GPU Programming, the C++ Way

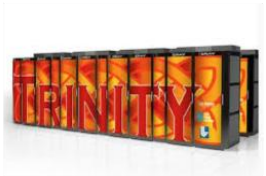
Lecture 18

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

The HPC Hardware Landscape

Current Generation: Programming Models OpenMP 3, CUDA and OpenACC depending on machine



LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3

LLNL SIERRA
IBM Power9 / NVIDIA Volta
CUDA / *OpenMP*[(a)]

ORNL Summit
IBM Power9 / NVIDIA Volta
CUDA / *OpenACC* / *OpenMP*[(a)]

SNL Astra
ARM CPUs
OpenMP 3

Riken Fugaku
ARM CPUs with SVE
OpenMP 3 / *OpenACC*[(b)]

Upcoming Generation: Programming Models OpenMP 5, CUDA, HIP and DPC++ depending on machine

NERSC Perlmutter
AMD CPU / NVIDIA GPU
CUDA / *OpenMP 5*[(c)]

ORNL Frontier
AMD CPU / AMD GPU
HIP / *OpenMP 5*[(d)]

ANL Aurora
Xeon CPUs / Intel GPUs
DPC++ / *OpenMP 5*[(e)]

LLNL El Capitan
AMD CPU / AMD GPU
HIP / *OpenMP 5*[(d)]

(a) Initially not working. Now more robust for Fortran than C++, but getting better

(b) Research effort

(c) OpenMP 5 by NVIDIA.

(d) OpenMP 5 by HPE.

(e) OpenMP 5 by Intel.

Industry Estimate: Cost of Coding

- Typical HPC production app: 300k-600k lines
- Large Scientific Libraries:
  - E3SM: 1,000k lines
  - Trilinos: 4,000k lines

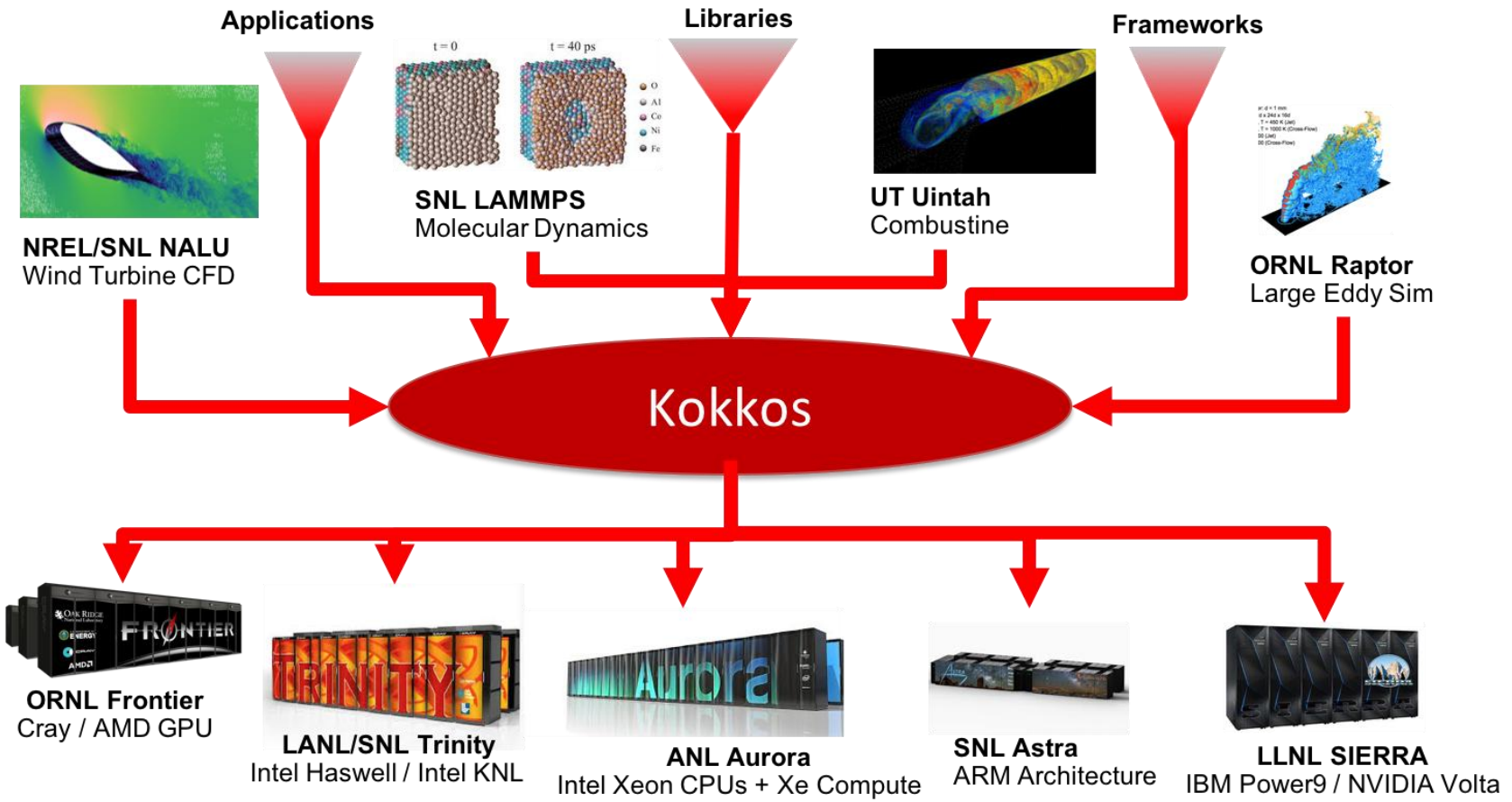
- Conservative estimate: need to rewrite 10% of an app to switch Programming Model

- A full time software engineer writes 10 lines of production code per hour: 20k LOC/year
- Just switching Programming Models costs multiple person-years per app!

