(); }
  }

template <typename Ex, typename ExPol, typename F, typename Red>
concept ExecutionSpaceOfReduction =
  ExecutionSpaceOf<Ex, ExPol, F> &&
  Reducer<Red> &&
  // Requirements imposed by Kokkos_Parallel_Reduce.hpp
  requires(
    Ex ex, ExPol const& policy, F f, Red red,
    Impl::ParallelReduce<F, ExPol, Red, Ex>& closure
  ) {
    { Impl::ParallelReduce<F, ExPol, Red, Ex>(f, policy); }
    { closure.execute(); }
  }
```

Perhaps, though, these should be part of some internal concepts (`ImplExecutionSpaceOf` , for instance) and the user-visible concept should exclude these requirements.

Support for `UniqueToken` adds the following requirements:

```
template <typename Ex>
concept UniqueTokenExecutionSpace =
  requires(
    Experimental::UniqueToken<Ex, Experimental::UniqueTokenScope::Instance> const& itok,
    Experimental::UniqueToken<Ex, Experimental::UniqueTokenScope::Global> const& gtok,
    typename Ex::size_type size
  ) {
    typename Experimental::UniqueToken<Ex, Experimental::UniqueTokenScope::Global>::size_t
    std::is_same_v<Ex, typename Experimental::UniqueToken<Ex, Experimental::UniqueTokenSco
    { itok.size() } -> typename Ex::size_type;
    { gtok.size() } -> typename Ex::size_type;
    { itok.acquire() } -> typename Ex::size_type;
    { gtok.acquire() } -> typename Ex::size_type;
    { itok.release(size) };
    { gtok.release(size) };
  }
  && CopyConstructible<Experimental::UniqueToken<Ex, Experimental::UniqueTokenScope::Insta
  && DefaultConstructible<Experimental::UniqueToken<Ex, Experimental::UniqueTokenScope::In
  && CopyConstructible<Experimental::UniqueToken<Ex, Experimental::UniqueTokenScope::Globa
  && DefaultConstructible<Experimental::UniqueToken<Ex, Experimental::UniqueTokenScope::Gl
```

An Additional Concept for `DeviceExecutionSpace`?

All of the device execution spaces, in their current state, have two extra member functions, `sleep()` and `wake()`. It's unclear whether this is intended to be general, but if it is, there is an additional concept in the hierarchy:

```
template <typename Ex>
concept DeviceExecutionSpace =
  ExecutionSpace<Ex> &&
  requires(Ex ex) {
    { ex.sleep() };
    { ex.wake() };
  }
```

Some *de facto* Requirements

There are other places where we're providing partial specializations using concrete execution spaces, such as `Impl::TeamPolicyInternal`. These also qualify as "requirements" on an `ExecutionSpace`, just like `Impl::ParallelFor<...>`. In many of these cases, it would be

nice if we could refactor some things to use a less “all-or-nothing” approach to customization than partial class template specialization.

Design Thoughts

The first thing that comes to mind is that `CopyConstructible<T> && DefaultConstructible<T> && Destructible<T>` is very close to `SemiRegular<T>`; all we need to do is add `operator==()`.

TODO more here

The `MemorySpace` Concept

Looking at the common functionality in the current implementations of `CudaSpace`, `CudaUVMSpace`, `HostSpace`, `OpenMPTargetSpace`, and `HBWSpace`, the current concept for `MemorySpace` looks something like:

```
template <typename Mem>
concept MemorySpace =
  CopyConstructible<Mem> &&
  DefaultConstructible<Mem> &&
  Destructible<Mem> &&
  // Member type requirements:
  requires() {
    std::is_same_v<Mem, typename Mem::memory_space>;
    Kokkos::is_execution_space<typename Mem::execution_space>::value;
    typename Mem::device_type;
  }
  // Required methods:
  requires(Mem m, size_t size, void* ptr) {
    { m.name() } -> const char*;
    { m.allocate(size) } -> void*;
    { m.deallocate(ptr, size) };
  };
```

Implementation Requirements

Most of the ways that the `MemorySpace` concept is used in generic contexts by Kokkos are in the `Impl` namespace.

```
template <typename Mem>
concept ImplMemorySpace =
  MemorySpace<Mem> &&
  DefaultConstructible<Impl::SharedAllocationRecord<Mem, void>> &&
  Destructible<Impl::SharedAllocationRecord<Mem, void>>
  requires(
    Mem mem, std::string label, size_t size,
    void* ptr, std::ostream& ostr, bool detail,
    Impl::SharedAllocationRecord<Mem, void> record,
    void (*dealloc)(Impl::SharedAllocationRecord<void, void>*)
  ) {
    { Impl::SharedAllocationRecord<Mem, void>(mem, label, size) };
    { Impl::SharedAllocationRecord<Mem, void>(mem, label, size, dealloc) };
    { record.get_label() } -> std::string;
    { Impl::SharedAllocationRecord<Mem, void>::allocate_tracked(mem, label, size) }
      -> void*;
    { Impl::SharedAllocationRecord<Mem, void>::reallocate_tracked(ptr, size) }
      -> void*;
    { Impl::SharedAllocationRecord<Mem, void>::deallocate_tracked(ptr) };
    { Impl::SharedAllocationRecord<Mem, void>::print_records(ostr, mem) };
    { Impl::SharedAllocationRecord<Mem, void>::print_records(ostr, mem, detail) };
    { Impl::SharedAllocationRecord<Mem, void>::get_record(ptr) }
      -> Impl::SharedAllocationRecord<Mem, void>*
  };

