The Kokkos Lectures

Module 5: SIMD, Streams and Tasking

August 14, 2020

Sandia National Laboratories is a multi-mission 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 contract DE-NA0003525.
SAND2020-8508 PE

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series:
  - Slides, recording and Q&A for the Lectures
- https://github.com/kokkos/kokkos/wiki:
  - Wiki including API reference
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

- 07/17 Module 1: Introduction, Building and Parallel Dispatch
- 07/24 Module 2: Views and Spaces
- 07/31 Module 3: Data Structures + MultiDimensional Loops
- 08/07 Module 4: Hierarchical Parallelism
- **08/14 Module 5: SIMD, Streams and Tasking**
- 08/21 Module 6: Internode: MPI and PGAS
- 08/28 Module 7: Tools: Profiling, Tuning and Debugging
- 09/04 Module 8: Kernels: Sparse and Dense Linear Algebra
- 09/11 Reserve Day

Hierarchal Parallelism

- **Hierarchical work** can be parallelized via hierarchical parallelism.
- Hierarchical parallelism is leveraged using **thread teams** launched with a `TeamPolicy`.
- Team "worksets" are processed by a team in nested `parallel_for` (or `reduce` or `scan`) calls with a `TeamThreadRange` and `ThreadVectorRange` policy.
- Execution can be restricted to a subset of the team with the `single` pattern using either a `PerTeam` or `PerThread` policy.
- Teams can be used to **reduce contention** for global resources even in "flat" algorithms.

Scratch Space

- **Scratch Memory** can be use with the `TeamPolicy` to provide thread or team **private** memory.
- Usecase: per work-item temporary storage or manual caching.
- Scratch memory exposes on-chip user managed caches (e.g. on NVIDIA GPUs)
- The size must be determined before launching a kernel.
- Two levels are available: large/slow and small/fast.

Unique Token

- **UniqueToken** give a thread safe portable way to divide thread specific resources
- **UniqueToken** can be sized to restrict ids to a range.
- A **Global** UniqueToken is available.

SIMD

How to vectorize code with explicit vector types.

Blocking behavior and Execution Space Instances

What is Kokkos's blocking behavior and Execution Space Instances

Tasking

Writing dynamic task graphs.

SIMD

Portable vector intrinsic types.

Learning objectives:

- How to use SIMD types to improve vectorization.
- SIMD Types as an alternative to ThreadVector loops.
- SIMD Types to achieve outer loop vectorization.

So far there were two options for achieving vectorization:

- **Hope For The Best**: Kokkos semantics make loops inherently vectorizable, sometimes the compiler figures it even out.
- **Hierarchical Parallelism**: `TeamVectorRange` and `ThreadVectorRange` help the compiler with hints such as `#pragma ivdep` or `#pragma omp simd`.

These strategies do run into limits though:

- Compilers often do not vectorize loops on their own.
- An optimal vectorization strategy would require *outer-loop vectorization*.
- Vectorization with `TeamVectorRange` sometimes requires artifically introducing an additional loop level.

A simple scenario where for outer-loop vectorization:

```
for(int i=0; i<N; i++) {
  // expect K to be small odd 1,3,5,7 for physics reasons
  for(int k=0; k<K; k++) b(i) += a(i,k);
}
```

Vectorization the K-Loop is not profitable:

- It is a short reduction.
- Remainders will eat up much time.

Using ThreadVectorRange is cumbersome and requires split of N-Loop:

```
parallel_for("VectorLoop",TeamPolicy<>(0,N/V,V),
  KOKKOS_LAMBDA ( const team_t& team ) {
  int i = team.league_rank() * V;
  for(int k=0; k<K; k++)
    parallel_for(ThreadVectorRange(team,V), [&](int ii) {
      b(i+ii) += a(i+ii,k);
    });
});
```

To help with this situation and (in particular in the past) fix the lack of auto-vectorizing compilers `SIMD-Types` have been invented. They:

- Are short vectors of scalars.
- Have operators such as += so one can use them like scalars.
- Are compile time sized.
- Usually map directly to hardware vector instructions.

Important concept: SIMD Type

A SIMD variable is a **short vector** which acts like a scalar.

Using such a `simd` type one can simply achieve *outer-loop* vectorization by using arrays of `simd` and dividing the loop range by its *size*.

Lets take a look back at the outer loop vectorization:

```
View<double*> b = ...
View<double**> a = ...
for(int i=0; i<N; i++) {
  // expect K to be small odd 1,3,5,7 for physics reasons
  for(int k=0; k<K; k++) b(i) += a(i,k);
}
```

Lets take a look back at the outer loop vectorization:

```
View<double*> b = ...
View<double**> a = ...
for(int i=0; i<N; i++) {
  // expect K to be small odd 1,3,5,7 for physics reasons
  for(int k=0; k<K; k++) b(i) += a(i,k);
}
```

Using SIMD types is conceptionally as simple as:

- Replace scalar type with SIMD type
- Adjust loop iteration count by SIMD length

```
View<SIMD<double>*> b = ...
View<SIMD<double>**> a = ...
int V = SIMD<double>::size();
for(int i=0; i<N/V; i++) {
  // expect K to be small odd 1,3,5,7 for physics reasons
  for(int k=0; k<K; k++) b(i) += a(i,k);
}
```

The ISO C++ standard has a *Technical Specification* for `simd` (in *parallelism v2*):

```
template< class T, class Abi >
class simd {
public:
  using value_type = T;
  using reference = /* impl defined */;
  using abi_type = Abi;
  static constexpr size_t size();
  void copy_from(T const*, aligned_tag);
  void copy_to(T*, aligned_tag) const;
  T& operator[] (size_t);
  //Element wise operators
};

// Element Wise non-member operators
```

One interesting innovation here is the `Abi` parameter allowing for different, hardware specific, implementations.

The most important in the proposal are:

- **scalar**: single element type.
- **fixed_size**$< N >$: stores `N` elements.
- **max_fixed_size**$< T >$: stores maximum number of elements for `T`.
- **native**: best fit for hardware.

One interesting innovation here is the `Abi` parameter allowing for different, hardware specific, implementations.

The most important in the proposal are:

- **scalar**: single element type.
- **fixed_size**$< N >$: stores `N` elements.
- **max_fixed_size**$< T >$: stores maximum number of elements for `T`.
- **native**: best fit for hardware.

But `std::experimental::simd` is not in the standard yet, and doesn't support GPUs ...

One interesting innovation here is the `Abi` parameter allowing for different, hardware specific, implementations.

The most important in the proposal are:

- **scalar**: single element type.
- **fixed_size**$< N >$: stores `N` elements.
- **max_fixed_size**$< T >$: stores maximum number of elements for `T`.
- **native**: best fit for hardware.

But `std::experimental::simd` is not in the standard yet, and doesn't support GPUs ...

It also has other problems making it insufficient for our codes ...

Just at Sandia we had at least **5** different SIMD types in use.

A unification effort was started with the goal of:

- Match the proposed `std::simd` API as far as possible.
- Support GPUs.
- Can be used stand-alone or in conjunction with Kokkos.
- Replaces all current implementations at Sandia for SIMD.

We now have an implementation developed by Dan Ibanez, which is close to meeting all of those criteria:

- For now available at https://github.com/kokkos/simd-math.
- Considered Experimental, but supports X86, ARM, Power, NVIDIA GPUs.
- Will be integrated into Kokkos soon.

As with the C++23 SIMD type it takes Scalar argument and ABI

```
template<class Scalar, class ABI>
simd;
```

Most common ABI types are:

- `simd::simd_abi::native`: whatever is best for the scalar type on the architecture
- `simd::simd_abi::scalar`: a single element
- `simd::simd_abi::pack<N>`: N scalars

Aliasing with Scalar Views is possible (but careful with Layout and length!)

```
using simd_t = simd::simd<double,simd::simd_abi::native>;
View<simd_t*> a("A",N);
View<double*> a_s(static_cast<double*>(a.data()),N*simd_t::size());

View<double*> b("B",M);
View<simd_t*> b_v(static_cast<simd_t*>(b.data()),M/simd_t::size());
```

Details:

- Location: `Exercises/simd/Begin/`
- Include the `simd.hpp` header.
- Change the data type of the views to use `simd::simd<double,simd::simd_abi:native>`.
- Create an unmanaged `View<double*>` of `results` using the `data()` function for the final reduction.

```
# Compile for CPU
make -j KOKKOS_DEVICES=OpenMP
# Compile for GPU
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./simd.cuda
```

Things to try:

- Vary problem size (-N ...; -K ...)
- Compare behavior of scalar vs vectorized on CPU and GPU

The above exercise used a **scalar** simd type on the **GPU**.
Why wouldn't we use a fixed_size instead?