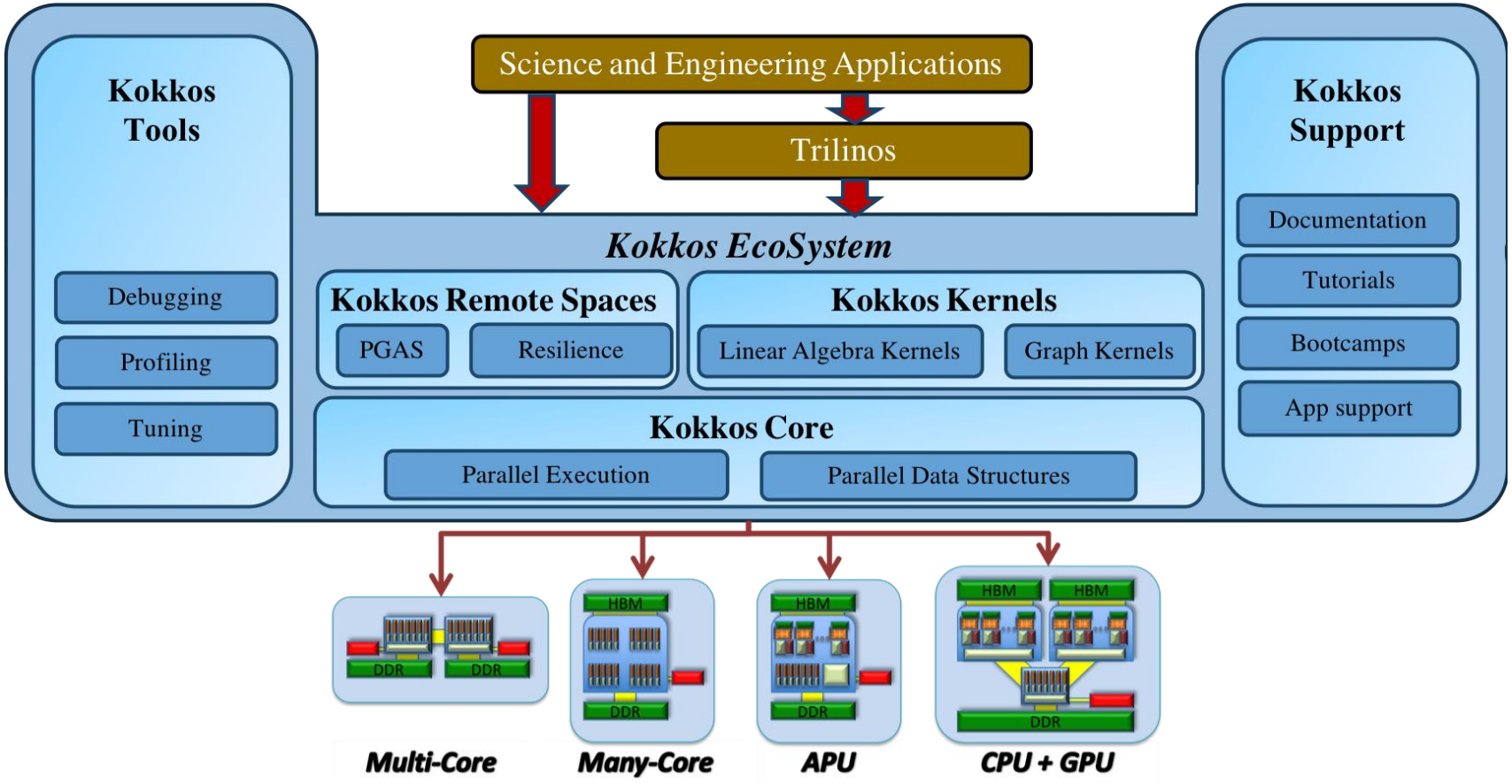
What is Kokkos?

- A C++ Programming Model for Performance Portability
  - Implemented as a template library on top CUDA, HIP, OpenMP, HPX, …
  - Aims to be descriptive not prescriptive
  - Aligns with developments in the C++ standard
- Expanding solution for common needs of modern science and engineering codes
  - Math libraries based on Kokkos
  - Tools for debugging, profiling and tuning
  - Utilities for integration with Fortran and Python
- Is an Open Source project with a growing community
  - Maintained and developed at https://github.com/kokkos
  - Hundreds of users at many large institutions

Example Uses of Kokkos



The Kokkos Ecosystem



Data-parallel Patterns (Algorithms)

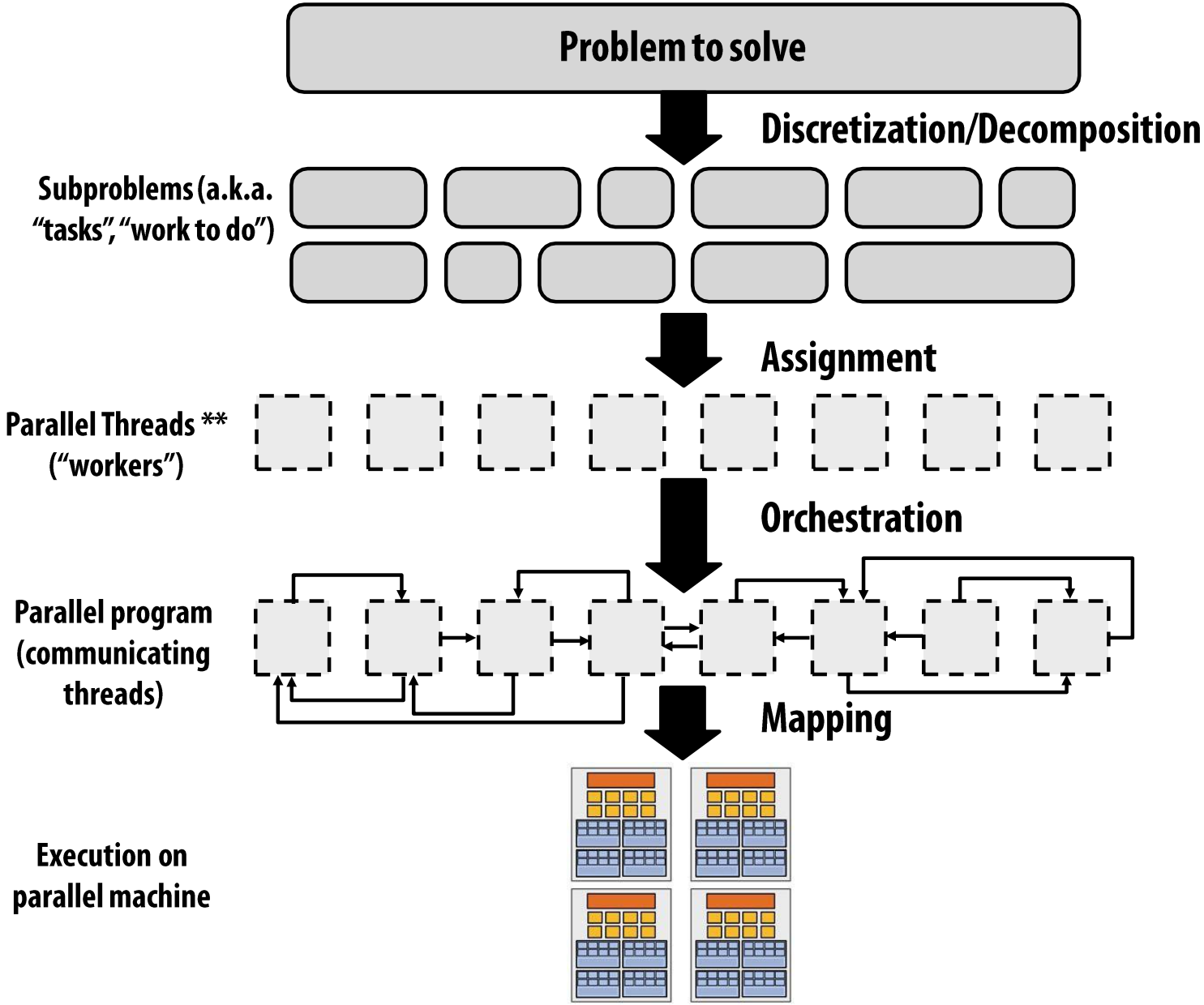
- We will look at:
  - How computational bodies are passed to the Kokkos runtime
  - How work is mapped to execution resources
  - The difference between `parallel_for` and `parallel_reduce`
  - Start parallelizing a simple example

