Location of Materials and Exercises

**Tutorial Materials:**
- clone `github.com/kokkos/kokkos-tutorials` into `{HOME}/kokkos-tutorials`
  - Slides are in `{HOME}/kokkos-tutorials/Intro-Short/Slides`
  - Exercises are in `{HOME}/kokkos-tutorials/Intro-Short/Exercises`
  
  *Exercises’ Makefiles look for `{HOME}/kokkos* - suggest clone to `{HOME}`*

- Advanced Tutorial and Exercises:
  - Slides are in `{HOME}/kokkos-tutorials/Intro-Full/Slides`
  - Additional exercises are in `{HOME}/kokkos-tutorials/Intro-Full/Exercises`

- Online Programming Guide, API Reference, Compilation Options - See the Wiki:
  `github.com/kokkos/kokkos/wiki`

**Library Repos and Requirements:**
- Git
- GCC 4.8.4 (or newer) OR Intel 15 (or newer) OR Clang 3.5.2 (or newer)
- CUDA nvcc 7.5 (or newer) AND NVIDIA compute capability 3.0 (or newer)
- clone `github.com/kokkos/kokkos` into `{HOME}/kokkos`
- clone `github.com/kokkos/kokkos-tools` into `{HOME}/kokkos-tools`
- clone `github.com/kokkos/kokkos-kernels` into `{HOME}/kokkos-kernels`
What is *Kokkos* and how does it address performance portability?

*Kokkos* is a *productive, portable, performant*, shared-memory programming model.

- is a C++ *library*, not a new language or language extension.
- supports *clear, concise, thread-scalable* parallel patterns.
- lets you write algorithms once and run on *many architectures* e.g. multi-core CPU, NVidia GPU, Xeon Phi, ...
- *minimizes* the amount of architecture-specific *implementation details* users must know.
- *solves the data layout problem* by using multi-dimensional arrays with architecture-dependent *layouts*
**Kokkos** is a *productive, portable, performant*, shared-memory programming model.

▶ is a C++ **library**, not a new language or language extension.
▶ supports **clear, concise, thread-scalable** parallel patterns.
▶ lets you write algorithms once and run on **many architectures**
  e.g. multi-core CPU, NVidia GPU, Xeon Phi, ...
▶ **minimizes** the amount of architecture-specific
  **implementation details** users must know.
▶ **solves the data layout problem** by using multi-dimensional
  arrays with architecture-dependent **layouts**

---

**Important Point**

For performance the memory access pattern
**must** depend on the architecture.
The Kokkos Ecosystem

- **MiniApps**
- **Applications**
  - Algorithms (Random, Sort)
  - Containers (Map, CrsGraph, Mem Pool)
  - Kokkos Core (Parallel Execution, Data Allocation, Data Transfer)
- **Trilinos** (Linear Solvers, Load Balancing, Discretization, Distributed Linear Algebra)
- **Kokkos – Kernels** (Sparse/Dense BLAS, Graph Kernels, Tensor Kernels)
- **Kokkos – Tools** (Kokkos aware Profiling and Debugging Tools)
- **Kokkos – Support Community** (Application Support, Developer Training)

Supported by:
- std::thread
- OpenMP
- CUDA
- ROCm
Kokkos Abstractions

- Data Structures
  - Memory Spaces ("Where")
    - HBM, DDR, Non-Volatile, Scratch
  - Memory Layouts
    - Row/Column-Major, Tiled, Strided
  - Memory Traits ("How")
    - Streaming, Atomic, Restrict

- Parallel Execution
  - Execution Spaces ("Where")
    - CPU, GPU, Executor Mechanism
  - Execution Patterns
    - parallel_for/reduce(scan), task-spawn
  - Execution Policies ("How")
    - Range, Team, Task-Graph
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        total += dot(left[element][qp], right[element][qp]);
    }
    elementValues[element] = total;
}
Concepts: Patterns, Policies, and Bodies

Pattern

\[
\text{for} \ (\text{element} = 0; \ \text{element} < \text{numElements}; \ \text{++element}) \ \{
\text{total} = 0;
\text{for} \ (\text{qp} = 0; \ \text{qp} < \text{numQPs}; \ \text{++qp}) \ \{
\text{total} += \text{dot}(\text{left}[\text{element}][\text{qp}], \ \text{right}[\text{element}][\text{qp}]);
\}
\text{elementValues}[\text{element}] = \text{total};
\}
\]

Terminology:

- **Pattern**: structure of the computations
  
  for, reduction, scan, task-graph, ...

- **Execution Policy**: how computations are executed
  static scheduling, dynamic scheduling, thread teams, ...

- **Computational Body**: code which performs each unit of work; e.g., the loop body

⇒ The **pattern** and **policy** drive the computational **body**.
<table>
<thead>
<tr>
<th>Concept</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Loops</td>
<td><code>parallel_for(N, KOKKOS_LAMBDA(int i) { ...BODY... });</code></td>
</tr>
</tbody>
</table>
| Parallel Reduction    | `parallel_reduce(RangePolicy<ExecSpace>(0,N), KOKKOS_LAMBDA(int i, double& upd) { ...
|                       |   BODY...                |
|                       |   upd += ...            |
|                       | }, result);             |
| Tightly Nested Loops  | `parallel_for(MDRangePolicy<Rank<3>>({0,0,0},{N1,N2,N3},{T1,T2,T3},
|                       |   KOKKOS_LAMBDA(int i, int j, int k) { ...BODY... });`             |
| Non-Tightly Nested Loops | `parallel_for(TeamPolicy<Schedule<Dynamic>>(N, TS), KOKKOS_LAMBDA(Team team) {
|                       |   ... COMMON CODE 1 ...
|                       |   parallel_for(TeamThreadRange(team, M(N)), [&] (int j) { ... INNER BODY... });
|                       |   ... COMMON CODE 2 ...
|                       | });`                                                                     |
| Task Dag              | `task_spawn(TaskTeam(scheduler, priority), KOKKOS_LAMBDA(Team team) { ...
|                       |   BODY });`                                                           |
| Data Allocation       | `View<double**, Layout, MemSpace> a("A", N, M);`                      |
| Data Transfer         | `deep_copy(a, b);`                                                     |
| Exec Spaces           | `Serial, Threads, OpenMP, Cuda, ROCm (experimental)`                   |
Prerequisite Knowledge of C++: class ctors, member variables, member functions, member operators, template arguments