- Using a `fixed_size` ABI will create a scalar of size `N` in each CUDA thread!
- Loading a `fixed_size` variable from memory would result in uncoalesced access.
- If you have correct layouts you get `outer-loop` vectorization implicitly on GPUs.

The above exercise used a **scalar** simd type on the **GPU**.
Why wouldn't we use a fixed_size instead?

- Using a `fixed_size` ABI will create a scalar of size `N` in each CUDA thread!
- Loading a `fixed_size` variable from memory would result in uncoalesced access.
- If you have correct layouts you get `outer-loop` vectorization implicitly on GPUs.

But what if you really want to use **warp**-level parallelziation for SIMD types?

The above exercise used a **scalar** simd type on the **GPU**.
Why wouldn't we use a fixed_size instead?

- Using a `fixed_size` ABI will create a scalar of size `N` in each CUDA thread!
- Loading a `fixed_size` variable from memory would result in uncoalesced access.
- If you have correct layouts you get `outer-loop` vectorization implicitly on GPUs.

But what if you really want to use **warp**-level parallelziation for SIMD types?
We need *two* SIMD types: a *storage* type and a *temporary* type!

Important concept: simd::storage_type

Every `simd<T,ABI>` has an associated `storage_type` typedef.

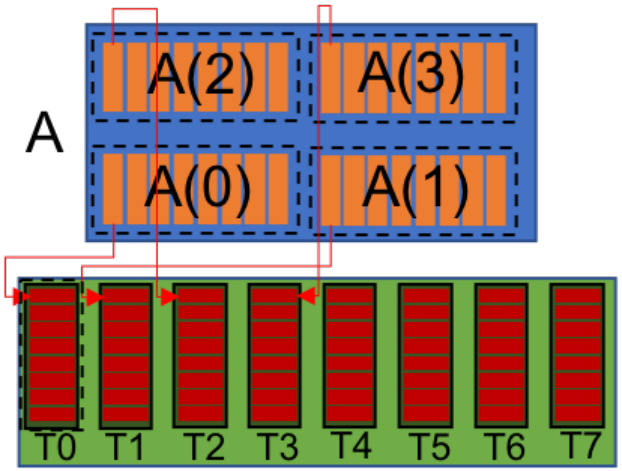
To help with the GPU issue we split types between **storage** types used for `Views`, and **temporary** variables.

- Most `simd::simd` types will just have the same `storage_type`.
- `simd<T,cuda_warp<N>>` will use warp level parallelism.
- `simd<T,cuda_warp<N>>::storage_type` is different though!
- Used in conjunction with `TeamPolicy`.

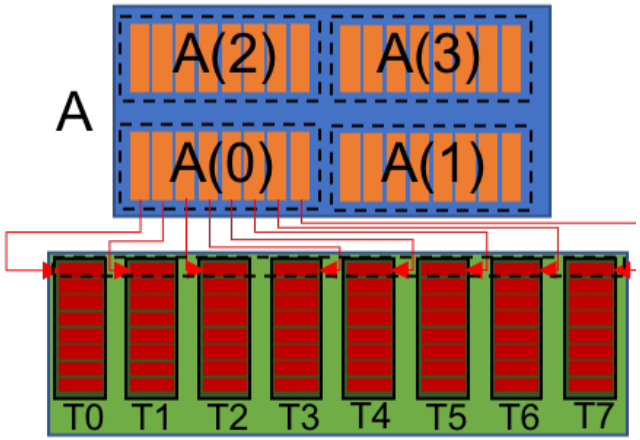
Illustrating difference between `pack` and `cuda_warp`

```
using ABI = ... ;
View<simd<double,ABI::storage_type> A(...);
parallel_for(TeamPolicy<>(N,AUTO,V),
 KOKKOS_LAMBDA(const teamt_t& team) {
  int i = team.league_rank()*team.team_size()+team.team_rank();
  simd<double,ABI> tmp = A(i);
});
```

```
using ABI = pack<8>; int V=1;
```




```
using ABI = cuda_warp<8>; int V=8;
```



Example of using storage_type:

```
// Using cuda_warp abi
using simd_t = simd::simd<T,simd::simd_abi::cuda_warp<V> >;
// Define simd_storage type
using simd_storage_t = simd_t::storage_type;
// Allocate memory
View<simd_storage_t**> data("D",N,M); // will hold N*M*V Ts

// Launch Loop with vectorlength V
parallel_for("Loop", TeamPolicy<>(N,M,V),
  KOKKOS_LAMBDA(const team_t& team) {
    int i = team.league_rank();
    parallel_for(TeamThreadRange(team,M), [&](int j) {
      // Load storage type into internal type;
      simd_t tmp = data(i,j);
      // Do something with it
      tmp *= 2.0;
      // write values back
      data(i,j) = tmp;
      // or inline:
      // data(i,j) = 2.0*simd_t(data(i,j));
  });
});
```

Details:

- Location: `Exercises/simd_warp/Begin/`
- Include the `simd.hpp` header.
- Change the data type of the views to use `simd::simd<double,simd::simd_abi:cuda_warp< 32 >>::storage_type`.
- Create an unmanaged `View<double*>` of `results` using the `data()` function for the final reduction.
- Use inside of the lambda the `simd::simd<double,simd::simd_abi:cuda_warp< 32 >>` as scalar type.

```
# Compile for GPU
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./simd.cuda
```

Kokkos SIMD supports math operations:

- Common stuff like `abs`, `sqrt`, `exp`, ...

It also supports masking:

```
// Scalar code with condition:
for(int i=0; i<N; i++) {
  if( a(i) < 100.0 ) b(i) = a(i);
  else b(i) = 100.0;
}

// Becomes
using simd_t = simd<double,simd_abi::native>;
using simd_mask_t = simd_t::mask_type;

for(int i=0; i<N/V; i++) {
  simd_t threshold(100.0), a_i(a_v(i));
  simd_mask_t is_smaller = threshold<a_i;

  b_v(i) = choose(is_smaller,a_i,threshold);
}
```

- SIMD types help vectorize code.
- In particular for **outer-loop** vectorization.
- There are **storage** and **temporary** types.
- Masking is supported too.
- Currently considered experimental at https://github.com/Kokkos/simd-math: please try it out and provide feedback.
- Will move into Kokkos proper likely in the next release.

Asynchronicity and Streams

The (non-)blocking behavior of Kokkos operations.

Learning objectives:

- What are blocking and non-blocking operations in Kokkos.
- What kind of work can overlap.
- How to wait for completion.
- How to run kernels simultaneously on a GPU.

Most operations in Kokkos are non-blocking

- The caller returns before the operation is finished
- The caller can do other things, while operations are executing

Most operations in Kokkos are non-blocking

- The caller returns before the operation is finished
- The caller can do other things, while operations are executing

So what is the ordering behavior?

- Execution Spaces have an ordered execution queue
- The queue is first-in/first-out (FIFO)

Most operations in Kokkos are non-blocking

- The caller returns before the operation is finished
- The caller can do other things, while operations are executing

So what is the ordering behavior?

- Execution Spaces have an ordered execution queue
- The queue is first-in/first-out (FIFO)

Important Point

Execution Spaces execute operations in dispatch order.

Execution Space Instances

- Each unique **Instance** of an execution space has its own queue
- Execution Policies can take an instance as the first argument
- deep_copy can take an instance as a first argument
- For every **Execution Space Type** there is a **default instance**
  - It is a singleton
  - The **default instance** is returned by the default constructor
  - Used if no specific instance is provided

Execution Space Instances

- Each unique **Instance** of an execution space has its own queue
- Execution Policies can take an instance as the first argument
- deep_copy can take an instance as a first argument
- For every **Execution Space Type** there is a **default instance**
  - It is a singleton
  - The **default instance** is returned by the default constructor
  - Used if no specific instance is provided