Data-parallel Patterns and Work

- Data parallel patterns and work

```
for (size_t atom_index = 0; atom_index != number_of_atoms; ++atom_index)
{
    atom_forces[atom_index] = calculate_force(... data...);
}
```

- Kokkos maps work to execution resources
  - Each iteration of a computational body is a unit of work
  - An iteration index identifies a particular unit of work
  - An iteration range identifies a total amount of work
- Important concept: Work mapping, orchestration, and assignment
  - You give an iteration range and computational body (kernel) to Kokkos, and Kokkos decides how to map that work to execution resources.


Problem to solve
Discretization/Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ** ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine

Using Kokkos for Data-parallel Patterns (Algorithms)

- How are computational bodies given to Kokkos?
  - As function objects (or lambdas), a common pattern in C++
- Quick review, a function object is a function with associated data. Example:

```
struct parallel_function_object {
    // ...
    void operator()(/* a work assignment */) const
    {
        // ... computational body ...
    }
    // ...
};
```

Using Kokkos for Data-parallel Patterns (Algorithms)

- How is work assigned to function object operators?
  - A total amount of work items is given to a Kokkos pattern (algorithm)

```
parallel_function_object fo;
Kokkos::parallel_for(number_of_iterations, fo);
```

  - And work items are assigned to function objects one-by-one:

```
struct parallel_function_object {
    void operator()(size_t const index) const { /* computational body */ }
};
```

- Warning: concurrency and order
  - Concurrency and ordering of parallel iterations is not guaranteed by the Kokkos runtime.

Using Kokkos for Data-parallel Patterns (Algorithms)

- The complete picture (using function objects):
  1) Defining the function object (operator+data):

```
struct atom_force_function_object {
    ForceType atom_forces;    // array of forces between atoms
    AtomDataType atom_data;   // array of data items for atoms

    atom_force_function_object(ForceType atom_forces, AtomDataType data)
      : atom_forces(forces), atom_data(data) {}

    void operator()(int64_t atom_index) const {
        atom_forces[atom_index] = calculate_force(atom_data);
    }
};
```

  2) Executing in parallel with Kokkos pattern (algorithm):

```
atom_force_function_object fo(atom_forces, data);
Kokkos::parallel_for(number_of_atoms, fo);
```

Using Kokkos for Data-parallel Patterns (Algorithms)

- The complete picture (using function objects):
  - Function objects are tedious -- Lambdas are concise
  - Defining the function object as a lambda and executing in parallel with Kokkos pattern

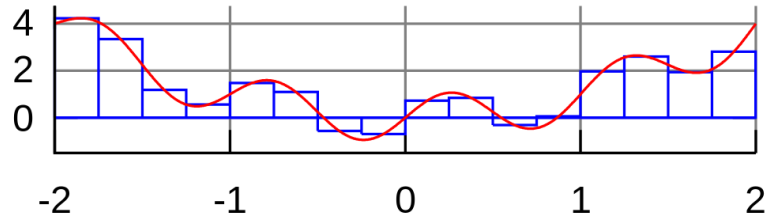
```
Kokkos::parallel_for(number_of_atoms,
    [=](int64_t atom_index) const {
        atom_forces[atom_index] = calculate_force(data);
    });
```

- A lambda is not magic, it is the compiler auto-generating a function object for you
- Warning: Lambda capture and C++ containers
  - For portability to GPU, a lambda must capture by value: `[=]`
  - Don't capture containers (e.g., `std::vector`) by value because it will copy the container's entire contents

Scalar Integration Example

- Riemann-sum-style numerical integration:

$$y = \int_{lower}^{upper} func(x)\, dx$$



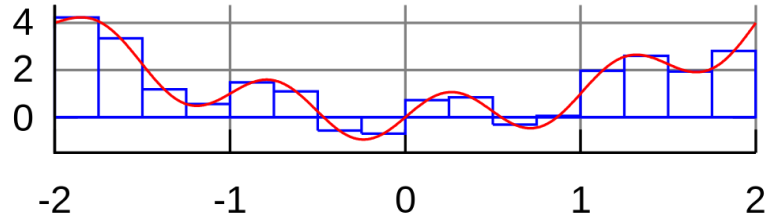

```
double total_integral = 0;
for (int64_t i = 0; i < number_of_intervals; ++i) {
    double const x = lower + (i / number_of_intervals) * (upper - lower);
    total_integral += func(x);
}
total_integral *= dx;
```

- How do we parallelize it?
  - Correctly?