Kokkos’ basic capabilities - today’s objectives:
- Simple 1D data parallel computational patterns
- Deciding where code is run and where data is placed
- Managing data access patterns for performance portability

Kokkos’ advanced capabilities not covered today:
- Thread safety, thread scalability, and atomic operations
- Hierarchical patterns for maximizing parallelism
- Multidimensional data parallelism
- Dynamic directed acyclic graph of tasks pattern
- Numerous plug-in points for extensibility
Data parallel patterns

Learning objectives:

- How computational bodies are passed to the Kokkos runtime.
- How work is mapped to cores.
- The difference between `parallel_for` and `parallel_reduce`.
- Start parallelizing a simple example.
Data parallel patterns and work

for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}

Kokkos maps \textbf{work} to cores
Using Kokkos for data parallel patterns (0)

Data parallel patterns and work

```cpp
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

Kokkos maps **work** to cores

- 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.
Data parallel patterns and work

```c
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

Kokkos maps **work** to cores

- 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**

You give an **iteration range** and **computational body** (kernel) to Kokkos, Kokkos maps iteration indices to cores and then runs the computational body on those cores.
Data parallel patterns and work

```c
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

Kokkos maps **work** to cores

- 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.

Preview of Kokkos::parallel_for API:

```
parallel_for (numberOfAtoms, ...);
```
How are computational bodies given to Kokkos?
How are computational bodies given to Kokkos?

As **functors** or *function objects*, a common pattern in C++. 
How are computational bodies given to Kokkos?

As **functors** or **function objects**, a common pattern in C++. 

Quick review, a **functor** is a function with data. Example:

```cpp
struct ParallelFunctor {
    ...
    void operator()( a work assignment ) const {
        /* ... computational body ... */
        ...
    }
};
```
Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern,

```cpp
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

Warning: concurrency and order

Concurrency and ordering of parallel iterations is not guaranteed by the Kokkos runtime.
How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern,

```
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

and work items are assigned to functors one-by-one:

```
struct ParallelFunctor {
  void operator()(const size_t index) const {...}
}
```
How is work assigned to functor operators?

A total amount of work items is given to a Kokkos pattern,

```cpp
ParallelFunctor functor;
Kokkos::parallel_for(numberOfIterations, functor);
```

and work items are assigned to functors one-by-one:

```cpp
struct ParallelFunctor {
  void operator()(const size_t index) const {...}
}
```

**Warning: concurrency and order**

Concurrency and ordering of parallel iterations is *not* guaranteed by the Kokkos runtime.
How is data passed to computational bodies?

```c
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

```c
struct AtomForceFunctor {

    ...;

    void operator()(const size_t atomIndex) const {
        atomForces[atomIndex] = calculateForce(...data...);
    }

    ...
}
```
How is data passed to computational bodies?

```c
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(...data...);
}
```

```c
struct AtomForceFunctor {
    ...
    void operator()(const size_t atomIndex) const {
        atomForces[atomIndex] = calculateForce(...data...);
    }
    ...
}
```

How does the body access the data?

**Important concept**

A parallel functor body must have access to all the data it needs through the functor’s **data members**.
Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    AtomDataType _atomData;

    AtomForceFunctor(_atomForces, _atomData) {...}

    void operator()(const size_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
}
```

How would we reproduce serial execution with this functor?

```cpp
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    atomForces[atomIndex] = calculateForce(data);
}
AtomForceFunctor functor(atomForces, data);
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {
    functor(atomIndex);
}
```
Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    AtomDataType _atomData;
    AtomForceFunctor(_atomForces, _atomData) {...}
    void operator()(const size_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
}
```

Q/ How would we reproduce serial execution with this functor?

```cpp
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
    atomForces[atomIndex] = calculateForce(data);
}
```
Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    AtomDataType _atomData;
    AtomForceFunctor(_atomForces, _atomData) {...}
    void operator()(const size_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
}
```

Q/ How would we reproduce serial execution with this functor?

Serial

```cpp
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
    atomForces[atomIndex] = calculateForce(data);
}
```

Functor

```cpp
AtomForceFunctor functor(atomForces, data);
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
    functor(atomIndex);
}
```
The complete picture (using functors):

1. Defining the functor (operator+data):

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    AtomDataType _atomData;

    AtomForceFunctor(atomForces, data) : 
        _atomForces(atomForces), _atomData(data) {}

    void operator()(const size_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
}
```

2. Executing in parallel with Kokkos pattern:

```cpp
AtomForceFunctor functor(atomForces, data);
Kokkos::parallel_for(numberOfAtoms, functor);
```
Functors are tedious $\Rightarrow$ **C++11 Lambdas** are concise

```cpp
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
    [=] (const size_t atomIndex) {
        atomForces[atomIndex] = calculateForce(data);
    });
```

A lambda is not *magic*, it is the compiler **auto-generating** a **functor** for you.
Functors are tedious ⇒ **C++11 Lambdas** are concise

```cpp
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
            [=] (const size_t atomIndex) {
                atomForces[atomIndex] = calculateForce(data);
            });
```

A lambda is not *magic*, it is the compiler **auto-generating** a **functor** 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.
How does this compare to OpenMP?