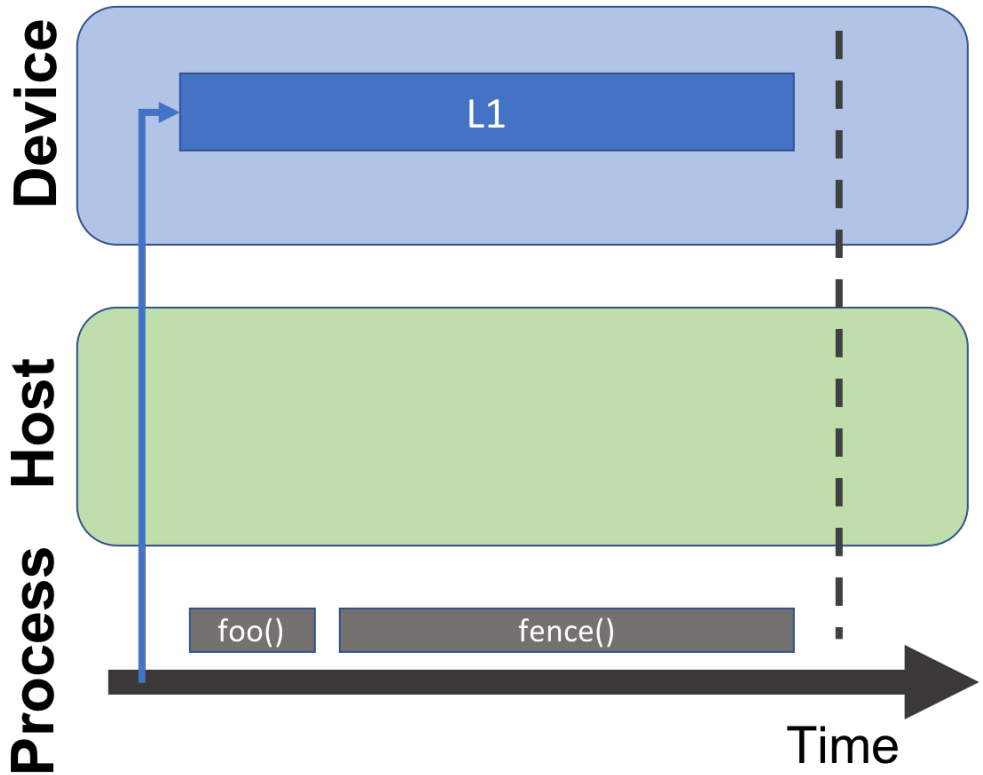
```
// This is equivalent:
RangePolicy<ExecSpace>
  policy_1(0, N);
RangePolicy<ExecSpace>
  policy_2(ExecSpace(), 0, N);
```

We use the following conventions in subsequent slides

```
// Execution Space types
using device = Kokkos::DefaultExecutionSpace;
using host = Kokkos::DefaultHostExecutionSpace;

// Execution Space instances
device dev1(..), dev2(..)
device host1(..), host2(..);

// Execution Policies
RangePolicy<device> policy_d(0,N), policy_device(0,N);
RangePolicy<host> policy_h(0,K), policy_host(0,K);
RangePolicy<device> policy_d1(dev1, 0,N), policy_d2(dev2, 0,N);
RangePolicy<host> policy_h1(host1, 0,N), policy_h2(host2, 0,N);

// Functors/Lambda for parallel_for
auto L1 = KOKKOS_LAMBDA(int i) {...};
auto L2 = ...; auto L3 = ...; auto L4 = ...; auto L5 = ...;
// Functors/Lambda for parallel_reduce
auto R1 = KOKKOS_LAMBDA(int i, double& lsum) {...};
```

Most Kokkos Operations are Asynchronous

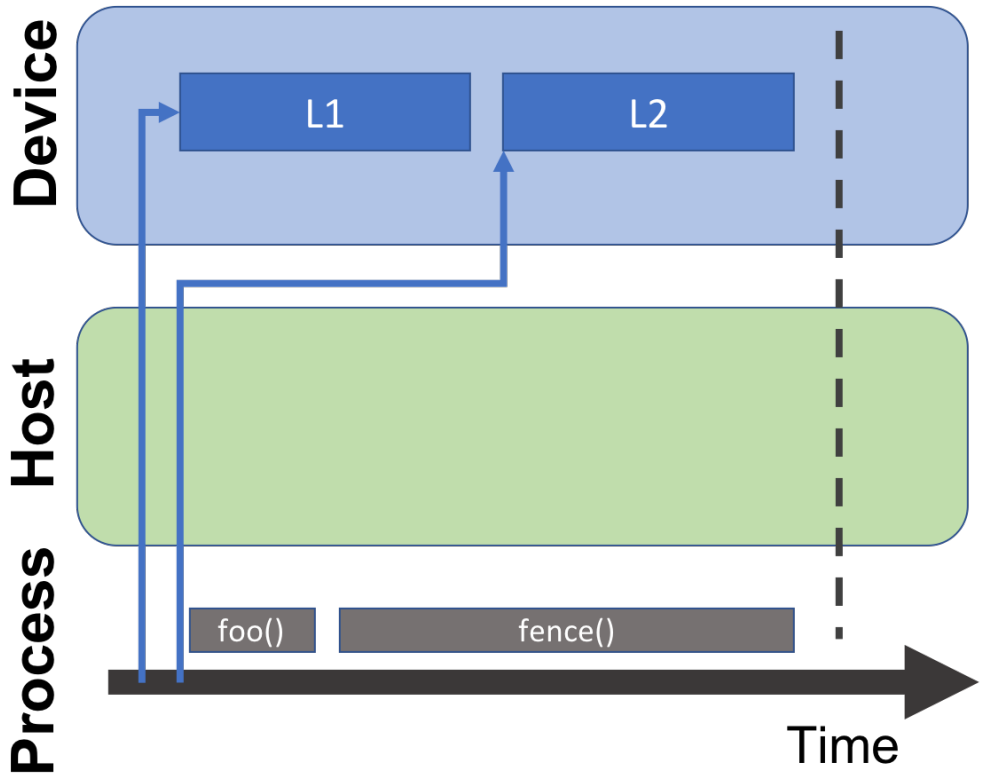
- Best to assume all of them are asynchronous
- They overlap with work in the process thread
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);

parallel_for("L1",
  policy_device, L1);
foo();
fence();
```

Execution Spaces execute in dispatch order

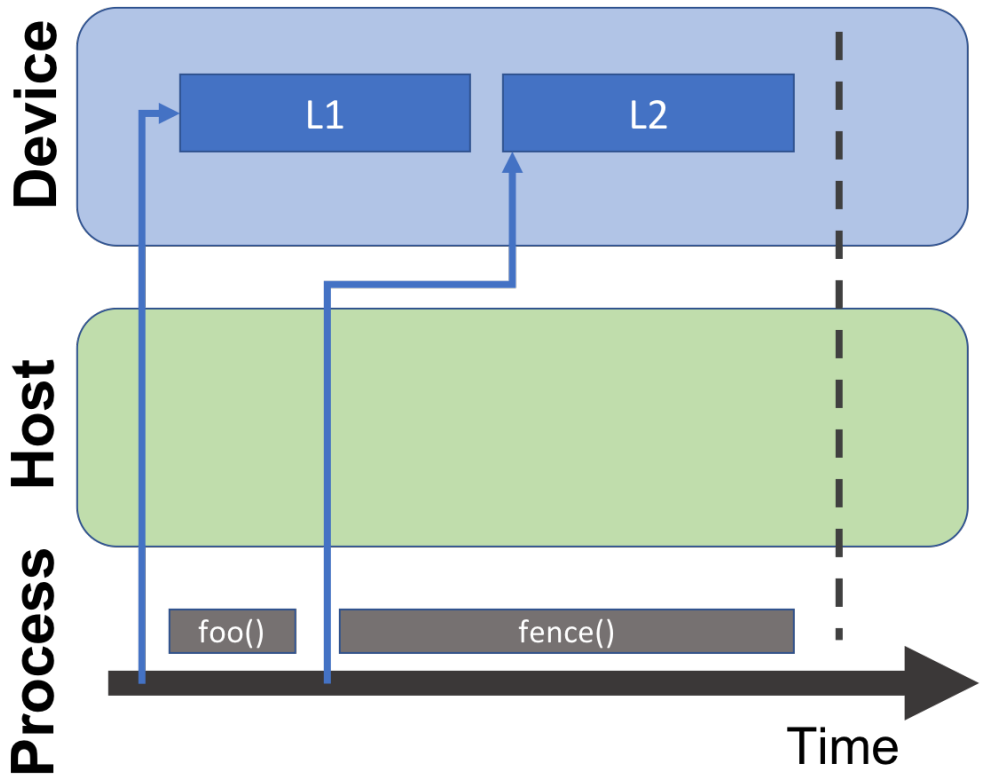
- Dispatches to the same space instance will never overlap
- Executed in order FIFO
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);

parallel_for("L1",
  policy_device, L1);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

Execution Spaces execute in dispatch order

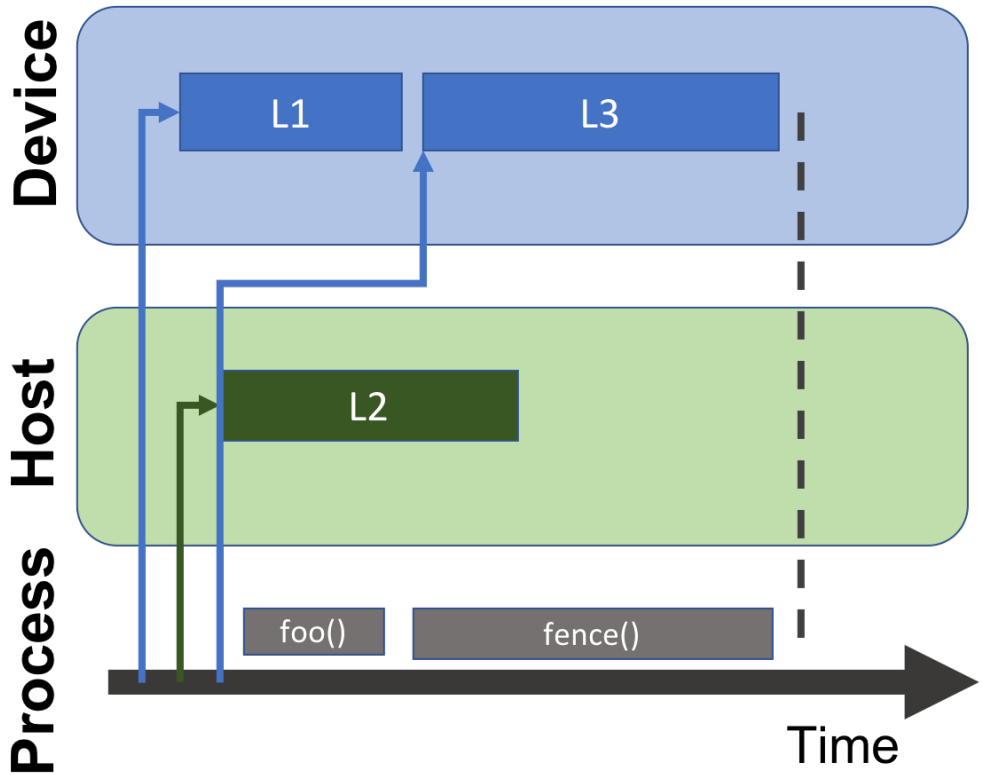
- Dispatches to the same space instance will never overlap
- Executed in order FIFO
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);

parallel_for("L1",
  policy_device, L1);
foo();
parallel_for("L2",
  policy_device, L2);
fence();
```