template <typename Mem1, typename Mem2, typename Ex>
concept ImplRelatableMemorySpaces =
  ImplMemorySpace<Mem1> &&
  ImplMemorySpace<Mem2> &&
  ExecutionSpace<Ex> &&
  requires(const void* ptr) {
    { Impl::MemorySpaceAccess<Mem1, Mem2>::assignable } -> bool;
    { Impl::MemorySpaceAccess<Mem1, Mem2>::accessible } -> bool;
    { Impl::MemorySpaceAccess<Mem1, Mem2>::deepcopy } -> bool;
    { Impl::VerifyExecutionCanAccessMemorySpace<Mem1, Mem2>::value } -> bool;
    { Impl::VerifyExecutionCanAccessMemorySpace<Mem1, Mem2>::verify() };
    { Impl::VerifyExecutionCanAccessMemorySpace<Mem1, Mem2>::verify(ptr) };
  } &&
  requires(Ex ex, void* dst, const void* src, size_t n) {
    { Impl::DeepCopy<Mem1, Mem2, Ex>(dst, src, n) };
    { Impl::DeepCopy<Mem1, Mem2, Ex>(exec, dst, src, n) };
  }
```

The `ExecutionPolicy` Concept

This is where I think we have the most work to do. We could achieve a significant complexity reduction by unifying disparate interfaces for, e.g., `RangePolicy<...>` and `ThreadVectorRange<...>`, into one hierarchy.

Looking at the current implementations of `RangePolicy<...>`, `MDRangePolicy<...>`, `TeamPolicy`, `Impl::TeamThreadRangeBoundariesStruct`, and `Impl::TeamVectorRangeBoundariesStruct`, all that I can find in common is:

```
template <typename ExPol>
concept BasicExecutionPolicy =
  CopyConstructible<ExPol> &&
  Destructible<ExPol> &&
  requires(ExPol ex) {
    ExPol::index_type;
  }
```

That is, of course, not a useful concept. If we exclude `Impl::TeamThreadRangeBoundariesStruct` and `Impl::TeamVectorRangeBoundariesStruct`, we get the tag also:

```
template <typename ExPol>
concept ExecutionPolicy =
  BasicExecutionPolicy<ExPol> &&
  requires(ExPol ex) {
    std::is_same_v<ExPol, typename ExPol::execution_policy>;
  }
```

which indicates that the policies that can be given to parallel algorithms inside of other algorithms weren't intended to be part of the same concept as the others (though I would argue maybe they should). `TeamPolicy` and `RangePolicy` both have functions for managing chunk sizes:

```
template <typename ExPol>
concept ChunkedExecutionPolicy =
  ExecutionPolicy<ExPol> &&
  requires(ExPol ex, typename ExPol::index_type size) {
    { ex.chunk_size() } -> typename ExPol::index_type;
    { ex.set_chunk_size(size) } -> ExPol&
  }
```

Chunk size is, of course, a bit more complicated with `MDRangePolicy`, but the generalization to chunks in each dimension is pretty straightforward, so we could unify concepts a bit here. The `IterateTile` abstraction is pretty nice, and seems like it could unify these concepts to reduce the amount of duplicate code in places like `impl/Kokkos_OpenMP_Parallel.hpp`.

It would be nice if there were some way to reduce the conceptual surface area by allowing users to think of a `RangePolicy` as a special case of `MDRangePolicy` with rank of 1, and to allow users to think of `RangePolicy` as a special case of `TeamPolicy` with `N` teams of size 1 each. Of course, we'd still provide the current interface as a shortcut, and would probably teach it the current way, but when users advance to the point where they're using all of these, it would be nice to have them think about one thing with two different axes rather than three different things.

Finally, it's not entirely clear to me why we need separate concepts for `TeamThreadRange` and `ThreadVectorRange`. In my mind, multiple levels of nested parallelism is just another axis along which to extend the execution policy concept, and it's not clear to me why we need to use up extra conceptual overhead to describe specific points in that hierarchy. (Again, I don't have any objections to the names specifically, just the extra cognitive load.)

It's entirely possible that there isn't significant simplification to be made here. Maybe the current separation of concerns is the simplest possible. But as long as we're looking at hardening Kokkos concepts, we should at least explore this space.

The `TeamMember` Concept

TODO

The `Functor` Concept

TODO

A Note on Implementation Delegation

TODO write this