Serial

```c
for (size_t i = 0; i < N; ++i) {
    /* loop body */
}
```

OpenMP

```c
#pragma omp parallel for
for (size_t i = 0; i < N; ++i) {
    /* loop body */
}
```

Kokkos

```c
parallel_for(N, [=] (const size_t i) {
    /* loop body */
});
```

Important concept

Simple Kokkos usage is **no more conceptually difficult** than OpenMP, the annotations just go in different places.
Riemann-sum-style numerical integration:

\[ y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) \, dx \]
Riemann-sum-style numerical integration:

\[ y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) \, dx \]

double totalIntegral = 0;
for (size_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
Riemann-sum-style numerical integration:

\[ y = \int_{lower}^{upper} \text{function}(x) \, dx \]

double totalIntegral = 0;
for (size_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;

How do we parallelize it? Correctly?
Scalar integration (0)

Riemann-sum-style numerical integration:

\[ y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) \, dx \]

Pattern?

```cpp
double totalIntegral = 0;  // Policy?
for (size_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
```

Body?

```
totalIntegral *= dx;
```

How do we parallelize it? Correctly?
An (incorrect) attempt:

```cpp
double totalIntegral = 0;
Kokkos::parallel_for(numberOfIntervals,
        [=] (const size_t index) {
            const double x =
                lower + (index/numberOfIntervals) * (upper - lower);
            totalIntegral += function(x);},
        );
totalIntegral *= dx;
```

First problem: compiler error; cannot increment `totalIntegral` (lambdas capture by value and are treated as const!)
Scalar integration (2)

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

```c
double totalIntegral = 0;
double * totalIntegralPointer = &totalIntegral;
Kokkos::parallel_for(numberOfIntervals,
                     [=] (const size_t index) {
                       const double x =
                           lower + (index/numberOfIntervals) * (upper - lower);
                       *totalIntegralPointer += function(x);},
                   );
totalIntegral *= dx;
```

Second problem: race condition
An (incorrect) solution to the (incorrect) attempt:

```cpp
double totalIntegral = 0;
double * totalIntegralPointer = &totalIntegral;
Kokkos::parallel_for(numberOfIntervals, [=] (const size_t index) {
    const double x = lower + (index/numberOfIntervals) * (upper - lower);
    *totalIntegralPointer += function(x);
},
);   
totalIntegral *= dx;
```

Second problem: race condition

<table>
<thead>
<tr>
<th>step</th>
<th>thread 0</th>
<th>thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>increment</td>
<td>load</td>
</tr>
<tr>
<td>2</td>
<td>write</td>
<td>increment</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>write</td>
</tr>
</tbody>
</table>
Root problem: we’re using the *wrong pattern*, *for* instead of *reduction*
Root problem: we’re using the **wrong pattern**, *for* instead of *reduction*

**Important concept: Reduction**

Reductions combine the results contributed by parallel work.
Root problem: we’re using the wrong pattern, for instead of reduction

Important concept: Reduction

Reductions combine the results contributed by parallel work.

How would we do this with OpenMP?

double finalReducedValue = 0;
#pragma omp parallel for reduction(+:finalReducedValue)
for (size_t i = 0; i < N; ++i) {
    finalReducedValue += ...
}

How will we do this with Kokkos?
Scalar integration (3)

**Root problem:** we’re using the **wrong pattern**, *for* instead of *reduction*

**Important concept: Reduction**

Reductions combine the results contributed by parallel work.

How would we do this with **OpenMP**?

```c
double finalReducedValue = 0;
#pragma omp parallel for reduction(+:finalReducedValue)
for (size_t i = 0; i < N; ++i) {
    finalReducedValue += ...
}
```

How will we do this with **Kokkos**?

```c
double finalReducedValue = 0;
parallel_reduce(N, functor, finalReducedValue);
```
Scalar integration (4)

Comparison using lambda

```plaintext
double totalIntegral = 0;
#pragma omp parallel for reduction(+:totalIntegral)
for (size_t i = 0; i < numberOfIntervals; ++i) {
    totalIntegral += function(...);
}
```

```plaintext
double totalIntegral = 0;
parallel_reduce(numberOfIntervals, 
    [=] (const size_t i, double & valueToUpdate) {
        valueToUpdate += function(...);
    },
    totalIntegral);
```

- 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.
Basic capabilities we haven’t covered

- Customizing `parallel_reduce` data type and reduction operator
  
  e.g., minimum, maximum, ...

- `parallel_scan` pattern for exclusive and inclusive prefix sum

- Using `tag dispatch` interface to allow non-trivial functors to have multiple “operator()” functions.
  
  very useful in large, complex applications
Views

Learning objectives:
▶ Motivation behind the View abstraction.
▶ Key View concepts and template parameters.
▶ The View life cycle.
Example: running daxpy on the GPU:

```c
double * x = new double[N]; // also y
parallel_for(N, [=] (const size_t i) {
    y[i] = a * x[i] + y[i];
});
```

```c
struct Functor {
    double * _x, * _y, a;
    void operator()(const size_t 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) which can be communicated to an accelerator (GPU).

⇒ Views"
Example: running daxpy on the GPU:

```c++
double *x = new double[N]; // also y
parallel_for(N, [=] (const size_t i) {
    y[i] = a * x[i] + y[i];
});

struct Functor {
    double *_x, *_y, a;
    void operator()(const size_t i) {
        _y[i] = _a * _x[i] + _y[i];
    }
};
```

**Problem:** x and y reside in CPU memory.
**Example: running daxpy on the GPU:**

```c
double * x = new double[N]; // also y
parallel_for(N, [=] (const size_t i) {
    y[i] = a * x[i] + y[i];
});
```

```c
struct Functor {
    double * _x, * _y, a;
    void operator()(const size_t 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) which can be communicated to an accelerator (GPU).

⇒ **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:

```cpp
View<double*, ...> x(...), y(...);
... populate x, y...

parallel_for(N, [=] (const size_t i) {
    // Views x and y are captured by value (copy)
    y(i) = a * x(i) + y(i);
});
```
**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:

```cpp
View<double*, ...> x(...), y(...);
... populate x, y...

parallel_for(N, [=] (const size_t i) {
  // Views x and y are captured by value (copy)
  y(i) = a * x(i) + y(i);
});
```

**Important point**

Views are **like pointers**, so copy them in your functors.
**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.
**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.