ExecutionSpaces are Independent

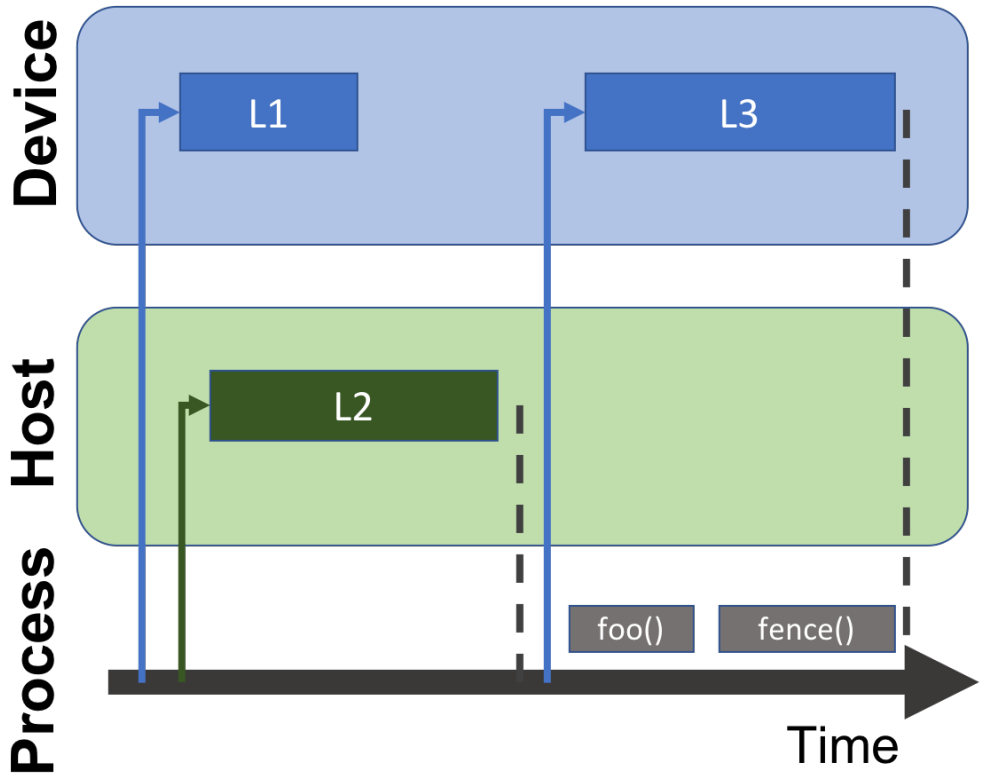
- Dispatches into different ExecutionSpaces may overlap.
- Overlap with process thread functions and each other
- Use `Kokkos::fence()` to wait for completion of all




```
RangePolicy<>
  policy_d(0,N)
RangePolicy<Host>
  policy_host(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);
FunctorL3 L3(...);
parallel_for("L1",
  policy_d, L1);
parallel_for("L2",
  policy_host, L2);
parallel_for("L3",
  policy_d, L3);
foo();
fence();
```

Reality: Some Host Backends Block

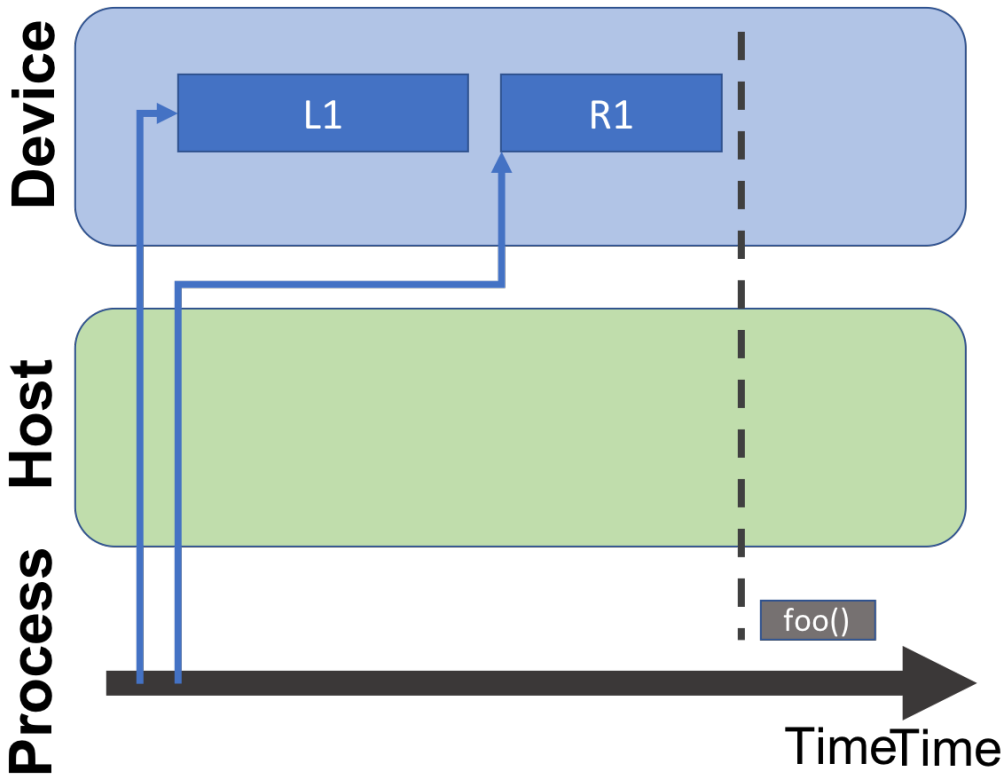
- Most host backends are blocking dispatches (except HPX)
- They never overlap with process thread functions
- But: **Do NOT rely on blocking behavior!!**




```
RangePolicy<>
  policy_d(0,N)
RangePolicy<Host>
  policy_host(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);
FunctorL3 L3(...);
parallel_for("L1",
  policy_d, L1);
parallel_for("L2",
  policy_host, L2);
parallel_for("L3",
  policy_d, L3);
foo();
fence();
```