Scalar Integration Example

- Riemann-sum-style numerical integration:

$$y = \int_{lower}^{upper} func(x)\, dx$$




```
double total_integral = 0;
Kokkos::parallel_for(number_of_intervals,
    [=](int64_t const index) {
        double const x = lower + (index / number_of_intervals) * (upper - lower);
        total_integral += func(x);
    },
);
total_integral *= dx;
```

- Compilation error (cannot increment `total_integral`)!
  - Lambda captures are `const` by default

Scalar Integration Example

- An (incorrect) solution to the (incorrect) attempt:

```
double total_integral = 0;
double* total_integral_pointer = &total_integral;
Kokkos::parallel_for(number_of_intervals,
    [=](int64_t const index) {
        double const x = lower + (index / number_of_intervals) * (upper - lower);
        *total_integral_pointer += func(x);
    });
total_integral *= dx;
```

- Body is executed in GPU address space!
  - Can't directly 'point' to CPU addresses
  - Use of pointers is generally a bad idea
- Race condition, `total_integral_pointer` is shared

Use `reduce` instead of plain `for`

- Reductions combine the results contributed by parallel work

```
double total_integral = 0;
Kokkos::parallel_reduce(
    number_of_intervals,
    [=](int64_t const index, double& value_to_update) {
        value_to_update += func(...);
    },
    total_integral);
```

- The operator takes two arguments: a work index and a value to update.
- The second argument is a thread-private value that is managed by Kokkos; it is not the final reduced value
- Before returning, the thread-local variables are reduced using the reduction operation (default `Kokkos::Sum<>`)

Kokkos Debug Support

- Giving unique names to each kernel is immensely helpful for debugging and profiling. You will regret it if you don't do that!
- Non-nested parallel patterns can take an optional string argument
  - The label doesn't need to be unique, but it is helpful
  - Anything convertible to `std::string`
  - Used by profiling and debugging tools

```
double total_integral = 0;
Kokkos::parallel_reduce(
    "reduction", number_of_intervals,
    [=](int64_t const index, double& value_to_update) {
        value_to_update += func(...);
    },
    total_integral);
```

Summary so far

- Simple usage is similar to C++ standard algorithms, advanced features are also straightforward
  - In fact, as we will see, you can use HPX to invoke Kokkos 'patterns'
- Three common data-parallel patterns are `parallel_for`, `parallel_reduce`, and `parallel_scan`
  - Kokkos calls these 'patterns', we call them 'algorithms'
- A parallel computation is characterized by its pattern, policy, and body
- User provides computational bodies as function objects or lambdas that handle a single work item
- All of this is very similar to C++ standard parallel algorithms

Kokkos Views

Kokkos Views: Motivation

- Example: running `daxpy` (vector scaling/addition) on the GPU:

```
double* x = new double[N];    // also y
Kokkos::parallel_for("DAXPY", N,
    [=](int64_t const i) {
      y[i] = a * x[i] + y[i];
    });
```

- Problem: x and y reside in CPU memory
- Solution: We need a way of storing data (multidimensional arrays) that can be communicated to an accelerator (GPU)
  - ⇒ Kokkos Views

Kokkos Views

- View abstraction
  - A lightweight C++ class with a pointer to array data and a little meta-data,
  - that is templated on the data type (and other things)
- High-level example of Views for `daxpy` using lambda:

```
Kokkos::View<double*, ...> x(...), y(...);
// ... populate x, y...
Kokkos::parallel_for("DAXPY", N,
    [=](int64_t const i) {
        // Views x and y are captured by value (copy)
        y(i) = a * x(i) + y(i);
    });
```

- Important point
  - Views are lightweight (like pointers), so copy them into your function objects

Kokkos Views

- View overview:
  - Multi-dimensional array of 0 or more dimensions
    - scalar (0), vector (1), matrix (2), etc.
  - Number of dimensions (rank) is fixed at compile-time.
  - Arrays are rectangular, not ragged.
  - Sizes of dimensions set at compile-time or runtime e.g., 2x20, 50x50, etc.
  - Access elements via ”(…)” operator.
- Example:

```
Kokkos::View<double***> data ("label", N0, N1, N2);  // 3 runtime, 0 compile
Kokkos::View<double**[N2]> data ("label", N0, N1);   // 2 runtime, 1 compile
Kokkos::View<double*[N2][N1]> data ("label", N0);    // 1 runtime, 2 compile
Kokkos::View<double[N2][N1][N0]> data ("label");     // 0 runtime, 3 compile
data (i, j, k) = 5.3;   // access
```

- Note: runtime-sized dimensions must come first.

Lifecycle of a Kokkos View

- Allocations only happen when explicitly specified
  - i.e., there are no hidden allocations
- Copy construction and assignment are shallow (like pointers)
  - So, you pass Views by value, not by reference
- Reference counting is used for automatic deallocation
  - They behave like `std::shared_ptr`
- Example:

```
Kokkos::View<double*[5]> a("a", N), b("b", K);
a = b;
View<double**> c(b);
b(0, 2) = 2;
c(0, 2) = 3;
print_value(a(0, 2));   // prints 3.0
```

View Properties

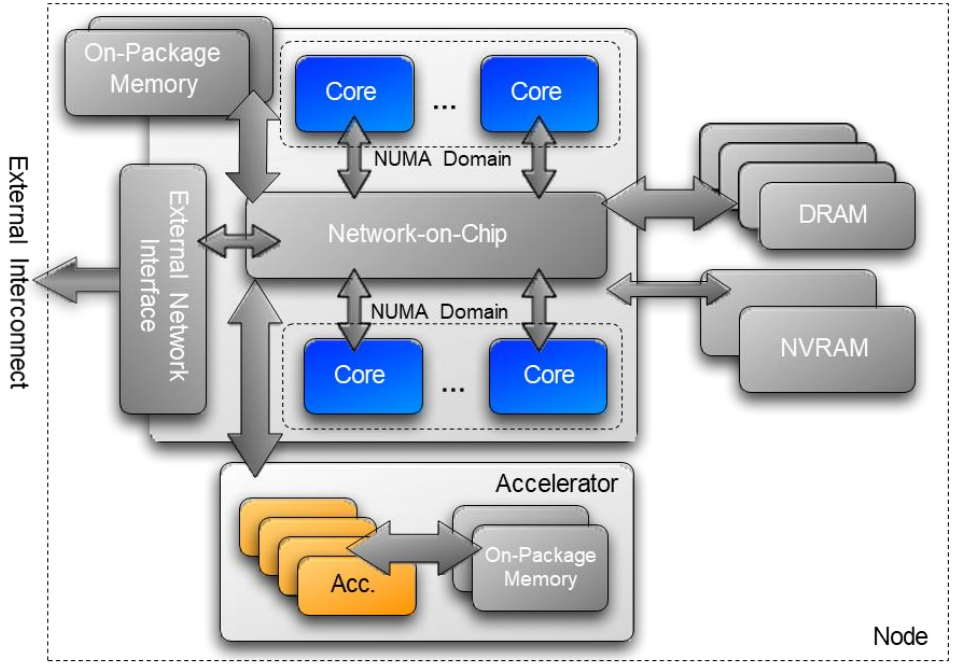
- Accessing a View's sizes is done via its `extent(dim)` function
- Static extents can additionally be accessed via `static_extent(dim)`
- You can retrieve a raw pointer via its `data()` function.
- The label can be accessed via `label()`
- Example:

```
Kokkos::View<double*[5]> a("A", N0);
assert(a.extent(0) == N0);
assert(a.extent(1) == 5);
static_assert(a.static_extent(1) == 5);
assert(a.data() != nullptr);
assert(a.label() == "A");
```