**Example:**

View<`double`***> data("label", N0, N1, N2); 3 run, 0 compile
View<`double`**[N2]> data("label", N0, N1);  2 run, 1 compile
View<`double`*[N1][N2]> data("label", N0); 1 run, 2 compile
View<`double`[N0][N1][N2]> data("label"); 0 run, 3 compile

Note: runtime-sized dimensions must come first.
**View** life cycle:

- 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 `shared_ptr`

Example:

```cpp
View < double *> a("a", N0), b("b", N0);
a = b;
View < double *> c(b);
a(0) = 1;
b(0) = 2;
c(0) = 3;
print a(0)
```

What gets printed? 3.0
**View life cycle:**

- 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 `shared_ptr`

**Example:**

```cpp
View<double*> a("a", N0), b("b", N0);
a = b;
View<double*> c(b);
a(0) = 1;
b(0) = 2;
c(0) = 3;
print a(0)
```

What gets printed?
**View life cycle:**

- 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 *shared_ptr*

**Example:**

```cpp
View<double*> a("a", N0), b("b", N0);
a = b;
View<double*> c(b);
a(0) = 1;
b(0) = 2;
c(0) = 3;
print a(0)
```

What gets printed? 3.0
Execution and Memory Spaces

Learning objectives:

▶ Heterogeneous nodes and the space abstractions.
▶ How to control where parallel bodies are run, execution space.
▶ How to control where view data resides, memory space.
▶ How to avoid illegal memory accesses and manage data movement.
▶ The need for Kokkos::initialize and finalize.
▶ Where to use Kokkos annotation macros for portability.
Thought experiment: Consider this code:

```c
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);
Kokkos::parallel_for(numberOfSomethings,
    [=] (const size_t somethingIndex) {
        const double y = ...;
        // do something interesting
    }
    );
```

▶ Where will section 1 be run? CPU? GPU?
▶ Where will section 2 be run? CPU? GPU?
▶ How do I control where code is executed?
Thought experiment: Consider this code:

```cpp
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(... data ...);
Kokkos::parallel_for(numberOfSomethings,
        [=] (const size_t somethingIndex) {
            const double y = ...;
            // do something interesting
        });
```

- Where will section 1 be run? CPU? GPU?
- Where will section 2 be run? CPU? GPU?
- How do I control where code is executed?
Thought experiment: Consider this code:

```cpp
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);
Kokkos::parallel_for(numberOfSomethings,
  [=] (const size_t somethingIndex) {
    const double y = ...;
    // do something interesting
  });
```

- Where will section 1 be run? CPU? GPU?
- Where will section 2 be run? CPU? GPU?
- How do I control where code is executed?

⇒ Execution spaces
Execution Space

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

Execution spaces: Serial, Threads, OpenMP, Cuda, ROCm, ...
Where will Host code be run? CPU? GPU?
⇒ Always in the host process

Where will Parallel code be run? CPU? GPU?
⇒ The default execution space

How do I control where the Parallel body is executed?
Changing the default execution space (at compilation), or specifying an execution space in the policy.

```cpp
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);
Kokkos::parallel_for(numberOfSomethings,
    [=] (const size_t somethingIndex) {
        const double y = ...;
        // do something interesting
    }
);
```
Execution spaces (2)

Where will Host code be run? CPU? GPU?
⇒ Always in the host process
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(... data ...);
Kokkos::parallel_for(numberOfSomeThings,
  [=] (const size_t somethingIndex) {
    const double y = ...;
    // do something interesting
  });

Where will Host code be run? CPU? GPU?
⇒ Always in the host process

Where will Parallel code be run? CPU? GPU?
⇒ The default execution space
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(... data ...);

Kokkos::parallel_for(numberOfSomethings,
    [=] (const size_t somethingIndex) {
        const double y = ...;
        // do something interesting
    }
);
Changing the parallel execution space:

```
parallel_for(
    RangePolicy< ExecutionSpace >(0, numberOfIntervals),
    [=] (const size_t i) {
        /* ... body ... */
    });
```

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.
Changing the parallel execution space:

```
parallel_for(
    RangePolicy< ExecutionSpace >((0, numberOfIntervals),
    [=] (const size_t i) {
        /* ... body ... */
    });
```

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.
Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```cpp
struct ParallelFunctor {
  KOKKOS_INLINE_FUNCTION
  double helperFunction(const size_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const size_t index) const {
    helperFunction(index);
  }
}

// Where kokkos defines:
#define KOKKOS_INLINE_FUNCTION inline /* #if CPU-only */
#define KOKKOS_INLINE_FUNCTION inline __device__ __host__ /* #if CPU+Cuda */
```
Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```cpp
struct ParallelFunctor {
    KOKKOS_INLINE_FUNCTION
double helperFunction(const size_t s) const {...}
    KOKKOS_INLINE_FUNCTION
    void operator()(const size_t index) const {
        helperFunction(index);
    }
}

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

Lambda annotation with KOKKOS_LAMBDA macro (requires CUDA 8.0)

```cpp
Kokkos::parallel_for(numberOfIterations, 
    KOKKOS_LAMBDA (const size_t index) {...});

// Where kokkos defines:
#define KOKKOS_LAMBDA [=] /* #if CPU-only */
#define KOKKOS_LAMBDA [=] __device__ /* #if CPU+Cuda */
```
Memory space motivating example: summing an array

```cpp
View<double*> data("data", size);
for (size_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy<SomeExampleExecutionSpace>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```
Memory space motivating example: summing an array

View< double* > data("data", size);
for ( size_t i = 0; i < size; ++i ) {
    data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< SomeExampleExecutionSpace >(0, size),
    KOKKOS_LAMBDA ( const size_t index, double & valueToUpdate ) {
        valueToUpdate += data(index);
    },
    sum);

Question: Where is the data stored? GPU memory? CPU memory? Both?
Memory space motivating example: summing an array

View<double*> data("data", size);
for (size_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy<SomeExampleExecutionSpace>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);