Reductions to Scalars are Blocking

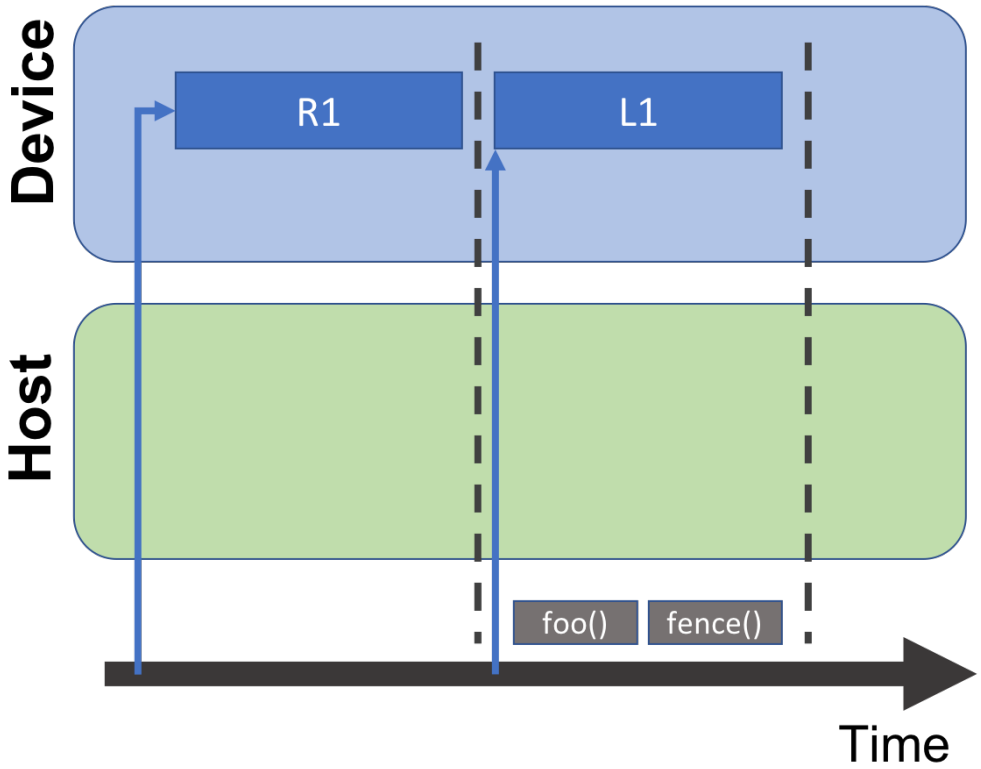
- The call only returns after result is available.
- FIFO implies, every other kernel in the same ExecutionSpace instance will be done too




```
RangePolicy<>
  policy_d(0,N)
FunctorL1 L1(...);
FunctorR1 R1(...);

double result;
parallel_for("L1",
  policy_d, L1);
parallel_reduce("R1",
  policy_d, R1, result);
foo();
```

Reductions to Scalars are Blocking

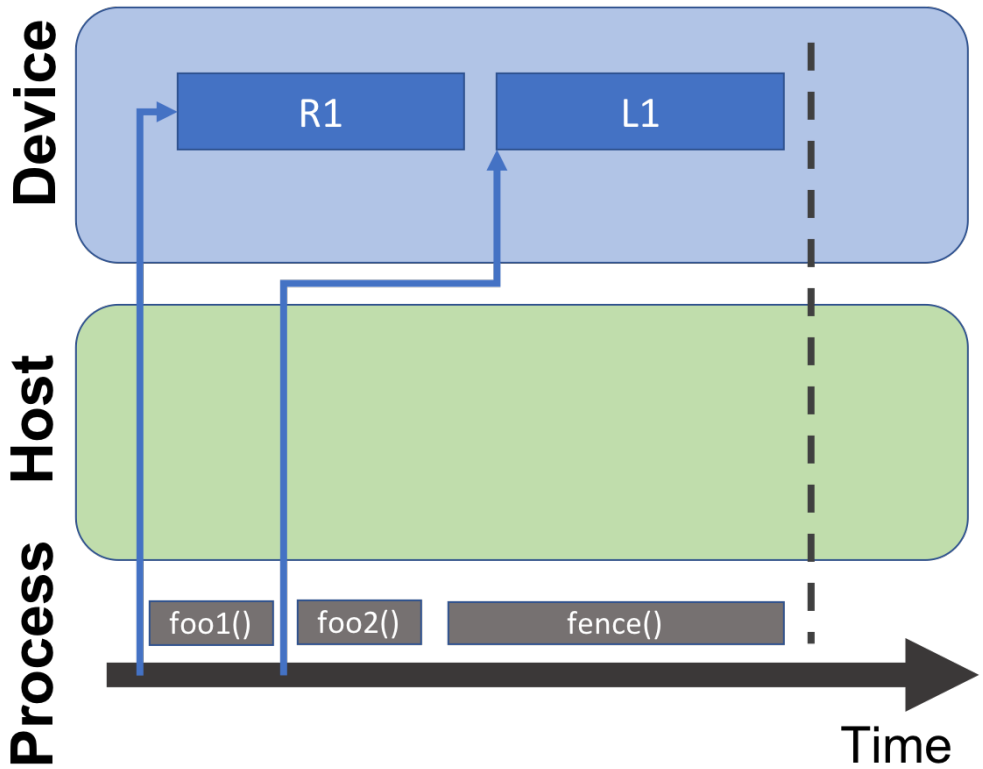
- The call only returns after result is available.
- For subsequent dispatches previous rules apply




```
RangePolicy<>
  policy_d(0,N)
FunctorL1 L1(...);
FunctorR1 R1(...);

double result;
parallel_reduce("R1",
  policy_d, R1, result);
parallel_for("L1",
  policy_d, L1);
foo();
fence();
```

Reductions to Views are Non-blocking

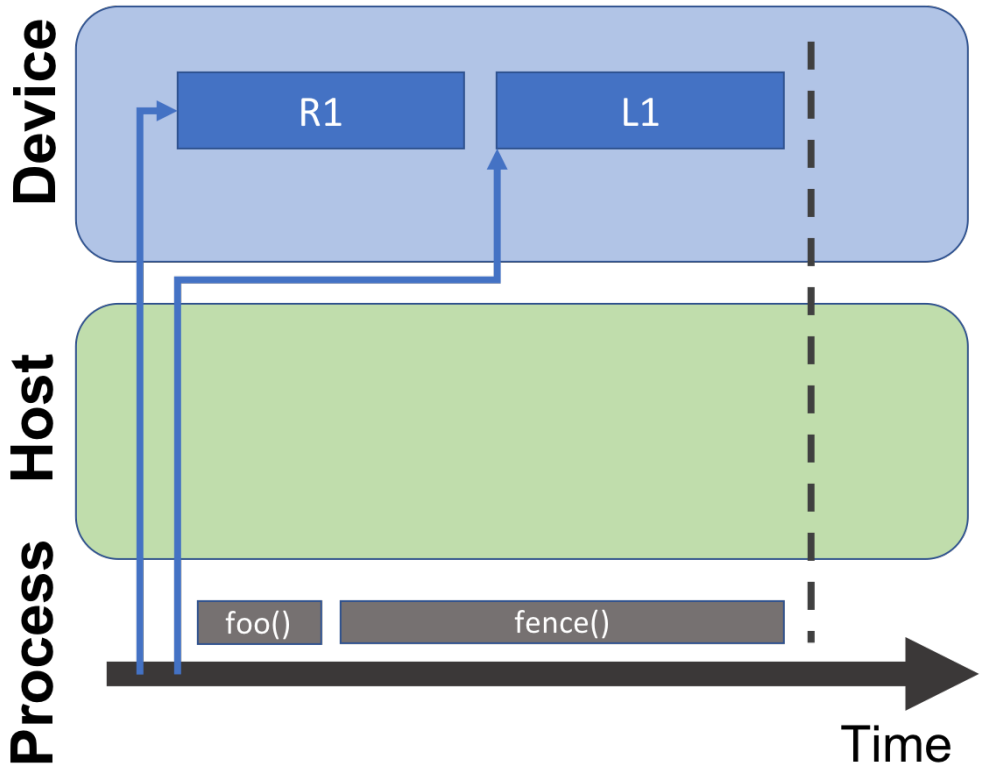
- Behave like a `parallel_for`
- Results are only available after a `Kokkos::fence()`
- Even true for unmanaged `Views` of host variables!




```
...
double result;
View<double, HostSpace>
  v_result(&result);
parallel_reduce("R1",
  policy_d,R1,v_result);
foo1();
parallel_for("L1",
  policy_d, L1);
foo2();
fence();
```