Execution Spaces

Execution Spaces

- A homogeneous set of cores and an execution mechanism (i.e., “place to run code”)



- Execution spaces: Serial, Threads, OpenMP, Cuda, HIP, HPX, ...

Execution Spaces

- Changing the parallel execution space:

```
// default execution space
Kokkos::parallel_for("Label",
    number_of_intervals,    // => RangePolicy<>(0, number_of_intervals)
    [=](int64_t const i) {
        /* ...  body  ... */
    });

// custom execution space
Kokkos::parallel_for("Label",
    Kokkos::RangePolicy<ExecutionSpace>(0, number_of_intervals),
    [=](int64_t const i) {
        /* ...  body  ... */
    });
```

Execution Spaces

- Requirements for enabling execution spaces:
  - Kokkos must be compiled with the execution spaces enabled
  - Execution spaces must be initialized (and finalized)
  - Functions must be marked with a macro for non-CPU spaces
  - Lambdas must be marked with a macro for non-CPU spaces

```
struct parallel_function_object {
    KOKKOS_INLINE_FUNCTION double helper_function(int64_t const index) const
    { /*...*/ }

    KOKKOS_INLINE_FUNCTION void operator()(int64_t const index) const
    {
        helper_function(index);
    }
};

// Where Kokkos defines:
#define KOKKOS_INLINE_FUNCTION inline                    /* #if CPU only */
#define KOKKOS_INLINE_FUNCTION inline __device__ __host__ /* #if CPU+Cuda */
```

Execution Spaces

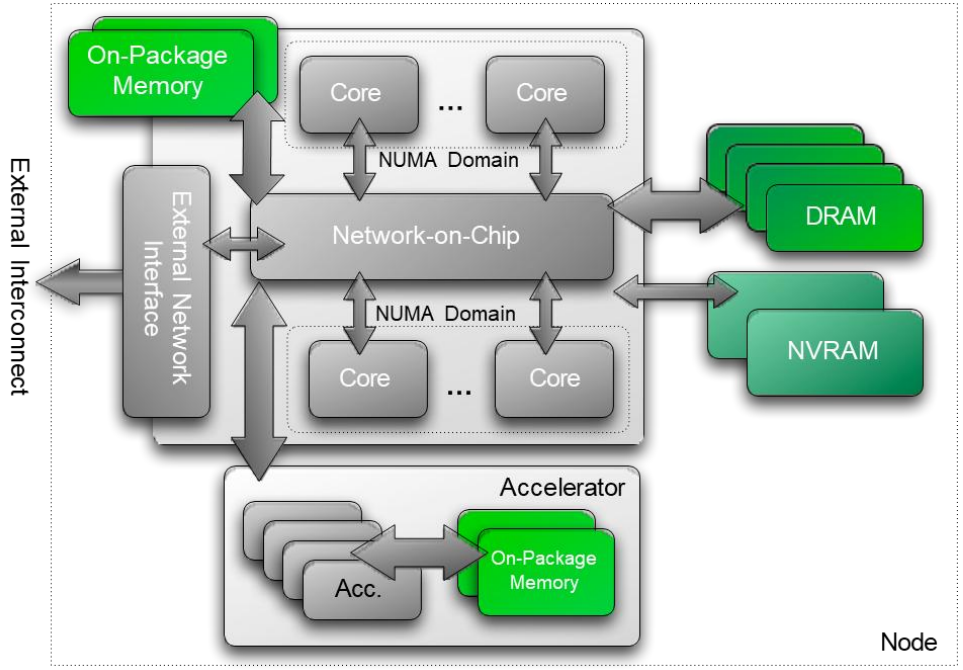
- Lambda annotation with KOKKOS_LAMBDA macro

```
Kokkos::parallel_for("Label",
    number_of_intervals,
    KOKKOS_LAMBA(int64_t const i) {
        /* ...  body  ... */
    });

#define KOKKOS_LAMBDA [=]                        /* #if CPU only */
#define KOKKOS_LAMBDA [=] __device__ __host__ /* #if CPU+Cuda */
```

Memory Spaces

Memory Spaces

- Memory space:
  - Explicitly-manageable memory resource (i.e., “place to put data”)



Memory Spaces

- Every view stores its data in a memory space set at compile time.

```
Kokkos::View<double***, MemorySpace> data(...);
```

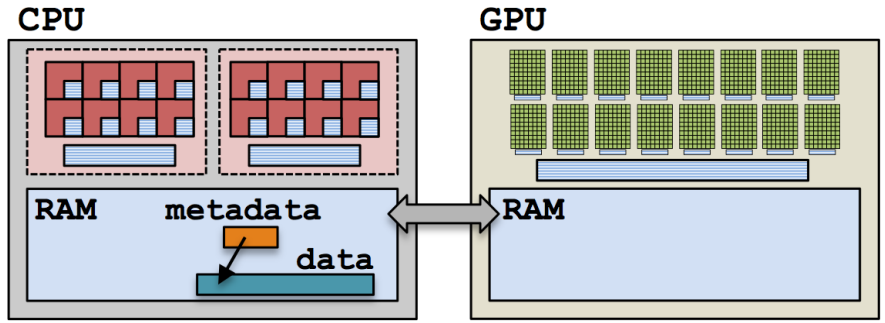
- Available memory spaces:
  - `HostSpace`, `CudaSpace`, `CudaUVMSpace`, ... more
- Each execution space has a default memory space
- If no Space is provided, the view's data resides in the default memory space of the default execution space

```
// Equivalent:
Kokkos::View<double*> a("A", N);
Kokkos::View<double*, Kokkos::DefaultExecutionSpace::memory_space> b("B", N);
```