Question: Where is the data stored? GPU memory? CPU memory? Both?
Memory space motivating example: summing an array

```cpp
View<double*> data("data", size);
for (size_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy<SomeExampleExecutionSpace>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

Question: Where is the data stored? GPU memory? CPU memory? Both?

⇒ Memory Spaces
Memory space:
explicitly-manageable memory resource
(i.e., “place to put data”)
Important concept: Memory spaces

Every view stores its data in a **memory space** set at compile time.
Important concept: Memory spaces

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

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

Available memory spaces:
- HostSpace, CudaSpace, CudaUVMSpace, ...

Each execution space has a default memory space, which is used if a space is provided. If no space is provided, the view's data resides in the default memory space of the default execution space.
Important concept: Memory spaces

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

- View\(<\text{double***}, \text{MemorySpace}\>\) data(...);
- Available **memory spaces**: HostSpace, CudaSpace, CudaUVMSpace, ... more
Important concept: Memory spaces

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

- `View<double***, MemorySpace> data(...);`
- Available **memory spaces**:
  - HostSpace, CudaSpace, CudaUVMSpace, ... more
- Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space
Important concept: Memory spaces

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

- `View<double***, MemorySpace> data(...);`
- Available **memory spaces**:
  - HostSpace, CudaSpace, CudaUVMSpace, ... more
- Each **execution space** has a default memory space, which is used if `Space` provided is actually an execution space
- If no `Space` is provided, the view’s data resides in the **default memory space** of the **default execution space**.
Example: HostSpace

View<double**, HostSpace> hostView(...constructor arguments...);
Example: HostSpace

View<\emph{double**}, \emph{HostSpace}> hostView(...constructor arguments...);

Example: CudaSpace

View<\emph{double**}, \emph{CudaSpace}> view(...constructor arguments...);
Anatomy of a kernel launch:

1. User declares views, allocate data.
2. User instantiates a functor with views.
3. User launches `parallel_something`:
   - Functor copied to the device.
   - Kernel is run.
   - Copy of functor on device released.

Note: **no deep copies** of array data are performed; *views are like pointers.*
Example: one view

```cpp
View<int*, Cuda> dev;
parallel_for(N, KOKKOS_LAMBDA (int i)
    dev(i) = ...;
});
```
Example: two views

View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for(N, KOKKOS_LAMBDA (int i) {
    dev(i) = ...;
    host(i) = ...;
});
Example: two views

View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for(N, KOKKOS_LAMBDA (int i) {
  dev(i) = ...;
  host(i) = ...;
});
Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

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

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy<Cuda>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += array(index);
    },
    sum);
```
Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

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

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< Cuda >(0, size),
    KOKKOS_LAMBD A (const size_t index, double & valueToUpdate) {
        valueToUpdate += array(index);
    },
    sum);
```
Execution and Memory spaces (4)

Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

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

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBD A (const size_t index, double & valueToUpdate) {
        valueToUpdate += array(index);
    },
    sum);
```
Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

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

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += array(index);  // illegal access
    },
    sum);
```
Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

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

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< Cuda >(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += array(index);  // illegal access
    },
    sum);
```

What’s the solution?

- CudaUVMMSpace
- CudaHostPinnedSpace (skipping)
- Mirroring
CudaUVMSpace

```c
View<double*, CudaUVMSpace> array
array = ...from file...
double sum = 0;
parallel_reduce(N,
    KOKKOS_LAMBDA (int i, double & d) {
        d += array(i);
    },
    sum);
```

Cuda runtime automatically handles data movement, at a performance hit.
Important concept: Mirrors

Mirrors are views of equivalent arrays residing in possibly different memory spaces.
Important concept: Mirrors

Mirrors are views of equivalent arrays residing in possibly different memory spaces.

Mirroring schematic

typedef Kokkos::View<double**, Space> ViewType;
ViewType view(...);
ViewType::HostMirror hostView =
   Kokkos::create_mirror_view(view);
1. **Create** a `view`'s array in some memory space.
   ```cpp
typedef Kokkos::View<double*, Space> viewType;
ViewType view(...);
```
1. Create a `view`'s array in some memory space.

   ```
   typedef Kokkos::View<double*, Space> ViewType;
   ViewType view(...);
   ```

2. Create `hostView`, a *mirror* of the `view`'s array residing in the host memory space.

   ```
   ViewType::HostMirror hostView = Kokkos::create_mirror_view(view);
   ```
1. **Create** a `view`'s array in some memory space.
   
   ```cpp
   typedef Kokkos::View<double*, Space> ViewType;
   ViewType view(...);
   ```

2. **Create** `hostView`, a *mirror* of the `view`'s array residing in the host memory space.
   
   ```cpp
   ViewType::HostMirror hostView = Kokkos::create_mirror_view(view);
   ```

3. **Populate** `hostView` on the host (from file, etc.).
Mirroring pattern

1. Create a view’s array in some memory space.
   
   ```cpp
   typedef Kokkos::View<double*, Space> ViewType;
   ViewType view(...);
   ```

2. Create hostView, a mirror of the view’s array residing in the host memory space.

   ```cpp
   ViewType::HostMirror hostView = Kokkos::create_mirror(view);
   ```

3. Populate hostView on the host (from file, etc.).

4. Deep copy hostView’s array to view’s array.

   ```cpp
   Kokkos::deep_copy(view, hostView);
   ```
1. **Create** a `view`'s array in some memory space.
   
   ```c++
   typedef Kokkos::View<double*, Space> ViewType;
   ViewType view(...);
   ```

2. **Create** `hostView`, a *mirror* of the `view`'s array residing in the host memory space.
   
   ```c++
   ViewType::HostMirror hostView =
   Kokkos::create_mirror_view(view);
   ```

3. **Populate** `hostView` on the host (from file, etc.).

4. **Deep copy** `hostView`'s array to `view`'s array.
   
   ```c++
   Kokkos::deep_copy(view, hostView);
   ```

5. **Launch** a kernel processing the `view`'s array.
   