Simple Parallel Loop

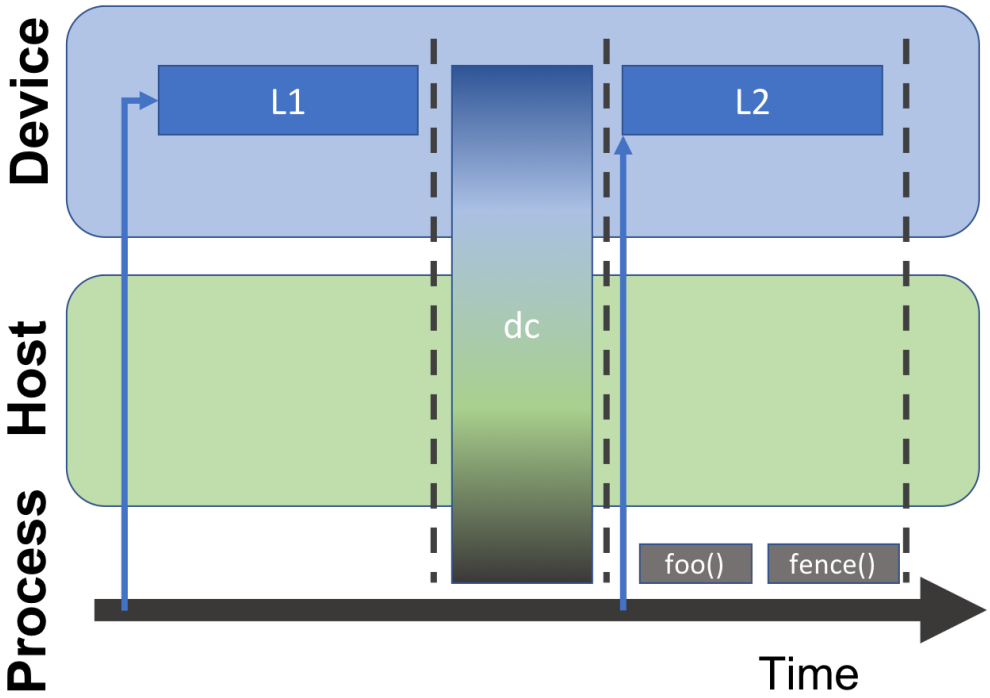
- Asynchronous
- Overlaps with host functions
- Use `Kokkos::fence()` to wait for completion




```
RangePolicy<>
  policy_device(0,N)
FunctorL1 L1(...);
FunctorL2 L2(...);

parallel_for("L1",
  policy_device, L1);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

2-Argument deep_copy is fully blocking

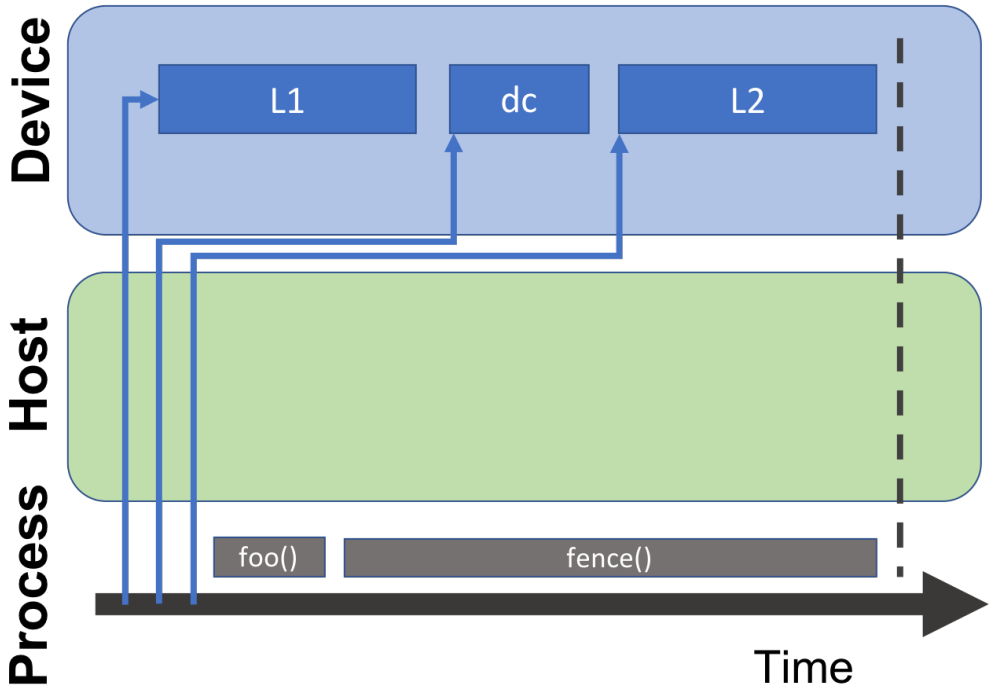
- Implies a full `fence` before the copy
- Copy is done by the time call returns.
- Even if it is a no-op due to `src == dst`!




```
parallel_for("L1",
  policy_device, L1);
deep_copy(dest,src);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

deep_copy with space argument are non-blocking

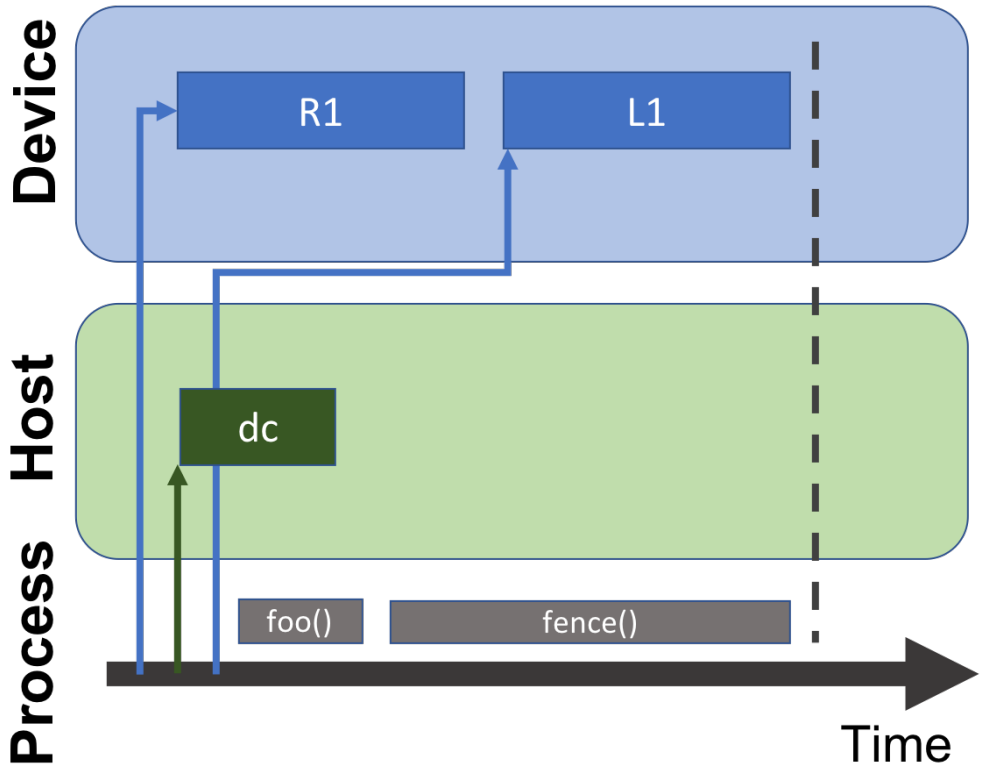
- Execute in dispatch order in the queue of the space
- Overlap with host process functions
- Use `Kokkos::fence()` to wait for completion




```
parallel_for("L1",
  policy_device, L1);
deep_copy(device,
  dest,src);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

deep_copy with space argument are non-blocking

- Execute in dispatch order in the queue of the space
- Overlap with other execution spaces
- Use `Kokkos::fence()` to wait for completion




```
parallel_for("L1",
  policy_device, L1);
deep_copy(host(),
  dest,src);
parallel_for("L2",
  policy_device, L2);
foo();
fence();
```

So what about CUDA streams?

Up to now we only used default execution space instances, but what if you want to have concurrent kernels on the GPU?

So what about CUDA streams?

Up to now we only used default execution space instances, but what if you want to have concurrent kernels on the GPU?

Execution Space Instances

Execution Space instances behave largely like CUDA streams

So what about CUDA streams?

Up to now we only used default execution space instances, but what if you want to have concurrent kernels on the GPU?

Execution Space Instances

Execution Space instances behave largely like CUDA streams

You can create different instances:

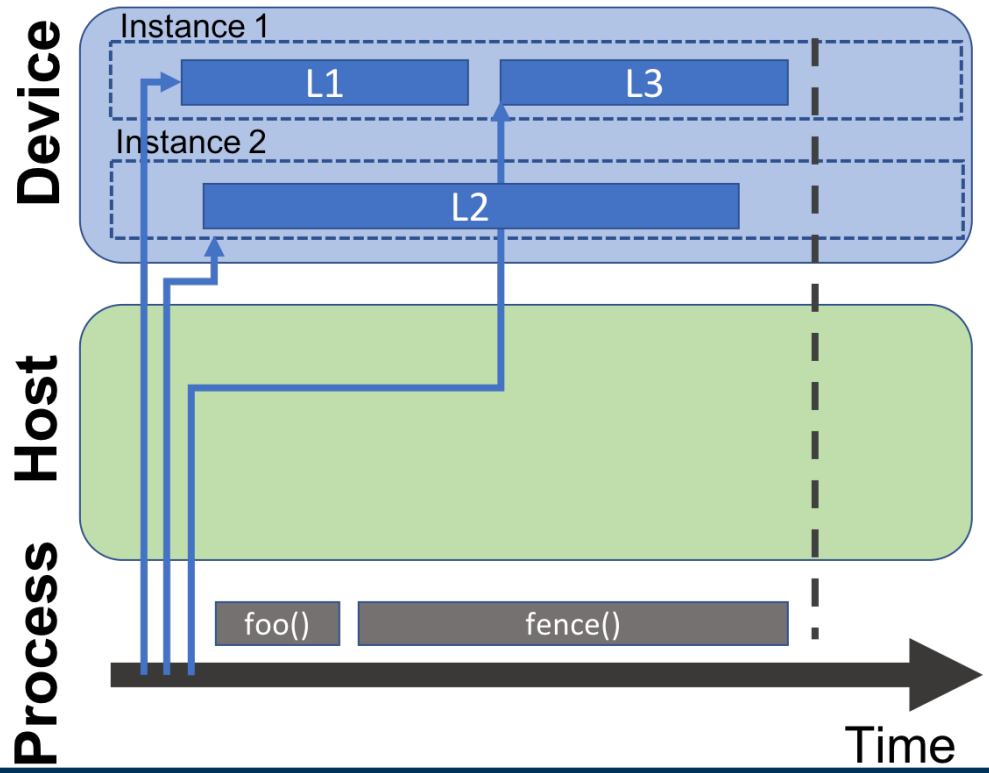
- Fairly new capability.
- Initial version more for interoperability with CUDA/HIP.
  - Give a `cudaStream_t` to the constructor of the instance:

```
cudaStream_t stream = ...;
Kokkos::Cuda cuda_instance(stream);
```

- Generic versions upcoming (e.g. create generic instances)
- Not all spaces support having multiple distinct instances.
- ExecutionSpace instances are like shared pointers.

Instances of Execution Spaces own a exec queue

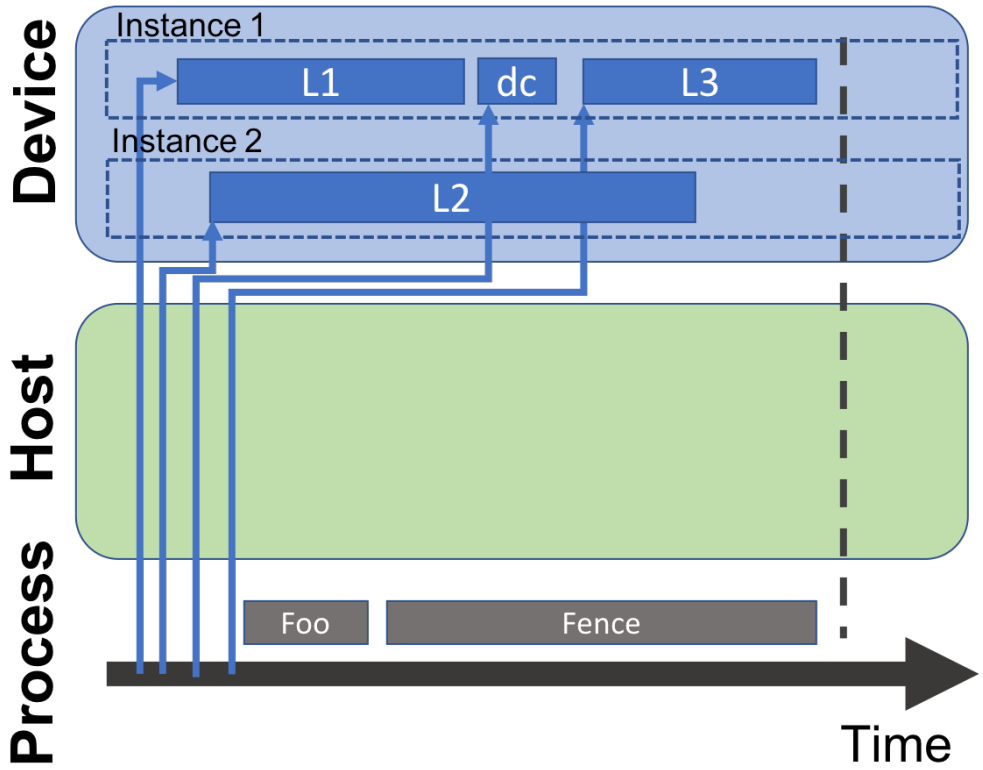
- Work dispatched to different instances overlaps with each other
- Overlaps with host process functions
- Use `Kokkos::fence()` to wait for completion of all




```
Device dev1(...);
Device dev2(...);
RangePolicy<Device>
  policy_d1(dev1,0,N);
RangePolicy<Device>
  policy_d2(dev2,0,N);

parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
parallel_for("L3",
  policy_d1, L3);
foo();
fence();
```

deep_copy with an instance argument also overlap

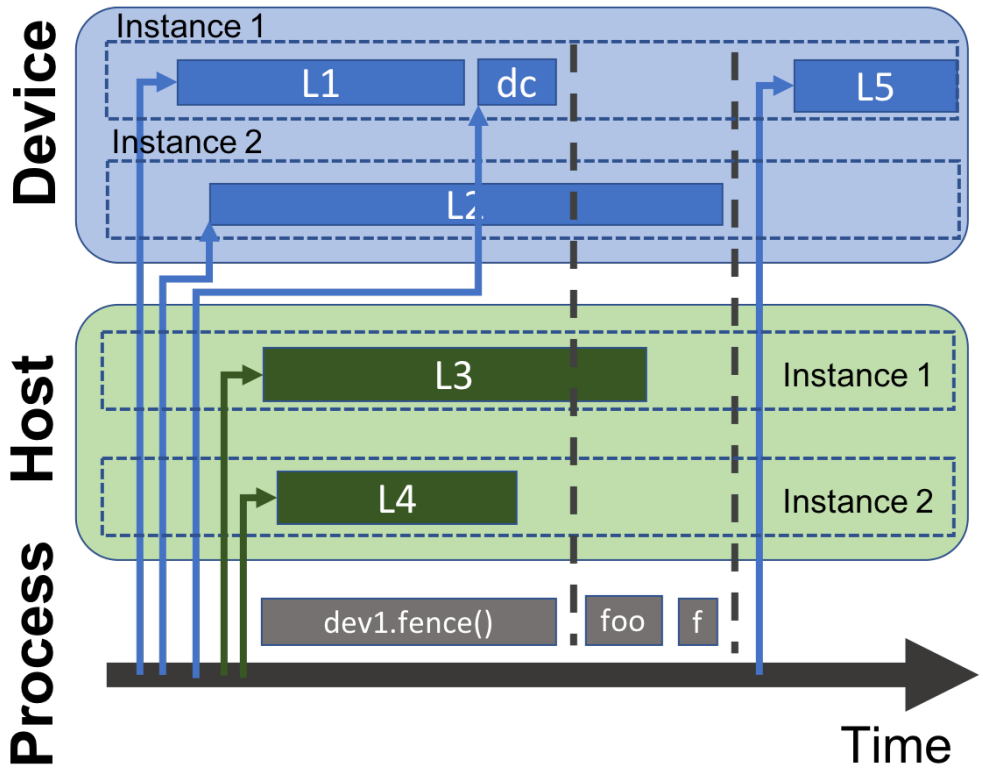
- deep_copy with an instance argument are like any other parallel operation
- Overlaps with parallel operations in other instance




```
parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
deep_copy(dev1,
  dest, src);
parallel_for("L3",
  policy_d1, L3);
foo();
fence();
```

There are instance fences

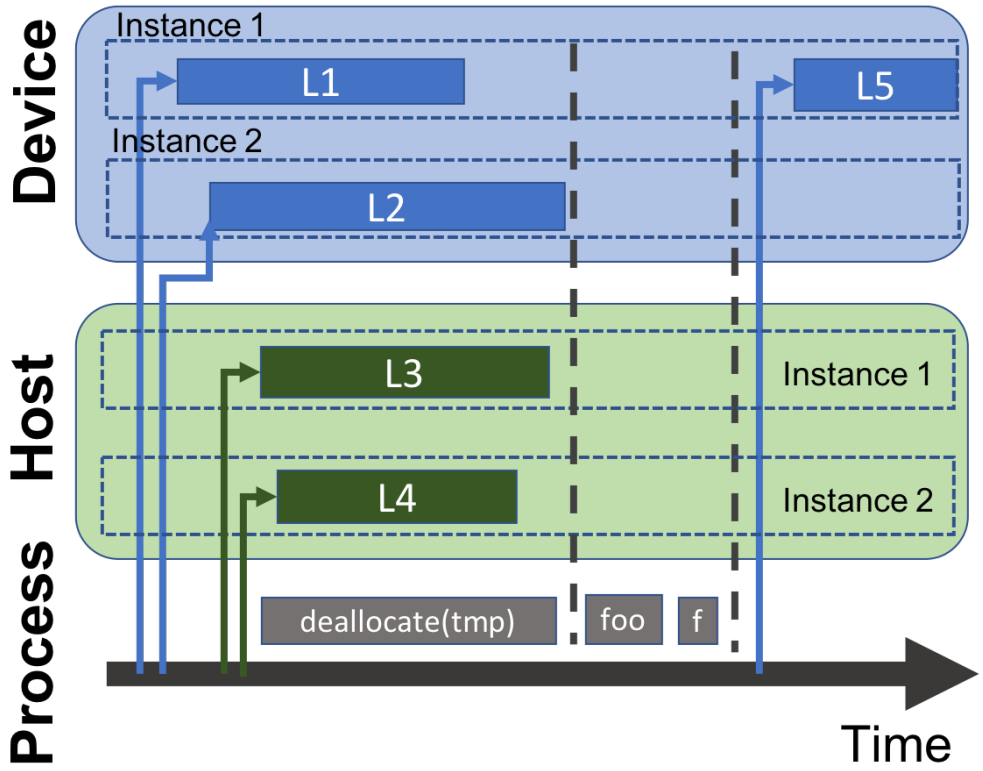
- Use instance specific fence to only wait on that instance
- Operations in other instances can overlap with that fence
- Use `Kokkos::fence()` to wait for all outstanding ops




```
parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
deep_copy(dev1,
  dest, src);
parallel_for("L3",
  policy_h1, L3);
parallel_for("L4",
  policy_h2, L4);
dev1.fence();
foo();
fence();
parallel_for("L5",
  policy_d1, L5);
```

Reality Check: Kokkos Views deallocation implies fence!

- Due to limitations of reference counting, deallocations fence!
- **Important:** this is implementation limitation not semantic!
- **Do NOT rely on deallocations fencing!**




```
{
View<..> tmp(...);
parallel_for("L1",
  policy_d1, L1);
parallel_for("L2",
  policy_d2, L2);
parallel_for("L3",
  policy_h1, L3);
parallel_for("L4",
  policy_h2, L4);
}
foo();
fence();
parallel_for("L5",
  policy_d1, L5);
```

- Execution Space Instances execute work in order of dispatch.
- Operations dispatched to different Execution Space Instances can overlap.
- Each Execution Space type has a default instance as a singleton.
- Use `Kokkos::fence()` to wait for completion of ALL outstanding work.
- Use `exec_space_instance.fence()` to wait for completion of outstanding work dispatched to a specific execution space instance.

Task parallelism

Fine-grained dependent execution.

Learning objectives:

- Basic interface for fine-grained tasking in Kokkos
- How to express dynamic dependency structures in Kokkos tasking
- When to use Kokkos tasking

Recall that **data parallel** code is composed of a pattern, a policy, and a functor

```
Kokkos::parallel_for(
  Kokkos::RangePolicy<>(exec_space, 0, N),
  SomeFunctor()
);
```

Task parallel code similarly has a pattern, a policy, and a functor

```
Kokkos::task_spawn(
  Kokkos::TaskSingle(scheduler, TaskPriority::High),
  SomeFunctor()
);
```

What does a task functor look like?

```
struct MyTask {
  using value_type = double;
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result);
};
```

- Tell Kokkos what the value type of your task's output is.
- Take a team member argument, analogous to the team member passed in by `Kokkos::TeamPolicy` in hierarchical parallelism
- The output is expressed by assigning to a parameter, similar to with `Kokkos::parallel_reduce`

What policies does Kokkos tasking provide?

- `Kokkos::TaskSingle()`
  - Run the task with a single worker thread
- `Kokkos::TaskTeam()`
  - Run the task with all of the threads in a team
  - Think of it like being inside of a `parallel_for` with a `TeamPolicy`
- Both policies take a scheduler, an optional predecessor, and an optional priority (more on schedulers and predecessors later)

What patterns does Kokkos tasking provide?

- `Kokkos::task_spawn()`
  - `Kokkos::host_spawn()` (same thing, but from host code)
- `Kokkos::respawn()`
  - Argument order is backwards; policy comes second!
  - First argument is ‘this‘ always (not ‘*this‘)
- `task_spawn()` and `host_spawn()` return a `Kokkos::Future` representing the completion of the task (see next slide), which can be used as a predecessor to another operation.

How do futures and dependencies work?

```
struct MyTask {
  using value_type = double;
  Kokkos::Future<double, Kokkos::DefaultExecutionSpace> dep;
  int depth;
  KOKKOS_INLINE_FUNCTION MyTask(int d) : depth(d) { }
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result) {
    if(depth == 1) result = 3.14;
    else if(dep.is_null()) {
      dep =
        Kokkos::task_spawn(
          Kokkos::TaskSingle(member.scheduler()),
          MyTask(depth-1)
        );
      Kokkos::respawn(this, dep);
    }
    else {
      result = depth * dep.get();
    }
  }
};
```

```
template <class Scheduler>
struct MyTask {
  using value_type = double;
  Kokkos::BasicFuture<double, Scheduler> dep;
  int depth;
  KOKKOS_INLINE_FUNCTION MyTask(int d) : depth(d) { }
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result);
};
```

Available Schedulers:

- `TaskScheduler<ExecSpace>`
- `TaskSchedulerMultiple<ExecSpace>`
- `ChaseLevTaskScheduler<ExecSpace>`

```
using execution_space = Kokkos::DefaultExecutionSpace;
using scheduler_type = Kokkos::TaskScheduler<execution_space>;
using memory_space = scheduler_type::memory_space;
using memory_pool_type = scheduler_type::memory_pool;
size_t memory_pool_size = 1 << 22;

auto scheduler =
  scheduler_type(memory_pool_type(memory_pool_size));

Kokkos::BasicFuture<double, scheduler_type> result =
  Kokkos::host_spawn(
    Kokkos::TaskSingle(scheduler),
    MyTask<scheduler_type>(10)
  );
Kokkos::wait(scheduler);
printf("Result␣is␣%f", result.get());
```

- Tasks always run to completion
- There is no way to wait or block inside of a task
  - `future.get()` does not block!
- Tasks that do not `respawn` themselves are complete
  - The value in the `result` parameter is made available through `future.get()` to any dependent tasks.
- The second argument to `respawn` can only be either a predecessor (future) or a scheduler, not a proper execution policy
  - We are fixing this to provide a more consistent overload in the next release.
- Tasks can only have one predecessor (at a time)
  - Use `scheduler.when_all()` to aggregate predecessors (see next slide)

```
using void_future =
  Kokkos::BasicFuture<void, scheduler_type>;
auto f1 =
  Kokkos::task_spawn(Kokkos::TaskSingle(scheduler), X{});
auto f2 =
  Kokkos::task_spawn(Kokkos::TaskSingle(scheduler), Y{});
void_future f_array[] = { f1, f2 };
void_future f_12 = scheduler.when_all(f_array, 2);
auto f3 =
  Kokkos::task_spawn(
    Kokkos::TaskSingle(scheduler, f_12), FuncXY{}
  );
```

- To create an aggregate Future, use `scheduler.when_all()`
- `scheduler.when_all()` always returns a `void` future.
- (Also, any future is implicitly convertible to a `void` future of the same Scheduler type)

Formula

$$F_N = F_{N-1} + F_{N-2}$$
$$F_0 = 0$$
$$F_1 = 1$$

Serial algorithm

```
int fib(int n) {
  if(n < 2) return n;
  else {
    return fib(n-1) + fib(n-2);
  }
}
```

Details:

- Location: `Exercises/tasking`
- Implement the `FibonacciTask` task functor recursively
- Spawn the root task from the host and wait for the scheduler to make it ready

Hints:

- Do the F_{N-1} and F_{N-2} subproblems in separate tasks
- Use a `scheduler.when_all()` to wait on the subproblems

SIMD Types

- SIMD types help vectorize code.
- In particular for **outer-loop** vectorization.
- There are **storage** and **temporary** types.
- Currently considered experimental at https://github.com/Kokkos/simd-math: please try it out and provide feedback.

Blocking Behavior and Streams

- Execution Space Instances execute work in order of dispatch.
- Operations in distinct Execution Space Instances can overlap.
- Each Execution Space type has a default instance.
- Use `Kokkos::fence()` to wait for completion of ALL outstanding work or `exec_space_instance.fence()` to wait on work in a specific execution space instance.

Python and Fortran Data Interoperability

- How to pass data back and forth between C++ Kokkos and Python/Fortran

Kokkos + MPI: how to make it work

- Basic usage
- Performance considerations

PGAS: Global Arrays via Kokkos

- How to write distributed code using a global arrays like interface

Don't Forget: Join our Slack Channel and drop into our office hours on Tuesday.

Updates at: kokkos.link/the-lectures-updates

Recordings/Slides: kokkos.link/the-lectures