Memory Spaces

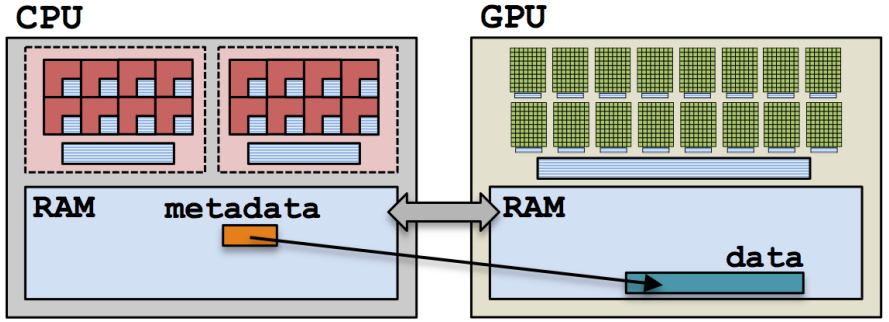
- Example: `HostSpace`

```
Kokkos::View<double**, Kokkos::HostSpace> host_view(/*...*/);
```



- Example: `CudaSpace`

```
Kokkos::View<double**, Kokkos::CudaSpace> cuda_view(/*...*/);
```



Execution and Memory Spaces

- Example: summing an array with the GPU
  - (failed) Attempt 1: View lives in CudaSpace

```
Kokkos::View<double*, Kokkos::CudaSpace> array("array", size);
for (int64_t i = 0; i < size; ++i)
{
    array(i) = // ...read from file
}

double sum = 0;
Kokkos::parallel_reduce(
    "Label", Kokkos::RangePolicy<Kokkos::Cuda>(0, size),
    KOKKOS_LAMBDA(int64_t const index, double& value_to_update) {
        value_to_update += func(array(index));
    },
    sum);
```

Fault!

Execution and Memory Spaces

- Example: summing an array with the GPU
  - (failed) Attempt 2: View lives in HostSpace

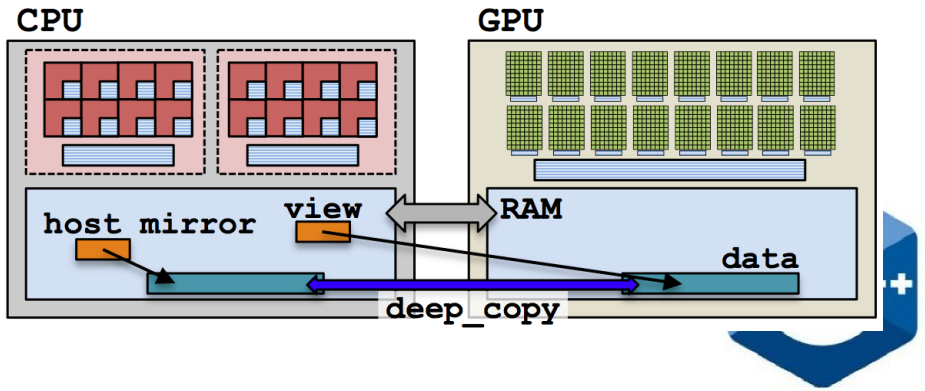
```
Kokkos::View<double*, Kokkos::HostSpace> array("array", size);
for (int64_t i = 0; i < size; ++i)
{
    array(i) = // ...read from file
}

double sum = 0;
Kokkos::parallel_reduce(
    "Label", Kokkos::RangePolicy<Kokkos::Cuda>(0, size),
    KOKKOS_LAMBDA(int64_t const index, double& value_to_update) {
        value_to_update += func(array(index));
    },
    sum);
```

Illegal Access!

Execution and Memory Spaces

- What's the solution?
  - `CudaUVMSpace`
  - `CudaHostPinnedSpace` (skipping)
  - Mirroring
- Mirrors are views of equivalent arrays residing in possibly different memory spaces

```
// Mirroring schematic
using view_type = Kokkos::View<double**, Space>;
view_type view(...);
view_type::HostMirror host_view =
   Kokkos::create_mirror_view(view);
```



Mirroring Pattern

- Create a `view`'s array in some memory space

```
using view_type = Kokkos::View<double*, Space>;
view_type view(...);
```

- Create `host_view`, a mirror of the `view`'s array residing in the host memory space.

```
view_type::HostMirror host_view = Kokkos::create_mirror_view(view);
```

- Populate `host_view` on the host (from file, etc.)
- Deep copy `host_view`'s array to `view`'s array

```
Kokkos::deep_copy(view, host_view);
```

- Launch a kernel processing the `view`'s array

```
Kokkos::parallel_for("Label", Kokkos::RangePolicy<Space>(0, size),
    KOKKOS_LAMBDA(int64_t const index) { /* use and change view; */ });
```

- If needed, deep copy the `view`'s updated array back to the `host_view`'s array to write file, etc.

```
Kokkos::deep_copy(host_view, view);
```

Mirrors of Views in HostSpace

- What if the View is in HostSpace too? Does it make a copy?

```
using view_type = Kokkos::View<double*, Space>;
view_type view("test", 10);
view_type::HostMirror host_view = Kokkos::create_mirror_view(view);
```

- `create_mirror_view` allocates data only if the host process cannot access `view`'s data, otherwise `host_view` references the same data
- `create_mirror` always allocates data
- Reminder: Kokkos never performs a hidden deep copy

Views and Spaces Summary

- Data is stored in Views that are “pointers” to multi-dimensional arrays residing in memory spaces
- Views abstract away platform-dependent allocation, (automatic) deallocation, and access
- Heterogeneous nodes have one or more memory spaces
- Mirroring is used for performant access to views in host and device memory
- Heterogeneous nodes have one or more execution spaces
- You control where parallel code is run by a template parameter on the execution policy, or by compile-time selection of the default execution space