   ```c++
   Kokkos::parallel_for(
       RangePolicy< Space>(0, size),
       KOKKOS_LAMBDA (...) { use and change view });
   ```
1. **Create** a `view`'s array in some memory space.
   
   ```cpp
   typedef Kokkos::View<double*, Space> ViewType;
   ViewType view(...);
   ```

2. **Create** `hostView`, a *mirror* of the `view`'s array residing in the host memory space.
   
   ```cpp
   ViewType::HostMirror hostView = Kokkos::create_mirror(view);
   ```

3. **Populate** `hostView` on the host (from file, etc.).

4. **Deep copy** `hostView`'s array to `view`'s array.
   
   ```cpp
   Kokkos::deep_copy(view, hostView);
   ```

5. **Launch** a kernel processing the `view`'s array.
   
   ```cpp
   Kokkos::parallel_for(
       RangePolicy<Space>(0, size),
       KOKKOS_LAMBDAB (...){ use and change view });
   ```

6. If needed, **deep copy** the `view`'s updated array back to the `hostView`'s array to write file, etc.
   
   ```cpp
   Kokkos::deep_copy(hostView, view);
   ```
What if the View is in HostSpace too? Does it make a copy?

```cpp
typedef Kokkos::View<double*, Space> ViewType;
ViewType view("test", 10);
ViewType::HostMirror hostView =
    Kokkos::create_mirror_view(view);
```

- `create_mirror_view` allocates data only if the host process cannot access `view`’s data, otherwise `hostView` references the same data.
- `create_mirror` always allocates data.
- Reminder: Kokkos never performs a hidden deep copy.
Recurring Exercise: Inner Product

Exercise: Inner product $<y, Ax>$

Details:
- $y$ is $N\times1$, $A$ is $N\times M$, $x$ is $M\times1$
- We’ll use this exercise throughout the tutorial
- Optional: Try Exercises 1-4 during break or evening hands-on session
The **first step** in using Kokkos is to include, initialize, and finalize:

```cpp
#include <Kokkos_Core.hpp>
int main(int argc, char** argv) {
  /* ... do any necessary setup (e.g., initialize MPI) ... */
  Kokkos::initialize(argc, argv);
  {
    /* ... do computations ... */
  }
  Kokkos::finalize();
  return 0;
}
```

(Optional) Command-line arguments:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--kokkos-threads=INT</td>
<td>total number of threads (or threads within NUMA region)</td>
</tr>
<tr>
<td>--kokkos-numa=INT</td>
<td>number of NUMA regions</td>
</tr>
<tr>
<td>--kokkos-device=INT</td>
<td>device (GPU) ID to use</td>
</tr>
</tbody>
</table>
Exercise #1: Inner Product, Flat Parallelism on the CPU

Exercise: Inner product $\langle y, A \ast x \rangle$

Details:
- Location: Intro-Short/Exercises/01/Begin/
- Look for comments labeled with “EXERCISE”
- Need to include, initialize, and finalize Kokkos library
- Parallelize loops with `parallel_for` or `parallel_reduce`
- Use lambdas instead of functors for computational bodies.
- For now, this will only use the CPU.
Compiling for CPU

# gcc using OpenMP (default) and Serial back-ends,
# (optional) non-default arch set with KOKKOS_ARCH
make -j KOKKOS_DEVICES=OpenMP,Serial KOKKOS_ARCH=SNB
# KOKKOS_ARCH Options: See the wiki at
# https://github.com/kokkos/kokkos/wiki/Compiling

Running on CPU with OpenMP back-end

# Set OpenMP affinity
export OMP_NUM_THREADS=8
export OMP_PROC_BIND=spread OMP_PLACES=threads
# Print example command line options:
./01_Exercise.host -h
# Run with defaults on CPU
./01_Exercise.host
# Run larger problem
./01_Exercise.host -S 26

Things to try:

- Vary number of threads
- Vary problem size (-S ...), Vary number of rows (-N ...
Managing memory access patterns for performance portability

Learning objectives:

▶ 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
▶ Why memory access patterns and layouts have such a performance impact (caching and coalescing).
▶ See a concrete example of the performance of various memory configurations.
double result = 0.0;
for (size_t row = 0; row < N; ++row) {
    double thisRowsSum = 0;
    for (size_t entry = 0; entry < M; ++entry) {
        thisRowsSum += A(row, entry) * x(entry);
    }
    result += y(row) * thisRowsSum;
}

Kokkos::parallel_reduce(
    RangePolicy<ExecutionSpace>(0, N),
    KOKKOS_LAMBDA(const size_t row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (size_t entry = 0; entry < M; ++entry) {
            thisRowsSum += A(row, entry) * x(entry);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);

Driving question:
How should $A$ be laid out in memory?
Kokkos::parallel_reduce(
    RangePolicy<ExecutionSpace>(0, N),
    KOKKOS_LAMBDA (const size_t row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (size_t entry = 0; entry < M; ++entry) {
            thisRowsSum += A(row, entry) * x(entry);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);
Example: inner product (0)

Kokkos::parallel_reduce(
    RangePolicy<ExecutionSpace>(0, N),
    KOKKOS_LAMBDA (const size_t row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (size_t entry = 0; entry < M; ++entry) {
            thisRowsSum += A(row, entry) * x(entry);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);

Driving question: How should A be laid out in memory?
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 multidimensional array Layout set at compile-time.

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

Every View has a multidimensional array Layout set at compile-time.

```
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, default for that memory space is used.  
  LayoutLeft for CudaSpace, LayoutRight for HostSpace.
- Layouts are extensible: ~50 lines
- Advanced layouts: LayoutStride, LayoutTiled, ...
Exercise 4 Summary:

- Added parallel_reduce and replaced ‘‘N’’ in parallel dispatch with RangePolicy<ExecSpace>
- Replaced raw pointer allocations with Kokkos::View’s for x, y, and A
- Added HostMirror Views and deep copy
- Added MemSpace to all Views and Layout to A
Exercise #4: Inner Product, Flat Parallelism

<y|Ax> Exercise 04 (Layout) Fixed Size

KNL: Xeon Phi 68c  HSW: Dual Xeon Haswell 2x16c  Pascal60: Nvidia GPU

Bandwidth (GB/s) vs. Number of Rows (N)

- HSW Left
- HSW Right
- KNL Left
- KNL Right
- Pascal60 Left
- Pascal60 Right
Thread independence:

```cpp
operator()(const size_t index, double & valueToUpdate) {
    const double d = _data(index);
    valueToUpdate += d;
}
```

Question: once a thread reads d, does it need to wait?

- **CPU** threads are independent.
  i.e., threads may execute at any rate.
- **GPU** threads are synchronized in groups (of 32).
  i.e., threads in groups must execute instructions together.
Thread independence:

```c
operator() (const size_t index, double & valueToUpdate) {
    const double d = _data(index);
    valueToUpdate += d;
}
```

Question: once a thread reads \( d \), does it need to wait?

- **CPU** threads are independent.
  - i.e., threads may execute at any rate.

- **GPU** threads are synchronized in groups (of 32).
  - i.e., threads in groups must execute instructions together.

In particular, all threads in a group (**warp**) must finished their loads before *any* thread can move on.

So, **how many cache lines** must be fetched before threads can move on?
Caching and coalescing (1)

**CPUs**: few (independent) cores with separate caches:

- CPUs: few (independent) cores with separate caches.

  - CPUs: few (independent) cores with separate caches.
  - CPUs: few (independent) cores with separate caches.
  - CPUs: few (independent) cores with separate caches.
  - CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.

- CPUs: few (independent) cores with separate caches.
**CPUs:** few (independent) cores with separate caches:

**GPUs:** many (synchronized) cores with a shared cache:
Important point

For performance, accesses to views in HostSpace must be cached, while access to views in CudaSpace must be coalesced.

**Caching**: if thread $t$’s current access is at position $i$, thread $t$’s next access should be at position $i+1$.

**Coalescing**: if thread $t$’s current access is at position $i$, thread $t+1$’s current access should be at position $i+1$. 
Important point

For performance, accesses to views in HostSpace must be **cached**, while access to views in CudaSpace must be **coalesced**.

**Caching**: if thread \( t \)'s current access is at position \( i \), thread \( t \)'s next access should be at position \( i+1 \).

**Coalescing**: if thread \( t \)'s current access is at position \( i \), thread \( t+1 \)'s current access should be at position \( i+1 \).

**Warning**

Uncoalesced access in CudaSpace **greatly** reduces performance (more than 10X)
Important point

For performance, accesses to views in HostSpace must be **cached**, while access to views in CudaSpace must be **coalesced**.

**Caching**: if thread $t$’s current access is at position $i$, thread $t$’s next access should be at position $i+1$.

**Coalescing**: if thread $t$’s current access is at position $i$, thread $t+1$’s current access should be at position $i+1$.

**Warning**

Uncoalesced access in CudaSpace *greatly* reduces performance (more than 10X)

Note: uncoalesced *read-only, random* access in CudaSpace is okay through Kokkos `const RandomAccess` views (*advanced tutorial*).
Consider the array summation example:

```cpp
View<double*, Space> data("data", size);
...populate data...

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< Space>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

Question: is this cached (for OpenMP) and coalesced (for CUDA)?
Consider the array summation example:

View<
double*, Space> data("data", size);
...populate data...

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< Space>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);

Question: is this cached (for OpenMP) and coalesced (for Cuda)?

Given P threads, **which indices** do we want thread 0 to handle?

- **Contiguous:** 0, 1, 2, ..., N/P
- **Strided:** 0, N/P, 2*N/P, ...

Consider the array summation example:

```cpp
View<double*, Space> data("data", size);
...populate data...

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy<Space>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += data(index);
    },
    sum);
```

Question: is this cached (for OpenMP) and coalesced (for Cuda)?

Given P threads, **which indices** do we want thread 0 to handle?

- **Contiguous:**
  0, 1, 2, ..., N/P
- **CPU**

- **Strided:**
  0, N/P, 2*N/P, ...
- **GPU**

Why?
Mapping indices to cores (1)

Iterating for the execution space:

```cpp
operator()(const size_t index, double & valueToUpdate) {
  const double d = _data(index);
  valueToUpdate += d;
}
```

As users we don’t control how indices are mapped to threads, so how do we achieve good memory access?
Mapping indices to cores (1)

**Iterating for the execution space:**

```cpp
operator()(const size_t index, double & valueToUpdate) {
    const double d = _data(index);
    valueToUpdate += d;
}
```

As users we don’t control how indices are mapped to threads, so how do we achieve good memory access?

**Important point**

Kokkos maps indices to cores in **contiguous chunks** on CPU execution spaces, and **strided** for Cuda.
Rule of Thumb

Kokkos index mapping and default layouts provide efficient access if iteration indices correspond to the first index of array.

Example:

```cpp
View<
double***>, ...> view(...);
...
Kokkos::parallel_for(... ,
    KOKKOS_LAMBDA (const size_t workIndex) {
        ...
        view(..., ..., workIndex ) = ...;
        view(..., workIndex, ...) = ...;
        view(workIndex, ..., ...) = ...;
    });
...
```
Important point

Performant memory access is achieved by Kokkos mapping parallel work indices and multidimensional array layout optimally for the architecture.
Important point

Performant memory access is achieved by Kokkos mapping parallel work indices and multidimensional array layout optimally for the architecture.

Analysis: row-major (LayoutRight)

- **HostSpace**: cached (good)
- **CudaSpace**: uncoalesced (bad)
Important point

Performant memory access is achieved by Kokkos mapping parallel work indices and multidimensional array layout optimally for the architecture.

Analysis: column-major (LayoutLeft)

- HostSpace: uncached (bad)
- CudaSpace: coalesced (good)
Analysis: Kokkos architecture-dependent

View<
double**, ExecutionSpace> A(N, M);
parallel_reduce(RangePolicy< ExecutionSpace>(0, N),
... thisRowsSum += A(j, i) * x(i);

- **HostSpace**: cached (good)
- **CudaSpace**: coalesced (good)
Example: inner product (5)

<y|Ax> Exercise 04 (Layout) Fixed Size

KNL: Xeon Phi 68c  HSW: Dual Xeon Haswell 2x16c  Pascal60: Nvidia GPU

Bandwidth (GB/s) vs. Number of Rows (N)

- Coalesced
- Cached
- Uncached
Kokkos advanced capabilities NOT covered today

- Thread safety, thread scalability, and atomic operations
- Hierarchical parallelism via team policies for thread teams
- Multidimensional range policy for tightly nested loops similar to OpenMP loop collapse
- Directed acyclic graph (DAG) of tasks pattern
  - Dynamic graph of heterogeneous tasks (maximum flexibility)
  - Static graph of homogeneous task (low overhead)
- Portable, thread scalable memory pool
- Plugging in customized multidimensional array data layout e.g., arbitrarily strided, hierarchical tiling
**Exercise:** Inner product $<y, A \ast x>$

**Details:**
- $y$ is $N \times 1$, $A$ is $N \times M$, $x$ is $M \times 1$
- We’ll use this exercise throughout the tutorial
- Optional: Try Exercises 1-4 during break or evening hands-on session
Exercise #1: include, initialize, finalize Kokkos

The **first step** in using Kokkos is to include, initialize, and finalize:

```cpp
#include <Kokkos_Core.hpp>
int main(int argc, char** argv) {
    /* ... do any necessary setup (e.g., initialize MPI) ... */
    Kokkos::initialize(argc, argv);
    {
        /* ... do computations ... */
    }
    Kokkos::finalize();
    return 0;
}
```

(Optional) Command-line arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--kokkos-threads=INT</td>
<td>total number of threads (or threads within NUMA region)</td>
</tr>
<tr>
<td>--kokkos-numa=INT</td>
<td>number of NUMA regions</td>
</tr>
<tr>
<td>--kokkos-device=INT</td>
<td>device (GPU) ID to use</td>
</tr>
</tbody>
</table>
Exercise #1: Inner Product, Flat Parallelism on the CPU

**Exercise:** Inner product $< y, A \times x >$

**Details:**
- Location: Intro-Short/Exercises/01/Begin/
- Look for comments labeled with “EXERCISE”
- Need to include, initialize, and finalize Kokkos library
- Parallelize loops with `parallel_for` or `parallel_reduce`
- Use lambdas instead of functors for computational bodies.
- For now, this will only use the CPU.
Exercise #1: logistics

Compiling for CPU

# gcc using OpenMP (default) and Serial back-ends,
# (optional) non-default arch set with KOKKOS_ARCH
make -j KOKKOS_DEVICES=OpenMP,Serial KOKKOS_ARCH=SNB
# KOKKOS_ARCH Options: See the wiki at
# https://github.com/kokkos/kokkos/wiki/Compiling

Running on CPU with OpenMP back-end

# Set OpenMP affinity
export OMP_NUM_THREADS=8
export OMP_PROC_BIND=spread OMP_PLACES=threads
# Print example command line options:
./01_Exercise.host -h
# Run with defaults on CPU
./01_Exercise.host
# Run larger problem
./01_Exercise.host -S 26

Things to try:

- Vary number of threads
- Vary problem size (-S ...), Vary number of rows (-N ...)
Exercise #1 results

<y,Ax> Exercise 01, Fixed Size

Bandwidth (GB/s) vs. Number of Rows (N)

- HSW
- KNL
- KNL (HBM)
Exercise #2: Inner Product, Flat Parallelism on the CPU, with Views

- **Location:** Intro-Short/Exercises/02/Begin/
- **Assignment:** Change data storage from arrays to Views.
- **Compile and run on CPU, and then on GPU with UVM**

```make
make -j KOKKOS_DEVICES=OpenMP  # CPU-only using OpenMP
make -j KOKKOS_DEVICES=Cuda \n
   KOKKOS_CUDA_OPTIONS=force_uvm,enable_lambda

# Run exercise
./02_Exercise.host -S 26
./02_Exercise.cuda -S 26
# Note the warnings, set appropriate environment variables
```

- Vary problem size: `-S #`
- Vary number of rows: `-N #`
- Vary repeats: `-nrepeat #`
- Compare performance of CPU vs GPU
Exercise #3: Flat Parallelism on the GPU, Views and Host Mirrors

Details:

▶ Location: Intro-Short/Exercises/03/Begin/
▶ Add HostMirror Views and deep copy
▶ Make sure you use the correct view in initialization and Kernel

```plaintext
# Compile for CPU
make -j KOKKOS_DEVICES=OpenMP
# Compile for GPU (we do not need UVM anymore)
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./03_Exercise.cuda -S 26
```

Things to try:

▶ Vary problem size and number of rows (-S ...; -N ...)
▶ Change number of repeats (-nrepeat ...)
▶ Compare behavior of CPU vs GPU
Details:

- Location: Intro-Short/Exercises/04/Begin/
- Replace ‘‘N’’ in parallel dispatch with RangePolicy<ExecSpace>
- Add MemSpace to all Views and Layout to A
- Experiment with the combinations of ExecSpace, Layout to view performance

Things to try:

- Vary problem size and number of rows (-S ...; -N ...)
- Change number of repeats (-nrepeat ...)
- Compare behavior of CPU vs GPU
- Compare using UVM vs not using UVM on GPUs
- Check what happens if MemSpace and ExecSpace do not match.
Exercise #4: Inner Product, Flat Parallelism

<y|Ax> Exercise 04 (Layout) Fixed Size

KNL: Xeon Phi 68c  HSW: Dual Xeon Haswell 2x16c  Pascal60: Nvidia GPU

Bandwidth (GB/s)

Number of Rows (N)

Why?