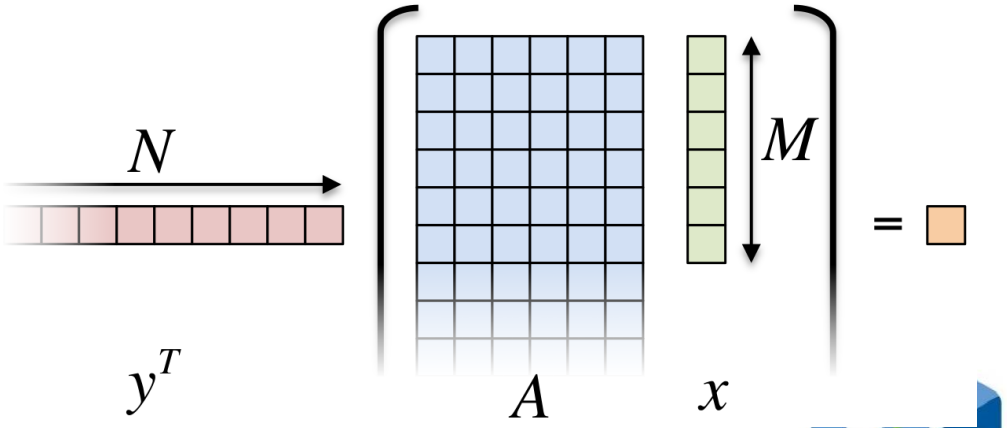
Managing Memory Access Patterns for Performance Portability

- How the View's Layout parameter controls data layout
- How memory access patterns result from Kokkos mapping parallel work indices and layout of multidimensional array data

Example: Inner Product

```
Kokkos::parallel_reduce(
    "Label", RangePolicy<ExecutionSpace>(0, N),
    KOKKOS_LAMBDA(size_t row, double& value_to_update) {
        double this_rows_sum = 0;
        for (size_t entry = 0; entry < M; ++entry)
            this_rows_sum += A(row, entry) * x(entry);
        value_to_update += y(row) * this_rows_sum;
    },
    result);
```

- Driving question:
  How should A be laid out in memory?

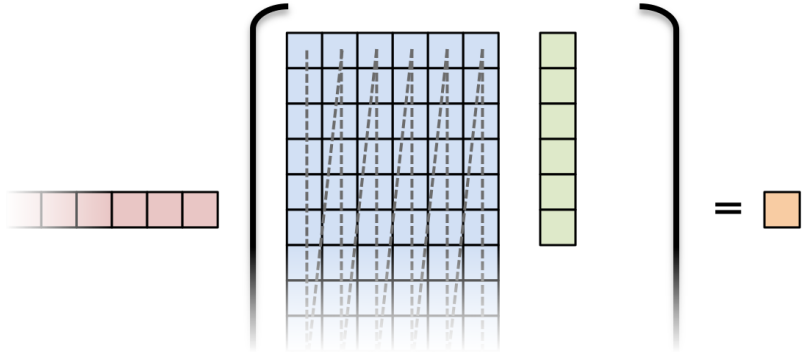


Example: Inner Product

- Layout is the mapping of multi-index to memory:

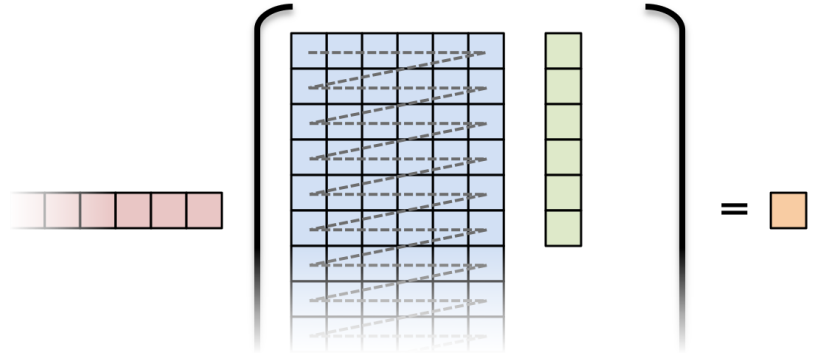
`LayoutLeft`

- in 2D, “column-major”



`LayoutRight`

- in 2D, “row-major”



Important concept: Layout

- Every View has a (optional) multidimensional array Layout set at compile-time

```
Kokkos::View<double***, Layout, Space> name(...);
```

- Most-common layouts are:
  - `LayoutLeft` and `LayoutRight`
  - `LayoutLeft`: left-most index is stride 1.
  - `LayoutRight`: right-most index is stride 1.
- If no layout specified, the default for that memory space is used.
  - `LayoutLeft` for `CudaSpace`
  - `LayoutRight` for `HostSpace`.
- Layouts are extensible: ≈ 50 lines
  - You could create your own, e.g., with custom striding or tiling
- Advanced layouts: `LayoutStride`, `LayoutTiled`, …

What was not covered

- Didn't cover many things
  - BuildSystem integration
  - Non-Sum reductions / multiple reductions
  - Multidimensional loops
  - Advanced data structures
  - Subviews
  - Atomic operations and Scatter Contribute patterns
  - Team Scratch memory (GPU shared memory)
  - SIMD vectorization
  - Tools for Profiling, Debugging and Tuning
  - Math Kernels
  - Etc.

Kokkos and HPX

Integrate Kokkos with HPX

- Use HPX/Kokkos interoperability library
  - https://github.com/STEllAR-GROUP/hpx-kokkos
  - Pre-installed in Docker container used for assignments
- It implements:
  - Asynchronous versions of `Kokkos::parallel_for`, `Kokkos::parallel_reduce`, and `Kokkos::parallel_scan`
  - Asynchronous version of `Kokkos::deep_copy`
  - HPX executors that forward work to corresponding Kokkos execution spaces
  - HPX execution policy that forwards work to corresponding Kokkos execution spaces for use with HPX parallel algorithms
    - Synchronous and asynchronous
  - HPX parallel algorithm specializations for the execution policy above

Build System Support

- Build system support is straightforward (using CMake)

```
project(spring2025-csc4700-hpx-kokkos)

set(CMAKE_CXX_STANDARD 23)      # possible values: 17, 20, 23

find_package(HPX REQUIRED)
find_package(Kokkos REQUIRED)
find_package(HPXKokkos REQUIRED)

# create an executable hpx_kokkos_example from hpx_kokkos_example.cpp
# and link it to HPXKokkos, Kokkos, and HPX
add_executable(hpx_kokkos_example hpx_kokkos_example.cpp)
target_link_libraries(hpx_kokkos_example PRIVATE
    HPXKokkos::hpx_kokkos Kokkos::kokkos HPX::hpx)
```

Asynchronous parallel_for

```
void async_parallel_for()
{
    Kokkos::View<int*, Kokkos::HostSpace> result_host("result_host", N);
    Kokkos::View<int*, Kokkos::CudaSpace> result("result", N);

    for (std::size_t i = 0; i != N; ++i) result_host(i) = 0;

    Kokkos::CudaSpace inst;
    hpx::future<void> f1 = hpx::kokkos::deep_copy_async(inst, result, result_host);

    hpx::future<void> f2 = hpx::kokkos::parallel_for_async(
        Kokkos::RangePolicy<Kokkos::CudaSpace>(inst, 0, N),
        KOKKOS_LAMBDA(int i) { result(i) = i; });

    hpx::future<void> f3 = hpx::kokkos::deep_copy_async(inst, result_host, result);

    hpx::wait_all(f1, f2, f3);

    for (std::size_t i = 0; i != N; ++i) assert(result_host(i) == i);
}
```

Asynchronous parallel_reduce

```
void async_parallel_reduce()
{
    Kokkos::View<int*, Kokkos::HostSpace> result_host("result_host", N);
    Kokkos::View<int*, Kokkos::CudaSpace> result("result", N);
    for (std::size_t i = 0; i != N; ++i) result_host(i) = i;

    hpx::future<void> f1 = hpx::kokkos::deep_copy_async(inst, result, result_host);

    int sum = 0;
    hpx::future<void> f2 = hpx::kokkos::parallel_reduce_async(
        Kokkos::RangePolicy<Kokkos::CudaSpace>(inst, 0, N),
        KOKKOS_LAMBDA(int const& i, int& acc) { acc += result(i); },
        Kokkos::Sum<int>(sum));

    hpx::future<void> f3 = hpx::kokkos::deep_copy_async(inst, result, result_host);

    hpx::wait_all(f1, f2, f3);
    assert(sum == N * (N - 1) / 2);
}
```

HPX Executors for Kokkos

- HPX executors that correspond to Kokkos execution spaces

```
namespace hpx::kokkos {

    // The following are always defined
    class default_executor;
    class default_host_executor;

    // The following are conditionally defined
    class cuda_executor;
    class hip_executor;
    class sycl_executor;
    class hpx_executor;
    class openmp_executor;
    class serial_executor;
}
```

Execution Policies for Kokkos

- The following execution policy can be used with HPX parallel algorithms
  - It uses the default Kokkos host execution space
  - Unless customized with `.on()`

```
namespace hpx::kokkos {

    constexpr kokkos_policy kok;
}
```

HPX Algorithms

```
void hpx_for_each()
{
    Kokkos::View<int*, Kokkos::HostSpace> data_host("data_host", N);
    Kokkos::View<int*, Kokkos::CudaSpace> data_cuda("data_cuda", N);
    for (std::size_t i = 0; i != N; ++i) data_host(i) = i;

    Kokkos::deep_copy(data_cuda, data_host);

    hpx::kokkos::cuda_executor exec;
    hpx::for_each(
        hpx::kokkos::kok
            .on(exec)
            .label("for_each_sync"),
        data_cuda.data(), data_cuda.data() + data_cuda.size(),
        KOKKOS_LAMBDA(int& x) { x *= 2; });

    Kokkos::deep_copy(data_host, cuda_data);

    for (std::size_t i = 0; i != N; ++i) assert(data_host(i) == 2 * i);
}
```

Asynchronous HPX Algorithms

```
void hpx_async_for_each()
{
    Kokkos::View<int*, Kokkos::HostSpace> data_host("data_host", N);
    Kokkos::View<int*, Kokkos::CudaSpace> data_cuda("data_cuda", N);
    for (std::size_t i = 0; i != N; ++i) data_host(i) = i;

    Kokkos::deep_copy(data_cuda, data_host);

    hpx::kokkos::cuda_executor exec;
    hpx::future<void> f = hpx::for_each(
        hpx::kokkos::kok(hpx::execution::task)
            .on(exec)
            .label("for_each_task"),
        data_cuda.data(), data_cuda.data() + data_cuda.size(),
        KOKKOS_LAMBDA(int& x) { x *= 3; });

    hpx::wait_all(f);

    Kokkos::deep_copy(data_host, data_cuda);

    for (std::size_t i = 0; i < N; ++i) assert(data_host(i) == 3 * i);
}
```

Multi-dimensional Arrays

```
void hpx_for_each_mdrange() {
    Kokkos::View<int**, Kokkos::HostSpace> data_host("data_host", N, M);
    Kokkos::View<int**, Kokkos::CudaSpace> data_cuda(
        "data_cuda", Kokkos::CudaSpace::array_layout, N, M);
    for (std::size_t i = 0; i != N; ++i)
        for (std::size_t j = 0; j != M; ++j)
            data_host(i, j) = 0;

    Kokkos::deep_copy(data_cuda, data_host);

    hpx::kokkos::cuda_executor exec;
    hpx::ranges::for_each(
        hpx::kokkos::kok.on(exec).label("for_each task mdrange"),
        Kokkos::MDRangePolicy<Kokkos::Rank<2>>({0, 0}, {N, M}),
        KOKKOS_LAMBDA(int i, int j) { data_cuda(i, j) = i + j; });

    Kokkos::deep_copy(data_host, data_cuda);

    for (std::size_t i = 0; i < n; ++i)
        for (std::size_t j = 0; j < m; ++j)
            assert(data_host(i, j) == i + j);
}
```

Conclusions

- Kokkos is a abstraction library that
  - Hides accelerator architectures and idiosyncrasies
  - Provides portable access to wide variety of 'backends' from sequential CPU based execution (debugging) to CUDA, HIP, SYCL, and HPX
- HPX parallel algorithms are very versatile and not limited to standard C++ parallel execution
  - Usable for accelerator parallelization
  - Usable for asynchronous operation

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP