Kokkos Tutorial

Jeff Amelang \(^2\), Christian R. Trott \(^1\), H. Carter Edwards \(^1\)

\(^1\)Sandia National Laboratories

\(^2\)Harvey Mudd College

Supercomputing’15, November 16, 2015

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

SAND2015-9620 C
Prerequisites for Tutorial Exercises

Compilers and Libraries for your Compute Node

▶ **CPU**: GCC 4.7.2 (or newer) OR Intel 14 (or newer) OR Clang 3.5.2 (or newer)

▶ **GPU**: CUDA nvcc 6.5.14 (or newer) AND NVIDIA compute capability 3.0 (or newer)

Install Kokkos and Exercises on your Compute Node

▶ **Kokkos**: [github.com/kokkos/kokkos](https://github.com/kokkos/kokkos),

  *clone in* `${HOME}/kokkos`

▶ **Tutorial**: [github.com/kokkos/kokkos-tutorials/SC15](https://github.com/kokkos/kokkos-tutorials/SC15)

  *makefiles look for* `${HOME}/kokkos`

Knowledge of **C++**: class constructors, member variables, member functions, member operators, template arguments
Understand Kokkos Programming Model Abstractions

- What, how and why of performance portability
- Productivity and hope for future-proofing
Understand Kokkos Programming Model Abstractions

- What, how and why of *performance portability*
- Productivity and hope for future-proofing

**Part One:**

- Simple data parallel computations
- Deciding where code is run and where data is placed
Understand Kokkos Programming Model Abstractions

- What, how and why of performance portability
- Productivity and hope for future-proofing

Part One:

- Simple data parallel computations
- Deciding where code is run and where data is placed

Part Two:

- Managing data access patterns for performance portability
- Thread safety and thread scalability
- Thread-teams for maximizing parallelism
High performance computers are increasingly **heterogenous**. 

MPI-only is no longer sufficient.

For **portability**: OpenMP, OpenACC, ... or Kokkos.

Only Kokkos obtains performant memory access patterns via **architecture-aware** arrays and work mapping. 

*i.e.*, not just portable, **performance portable**.

With Kokkos, **simple things stay simple** (parallel-for, etc.). 

*i.e.*, it’s **no more difficult** than OpenMP.

**Advanced performance-optimizing patterns are simpler** with Kokkos than with native versions. 

*i.e.*, you’re **not missing out** on advanced features.
Kokkos and the HPC Landscape

Learning objectives:

- How Kokkos fits in the context of modern HPC.
- Kokkos scope, goals, and philosophy.
- Difference between Kokkos and `#pragma` methods.
Compute nodes will be **heterogeneous** in cores *and* memory:
Many-core revolution: 20-year “just recompile” free ride is over.

How much do I have to learn and change to use these nodes?
Key Considerations for GPUs:

- GPUs support **thousands** of simultaneously-executing threads.
- You need $O(10,000)$ threads to use a GPU effectively.
- Cores are "simple" - no transistors are dedicated to branch prediction, out of order execution, etc. Instead, more cores.
- Current GPUs can’t *performantly* access CPU memory, you have to move data
- *GPU cores cannot run* MPI’s *heavy processes.*
Operating assumptions:

- Compute nodes have \( \sim 50 \) complex cores, \( \sim 5000 \) simple cores, and heterogenous memory.
- Separate inter-node and intra-node programming models e.g., message passing + threading)
Operating assumptions:

- Compute nodes have ~50 complex cores, ~5000 simple cores, and heterogenous memory.
- Separate inter-node and intra-node programming models e.g., message passing + threading)

Goal: run on multiple architectures.

Solutions:

- Maintain separate versions for each target architecture (Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)
Operating assumptions:
- Compute nodes have \( \sim 50 \) complex cores, \( \sim 5000 \) simple cores, and heterogeneous memory.
- Separate inter-node and intra-node programming models e.g., message passing + threading.

Goal: run on multiple architectures.

Solutions:
- Maintain separate versions for each target architecture (Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)
- Use a language or a library that runs on multiple architectures (e.g., OpenMP, OpenACC, OpenCL, Kokkos)
  - Note: not all alternatives support heterogeneous memory.
Operating assumptions:

- Compute nodes have ~50 complex cores, ~5000 simple cores, and heterogenous memory.
- Separate inter-node and intra-node programming models e.g., message passing + threading)

Goal: run on multiple architectures.

Solutions:

- Maintain separate versions for each target architecture (Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)
- Use a language or a library that runs on multiple architectures (e.g., OpenMP, OpenACC, OpenCL, Kokkos)
  - Note: not all alternatives support heterogenous memory
Important Point

There's a difference between *portability* and *performance portability*.

**Example**: implementations may target particular architectures and may not be *thread scalable*.

(e.g., locks on CPU won't scale to 100,000 threads on GPU)
Important Point
There’s a difference between portability and performance portability.

Example: implementations may target particular architectures and may not be thread scalable.
(e.g., locks on CPU won’t scale to 100,000 threads on GPU)

Goal: write one implementation which:

- compiles and runs on multiple architectures,
- obtains performant memory access patterns across architectures,
- can leverage architecture-specific features where possible.
Important Point

There’s a difference between *portability* and *performance portability*.

**Example:** implementations may target particular architectures and may not be *thread scalable*.
   
   (e.g., locks on CPU won’t scale to 100,000 threads on GPU)

**Goal:** write **one implementation** which:

- compiles and **runs on multiple architectures**,
- obtains **performant memory access patterns** across architectures,
- can leverage **architecture-specific features** where possible.

**Kokkos:** performance portability across manycore architectures.
Threaded (intra-node) data parallelism

Learning objectives:

- Terminology of pattern, policy, and body.
- The data layout problem.
Loop bodies are prime candidates for **data parallelism**.

**Test:** Same answer if the loop iterates backwards? random order?
Loop bodies are prime candidates for **data parallelism**.

**Test**: Same answer if the loop iterates backwards? random order?

**Examples:**

- Thermodynamic quantities at quadrature points in FEA:

```plaintext
define function
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;
}
```
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

```c
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;
}
```

**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.
What if we want to thread the FEA algorithm?

```c
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;
}
```

(Change the execution policy from "serial" to "parallel.")

OpenMP is simple for parallelizing loops on multi-core CPUs, but what if we then want to do this on other architectures? Intel MIC and NVIDIA GPU and AMD Fusion and...
What if we want to **thread** the FEA algorithm?

```c
#pragma omp parallel for
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;
}
```

(Change the *execution policy* from "serial" to "parallel.")
What if we want to **thread** the FEA algorithm?

```c
#pragma omp parallel for
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;
}
```

(Change the *execution policy* from “serial” to “parallel.”)

OpenMP is simple for parallelizing loops on multi-core CPUs, but what if we then want to do this on **other architectures**?

Intel MIC and NVIDIA GPU and AMD Fusion and ...
**Option 1: OpenMP 4.0**

```c
#pragma omp target data map(...)  
#pragma omp teams num_teams(...) num_threads(...) private(...)  
#pragma omp distribute
for (element = 0; element < numElements; ++element) {
    total = 0
#pragma omp parallel for
    for (qp = 0; qp < numQPs; ++qp)
        total += dot(left[element][qp], right[element][qp]);
    elementValues[element] = total;
}
```
Option 1: OpenMP 4.0

```c
#pragma omp target data map(...)  
#pragma omp teams num_teams(...) num_threads(...) private(...)  
#pragma omp distribute  
for (element = 0; element < numElements; ++element) {
    total = 0  
#pragma omp parallel for  
    for (qp = 0; qp < numQPs; ++qp)
        total += dot(left[element][qp], right[element][qp]);
    elementValues[element] = total;
}
```

Option 2: OpenACC

```c
#pragma acc parallel copy(...) num_gangs(...) vector_length(...)  
#pragma acc loop gang vector  
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;
}
```
A standard thread parallel programming model may give you portable parallel execution if it is supported on the target architecture.

But what about performance?
A standard thread parallel programming model may give you portable parallel execution if it is supported on the target architecture.

But what about performance?

Performance depends upon the computation’s memory access pattern.
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        for (i = 0; i < vectorSize; ++i) {
            total +=
                left[element * numQPs * vectorSize +
                    qp * vectorSize + i] *
                right[element * numQPs * vectorSize +
                    qp * vectorSize + i];
        }
    }
    elementValues[element] = total;
}
#pragma something, opencl, etc.

```c
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        for (i = 0; i < vectorSize; ++i) {
            total +=
                left[element * numQPs * vectorSize +
                qp * vectorSize + i] *
                right[element * numQPs * vectorSize +
                qp * vectorSize + i];
        }
    }
    elementValues[element] = total;
}
```

**Memory access pattern problem:** CPU data layout reduces GPU performance by more than 10X.
Problem: memory access pattern

```c
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
    total = 0;
    for (qp = 0; qp < numQPs; ++qp) {
        for (i = 0; i < vectorSize; ++i) {
            total +=
                left[element * numQPs * vectorSize +
                    qp * vectorSize + i] *
                right[element * numQPs * vectorSize +
                    qp * vectorSize + i];
        }
    }
    elementValues[element] = total;
}
```

**Memory access pattern problem:** CPU data layout reduces GPU performance by more than 10X.

**Important Point**

For performance, the memory access pattern *must* depend on the architecture.
How does Kokkos 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 GPGPU, 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*
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

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

Kokkos maps work to 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.
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.
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:

```c++
struct ParallelFunctor {
    ...
    void operator()( a work assignment ) const {
        /* ... computational body ... */
    }
    ...
};
```
How is work assigned to functor operators?
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

**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,

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

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

```cpp
struct Functor {
    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,

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

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

```c++
struct Functor {
    void operator()(const size_t index) const {...}
}
```

Warning: concurrency and order

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

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

```cpp
struct AtomForceFunctor {
    ...
    void operator()(const size_t atomIndex) const {
        atomForces[atomIndex] = calculateForce(...data...);
    }
    ...
}
```
How is data passed to computational bodies?

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

```cpp
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
class AtomForceFunctor {
    ForceType _atomForces;
    AtomDataType _atomData;
    void operator()(const size_t atomIndex) const {
        _atomForces[atomIndex] = calculateForce(_atomData);
    }
};
```
Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    AtomDataType _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);
}
```
Using Kokkos for data parallel patterns (5)

Putting it all together: the complete functor:

```cpp
struct AtomForceFunctor {
    ForceType _atomForces;
    AtomDataType _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);
}
```cpp```

Functor

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

1. Defining the functor (operator+data):

```c++
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:

```c++
AtomForceFunctor functor(atomForces, data);
Kokkos::parallel_for(numberOfAtoms, functor);
```
Functors are verbose $\Rightarrow$ \texttt{C++11 Lambda} are concise

\begin{verbatim}
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
    [=] (const size_t atomIndex) {
        atomForces[atomIndex] = calculateForce(data);
    }
);  
\end{verbatim}
Functors are verbose ⇒ **C++11 Lambda** 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 verbose ⇒  **C++11 Lambda** 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 (e.g., to GPU) a lambda must capture by value [=]. Don't capture containers (e.g., std::vector) by value because this copies 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:

\[ y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) \, dx \]
Scalar 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;
Scalar integration (0)

Riemann-sum-style numerical integration:

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

```cpp
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 would we \texttt{parallelize} it?
Riemann-sum-style numerical integration:

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

How would we parallelize it?
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!)
An (incorrect) solution to the (incorrect) attempt:

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

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?

```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);
```
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?

double finalReducedValue = 0;
parallel_reduce(N, functor, finalReducedValue);
Example: Scalar integration

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

```c
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 value to update is an **thread-private value** that is made and used by Kokkos; it is not the final reduced value.
Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.
Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

Simplistic data-parallel performance model: \( \text{Time} = \alpha + \frac{\beta \cdot N}{P} \)

- \( \alpha \) = dispatch overhead
- \( \beta \) = time for a unit of work
- \( N \) = number of units of work
- \( P \) = available concurrency
Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

Simplistic data-parallel performance model: \( \text{Time} = \alpha + \frac{\beta \ast N}{P} \)

- \( \alpha \) = dispatch overhead
- \( \beta \) = time for a unit of work
- \( N \) = number of units of work
- \( P \) = available concurrency

\[
\text{Speedup} = P \div \left(1 + \frac{\alpha \ast P}{\beta \ast N}\right)
\]

- Should have \( \alpha \ast P \ll \beta \ast N \)
- All runtimes strive to minimize launch overhead \( \alpha \)
- Find more parallelism to increase \( N \)
- Merge (fuse) parallel operations to increase \( \beta \)
**Results:** illustrates simple speedup model \( P \div \left( 1 + \frac{\alpha P}{\beta N} \right) \)
Exercise: Inner product $\langle y, A \times x \rangle$

Details:
- $y$ is $N \times 1$, $A$ is $N \times M$, $x$ is $M \times 1$
- We’ll use this exercise throughout the tutorial
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>
Compiling for CPU

```
cd ~/kokkos-tutorial/SC15/Exercises/01/
# gcc using OpenMP (default) and Serial back-ends
make -j 4 [KOKKOS_DEVICES=OpenMP,Serial]
# Intel using OpenMP (default) and Serial back-ends
make -j 4 CXX=icpc [KOKKOS_DEVICES=OpenMP,Serial]
# Intel using OpenMP for Xeon Phi Knights Corner cross-compile
# For execution natively on the KNC. NOT for offload.
make -j CXX=icpc [KOKKOS_DEVICES=OpenMP,Serial] KOKKOS_ARCH=KNC
```

Running on CPU with OpenMP back-end

```
# Set OpenMP affinity
export GOMP_CPU_AFFINITY=0-NumberOfCoresOnASingleSocket
# Print example command line options:
./exercise.host -h
# Run with defaults on CPU
./exercise.host
```
Exercise #1: Inner Product, Flat Parallelism on the CPU

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

Details:
- Location: ~/kokkos-tutorials/SC15/Exercises/01/
- See
  ~/kokkos-tutorials/SC15/Exercises/HOW_TO_COMPILE_AND_RUN
- Look for comments labeled with “EXERCISE”
- 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 results

<y,Ax> Exercise01, fixed problem size

Bandwidth (GB/s) vs. number of rows for KNC and HSW.
Review: Simple parallel reduce using a lambda:

```cpp
ReductionType reducedValue; // initial value irrelevant
Kokkos::parallel_reduce(numberOfIterations,
  [=] (const size_t index,
    ReductionType & valueToUpdate) {
    valueToUpdate += // ... contribution for index
  },
  reducedValue);
```
Review: Simple parallel reduce using a lambda:

```cpp
ReductionType reducedValue; // initial value irrelevant
Kokkos::parallel_reduce(numberOfIterations,
    [=] (const size_t index,
        ReductionType & valueToUpdate) {
        valueToUpdate += // ... contribution for index
    },
    reducedValue);
```

**Limitation** of using defaults: the reduced value is (re-)initialized to zero and is reduced with operator+=.
Review: Simple parallel reduce using a lambda:

```c++
ReductionType reducedValue; // initial value irrelevant
Kokkos::parallel_reduce(numberOfIterations,
    [=] (const size_t index,
        ReductionType & valueToUpdate)
    {
        valueToUpdate += // ... contribution for index
    },
    reducedValue);
```

**Limitation** of using defaults: the reduced value is (re-)initialized to zero and is reduced with operator\(+=\).

For non-trivial reductions you need to use a **general reduction** functor.
How do you do **arbitrary reductions**?

**Example: finding index of closest point**

```c
Point searchLocation = ...;
size_t indexOfClosest = 0;
for (size_t i = 1; i < numberOfPoints; ++i) {
    if (magnitude(searchLocation - points[i]) <
        magnitude(searchLocation - points[indexOfClosest])) {
        indexOfClosest = i;
    }
}
```

This isn't possible with OpenMP's reduction clause. Manual threading versions must avoid false sharing. Parallel programming models should support robust, arbitrary, performant reductions tuned to the architecture.
How do you do **arbitrary reductions**?

**Example: finding index of closest point**

```c
Point searchLocation = ...;
size_t indexOfClosest = 0;
for (size_t i = 1; i < numberOfPoints; ++i) {
    if (magnitude(searchLocation - points[i]) <
        magnitude(searchLocation - points[indexOfClosest]))) {
        indexOfClosest = i;
    }
}
```

- This **isn’t possible** with openmp’s reduction clause
- Manual threading versions must avoid **false sharing**
- Parallel programming models should support **robust, arbitrary, performant** reductions **tuned to the architecture**.
General reductions:

**What information** must we provide to do a reduction?

- The **type** of the value to reduce ("value_type")
- How to combine ("join") two value_types
- How to **initialize** a value_type

```cpp
struct ParallelFunctor {
    typedef double value_type;
    void operator()(const size_t index,
                    value_type & valueToUpdate) const {...}  
    void join(volatile value_type & destination,
              const volatile value_type & source) const {...}
    void init(value_type & initialValue) const {...}
};
```
Advanced features we haven’t covered

▶ Exclusive and inclusive `prefix scan` with the `parallel_scan` pattern.
▶ Using `tag dispatch` interface to allow non-trivial functors to have multiple “`operator()`” functions.
▶ Directed acyclic graph (DAG) of tasks pattern (experimental).
▶ **Concurrently** executing parallel kernels on CPU and GPU (experimental).
▶ Hierarchical parallelism with **team policies**, covered later.
Simple usage is similar to OpenMP, advanced features are also straightforward.

Three common data-parallel patterns are `parallel_for`, `parallel_reduce`, and `parallel_scan`.

A parallel computation is characterized by its pattern, policy, space, and body.

User provides computational bodies as functors or lambdas which handle a single work item.
Views

Learning objectives:

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

```cpp
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.
Solution: We need a way of storing data (multidimensional arrays) which can be communicated to accelerator (GPU).
⇒ Views, Lambda, Functor
Example: running daxpy on the GPU:

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:

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

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

```c
void assignValueInView(View<double *> data) { data(0) = 3; }

View<double *> a("a", N0), b("b", N0);
a(0) = 1;
b(0) = 2;
a = b;
View<double *> c(b);
assignValueInView(c);
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**.

**Example:**

```cpp
void assignValueInView(View<double*> data) { data(0) = 3; }

View<double*> a("a", N0), b("b", N0);
  a(0) = 1;
  b(0) = 2;
  a = b;
View<double*> c(b);
assignValueInView(c);
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**.

**Example:**

```cpp
void assignValueInView(View<double*> data) { data(0) = 3; }

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

What gets printed? 3.0
Exercise: Inner product $< y, A \cdot x >$

Details:
- Location: ~/kokkos-tutorials/SC15/Exercises/02/
- Change data storage from arrays to Views.
- Use lambdas instead of functors for computational bodies.
- For now, this will only use the CPU.
Advanced features we haven’t covered

- **Memory space** in which view’s data resides *covered next*.
- **deep_copy** view’s data; *covered later.*
  Note: Kokkos *never* hides a deep_copy of data.
- **Layout** of multidimensional array; *covered later.*
- **Memory traits**; *covered later.*
- **Subview**: Generating a view that is a “slice” of other multidimensional array view; *will not be covered today.*
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 memory 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?

⇒ Execution spaces
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, ...
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.

Host

```cpp
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(... data ...);
```

Parallel

```cpp
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 Host code be run? CPU? GPU?
⇒ Always in the **host process**

Where will Parallel code be run? CPU? GPU?
⇒ The **default execution space**
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**.
Changing the parallel execution space:

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

```
Default
```
```
parallel_for(
    numberOfIntervals, // == RangePolicy<(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

```
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 (CUDA requires v 7.5)

```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

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< ExecutionSpace >(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<
    ExecutionSpace>(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<ExecutionSpace>(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<

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

double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy<ExecutionSpace>(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(...);`
Important concept: Memory spaces

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

- View\langle\text{double***}, \text{MemorySpace}\rangle \text{ 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\(<\text{double}***, Memory\ Space> \text{ data(\ldots);}\)
- 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(...);
Example: HostSpace

View<\texttt{double**}, \texttt{HostSpace}> \texttt{hostView(...);}

Example: CudaSpace

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

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

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

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

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

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

(failed) Attempt 1:

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:

```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_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += array(index);
    },
    sum);
```
Example (redux): summing an array with the GPU

(failed) Attempt 2:

```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);
  },
  sum);
```
Example (redux): summing an array with the GPU

(failed) Attempt 2:

```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:

```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?

- CudaUVMSpace
- CudaHostPinnedSpace
- Mirroring
CudaUVMSpace

View<
double*,

CudaUVMSpace> array
array = ...from file...
double sum = 0;
parallel_reduce(N,
[=] (int i,
    double & d) {
    d += array(i);
},
sum);

Cuda runtime automatically handles data movement, at performance hit.
CudaHostPinnedSpace

View<double*,
    CudaHost...> array;
array = ...from file...
double sum = 0;
parallel_reduce(N,
    [=] (int i,
        double & d) {
            d += array(i);
    },
    sum);

Cuda runtime allows cuda-code access to CPU memory,
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);

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>host mirror</td>
<td>RAM</td>
</tr>
<tr>
<td>view</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td>deep_copy</td>
</tr>
</tbody>
</table>
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.
   
   ```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);
```
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(view);
   ```

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

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.
   
   ```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.).

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_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(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_LAMBDAA (...){ 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);
   ```
View and Spaces Section 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.
- Heterogenous nodes have one or more memory spaces.
- Mirroring is used for performant access to views in host and device memory.
- Heterogenous 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

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.
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);
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);

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<

\texttt{double***, Layout, Space}>\ name(...);\n
\texttt{LayoutLeft}: left-most index is stride 1.

\texttt{LayoutRight}: right-most index is stride 1.

If no layout specified, default for that memory space is used.

\texttt{LayoutLeft} for \texttt{CudaSpace}, \texttt{LayoutRight} for \texttt{HostSpace}.

Layouts are extensible: \texttt{\sim} 50 lines

Advanced layouts: \texttt{LayoutStride}, \texttt{LayoutTiled}, ...
Important concept: Layout

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

```c
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 #3: Inner Product, Flat Parallelism

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

**Details:**
- Location: ~/kokkos-tutorials/SC15/Exercises/03/
- Use lambdas instead of functors for computational bodies.
- Replace ‘‘N’’ in parallel dispatch with RangePolicy<Space>
- Add Space to all Views and Layout to A
- Experiment with the combinations of Space, Layout to view performance
Exercise #3: Inner Product, Flat Parallelism

<y,Ax> Exercise 03, fixed problem size

Why?
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?
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.
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:

```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.

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

![Diagram of CPU caches]

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

![Diagram of GPU caches]
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 (more later).
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 (more later).
Consider the array summation example:

\[
\text{View<double*, Space> data("data", size);}
\]

\[
\ldots \text{populate data...}
\]

\[
\text{double sum = 0;}
\]

\[
\text{Kokkos::parallel_reduce(}
\]

\[
\text{RangePolicy< Space>(0, size),}
\]

\[
\text{KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {}
\]

\[
\text{valueToUpdate += data(index);}
\]

\[
},
\]

\[
\text{sum);
\]

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

\[
\text{View<double*, Space> data("data", size);} \\
...\text{populate data...}
\]

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

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

Given \( P \) threads, \textbf{which indices} do we want thread 0 to handle?

\textbf{Contiguous:} \quad 0, 1, 2, \ldots, N/P

\textbf{Strided:} \quad 0, N/P, 2*N/P, \ldots
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, \textbf{which indices} do we want thread 0 to handle?

- \textbf{Contiguous}: 0, 1, 2, ..., \( N/P \)
- \textbf{Strided}: 0, \( N/P \), 2*\( N/P \), ...

\textbf{CPU} \hspace{2cm} \textbf{GPU}

Why?
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?
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.
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.

**Important point**

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

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

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

Analysis: row-major (LayoutRight)
Important point

Performance 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

Performance memory access is achieved by Kokkos mapping parallel work indices and multidimensional array layout \textit{optimally for the architecture}.

\textbf{Analysis: column-major (LayoutLeft)}
Important point

Performance memory access is achieved by Kokkos mapping parallel work indices \textbf{and} multidimensional array layout \textit{optimally for the architecture.}

Analysis: \textbf{column-major (LayoutLeft)}

- \textbf{HostSpace}: uncached (bad)
- \textbf{CudaSpace}: coalesced (good)
Analysis: Kokkos architecture-dependent

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

(a) OpenMP
(b) Cuda

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

Layout performance, revisited

<y,Ax> Exercise03, fixed problem size

<table>
<thead>
<tr>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>K40 Right</td>
</tr>
<tr>
<td>K40 Left</td>
</tr>
<tr>
<td>KNC Right</td>
</tr>
<tr>
<td>KNC Left</td>
</tr>
<tr>
<td>HSW Right</td>
</tr>
<tr>
<td>HSW Left</td>
</tr>
</tbody>
</table>

Legend:
- Coalesced
- Cached
- Uncached

Number of rows vs. Bandwidth (GB/s) graph.
Every View has a Layout set at compile-time through a template parameter.

LayoutRight and LayoutLeft are most common.

Views in HostSpace default to LayoutRight and Views in CudaSpace default to LayoutLeft.

Layouts are extensible and flexible.

For performance, memory access patterns must result in caching on a CPU and coalescing on a GPU.

Kokkos maps parallel work indices and multidimensional array layout for performance portable memory access patterns.

There is nothing in OpenMP, OpenACC, or OpenCL to manage layouts.

⇒ You'll need multiple versions of code or pay the performance penalty.
Thread safety and atomic operations

Learning objectives:

▶ Understand that coordination techniques for low-count CPU threading are not scalable.

▶ Understand how atomics can parallelize the scatter-add pattern.

▶ Gain performance intuition for atomics on the CPU and GPU, for different data types and contention rates.
**Examples: Histogram**

**Histogram kernel:**

```cpp
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    ++_histogram(bucketIndex);
});
```

**Problem:** Multiple threads may try to write to the same location.

**Solution strategies:**
- Locks
- Thread-private copies
- Atomics

http://www.farmaceuticas.com.br/tag/graficos/
**Histogram kernel:**

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    ++_histogram(bucketIndex);
});
```

**Problem:** Multiple threads may try to write to the same location.

http://www.farmaceuticas.com.br/tag/graficos/
**Histogram kernel:**

```cpp
class parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    ++_histogram(bucketIndex);
});
```

**Problem:** Multiple threads may try to write to the same location.

**Solution strategies:**
- Locks
- Thread-private copies
- Atomics

http://www.farmaceuticas.com.br/tag/graficos/
Thread safety solution: Locks

```cpp
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    // LOCK the lock that protects bucket bucketIndex
    ++_histogram(bucketIndex);
    // UNLOCK the lock that protects bucket bucketIndex
});
```
Thread safety solution: Locks

```cpp
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    // LOCK the lock that protects bucket bucketIndex
    ++_histogram(bucketIndex);
    // UNLOCK the lock that protects bucket bucketIndex
});
```

**Problem**: contention is too high at $O(10,000)$ threads.
Thread safety solution: Thread-private copies

```plaintext
#pragma omp parallel shared(histogram)
{
    HistogramType thisThreadsHistogram(histogram.size())
#pragma omp for nowait
    for each input {
        ...  
        const int value = ...;
        const int bucketIndex = computeBucketIndex(value);
        ++thisThreadsHistogram(bucketIndex);
    }
#pragma omp critical
    for each bucket {
        histogram[bucketIndex] += thisThreadsHistogram[bucketIndex];
    }
}
```
Thread safety solution: Thread-private copies

```
#pragma omp parallel shared(histogram)
{
    HistogramType thisThreadsHistogram(histogram.size())
    #pragma omp for nowait
    for each input {
        ... 
        const int value = ...;
        const int bucketIndex = computeBucketIndex(value);
        ++thisThreadsHistogram(bucketIndex);
    }
    #pragma omp critical
    for each bucket {
        histogram[bucketIndex] += thisThreadsHistogram[bucketIndex];
    }
}
```

**Problems:** insufficient memory for `thisThreadsHistogram` ratio of parallel/serial work too low.
Thread safety solution: Atomics

```cpp
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    Kokkos::atomic_add(&_histogram(bucketIndex), 1);
});
```

Atomics are the only scalable solution to thread safety. Locks or data replication are strongly discouraged.
Thread safety solution: Atomics

```cpp
parallel_for(N, KOKKOS_LAMBDAC(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    Kokkos::atomic_add(&_histogram(bucketIndex), 1);
});
```

- Atomics are the **only scalable** solution to thread safety.
Thread safety solution: Atomics

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const int value = ...;
    const int bucketIndex = computeBucketIndex(value);
    Kokkos::atomic_add(&_histogram(bucketIndex), 1);
});
```

- Atomics are the **only scalable** solution to thread safety.
- Locks or data replication are **strongly discouraged**.
How expensive are atomics?

Thought experiment: scalar integration

```c
operator() (const unsigned int intervalIndex ,
            double & valueToUpdate) const {
    double contribution = function (...);
    valueToUpdate += contribution;
}
```
How expensive are atomics?

Thought experiment: scalar integration

```cpp
operator() (const unsigned int intervalIndex,
            double & valueToUpdate) const {
    double contribution = function(...);
    valueToUpdate += contribution;
}
```

Idea: what if we instead do this with `parallel_for` and atomics?

```cpp
operator() (const unsigned int intervalIndex) const {
    const double contribution = function(...);
    Kokkos::atomic_add(&globalSum, contribution);
}
```

How much of a performance penalty is incurred?
Two costs: (independent) work and coordination.

```c++
parallel_reduce(numberOfIntervals,
    KOKKOS_LAMBDA (const unsigned int intervalIndex,
        double & valueToUpdate) {
        valueToUpdate += function(...);
    }, totalIntegral);
```

▶ This is the most extreme case: all coordination and no work.
▶ Contention is captured by the `atomicStride`.

`atomicStride` → 1 ⇒ Scalar integration
`atomicStride` → large ⇒ Independent
**Two costs:** (independent) work and coordination.

```cpp
parallel_reduce(numberOfIntervals, 
    KOKKOS_LAMBDA (const unsigned int intervalIndex, 
                  double & valueToUpdate) {
        valueToUpdate += function(...);
    }, totalIntegral);
```

Experimental setup

```cpp
operator()(const unsigned int index) const {
    Kokkos::atomic_add(&globalSums[index % atomicStride], 1);
}
```

- This is the most extreme case: all coordination and no work.
- Contention is captured by the atomicStride.
  - atomicStride → 1 ⇒ Scalar integration
  - atomicStride → large ⇒ Independent
Atomics performance: 1 million adds, no work per kernel

Slowdown from atomics: Summary for 1 million adds, mod, 0 pows

Note: log scale

log10(speedup over independent) [-]
log10(contention) [-]
**Performance of atomics (2)**

**Atomics performance:** 1 million adds, **no** work per kernel

![Graph showing slowdown from atomics: No penalty for low contention and high penalty for high contention.](image)

- **Note:** log scale
- **log10(speedup over independent)** [\(-\)]
- **log10(contention)** [\([-\)]

- **CUDA**:
  - `double`
  - `size_t`
  - `float`
  - `unsigned`
- **OMP**:
  - `double`
  - `size_t`
  - `float`
  - `unsigned`
- **PHI**:
  - `double`
  - `size_t`
  - `float`
  - `unsigned`
**Performance of atomics (3)**

**Atomics performance:** 1 million adds, **some** work per kernel

Slowdown from atomics: Summary for 1 million adds, mod, 2 pows

Note: log scale

- No penalty for any contention
- High penalty for high contention

Graph: Log-log plot showing the slowdown in performance of atomics compared to independent operations. The x-axis represents the log10 of contention, while the y-axis represents the log10 of the speedup over independent operations. Different colors and line styles indicate various data types and parallel programming environments.
Performance of atomics (4)

**Atomics performance:** 1 million adds, **lots of** work per kernel

Slowdown from atomics: Summary for 1 million adds, mod, 5 pows

**No penalty for any contention**

**High penalty for high contention**

Note: log scale

- **cuda double**
- **cuda size_t**
- **cuda float**
- **cuda unsigned**
- **omp double**
- **omp size_t**
- **omp float**
- **omp unsigned**
- **phi double**
- **phi size_t**
- **phi float**
- **phi unsigned**
Advanced features

Atomics on arbitrary types:

- Atomic operations work if the corresponding operator exists, i.e., `atomic_add` works on any data type with `+`.
- Atomic exchange works on any data type.

```cpp
// Assign *dest to val, return former value of *dest
template<typename T>
T atomic_exchange(T * dest, T val);
// If *dest == comp then assign *dest to val
// Return true if succeeds.
template<typename T>
bool atomic_compare_exchange_strong(T * dest, T comp, T val);
```
View **memory traits**:  

- Beyond a Layout and Space, Views can have memory traits.
- Memory traits either provide _convenience_ or allow for certain **hardware-specific optimizations** to be performed.

Example: If all accesses to a View will be atomic, use the Atomic memory trait:

```cpp
View< double**, Layout, Space, 
    MemoryTraits< Atomic > > forces( ... );
```
Memory traits:

- Beyond a Layout and Space, Views can have memory traits.
- Memory traits either provide convenience or allow for certain hardware-specific optimizations to be performed.

Example: If all accesses to a View will be atomic, use the Atomic memory trait:

```cpp
View<double**, Layout, Space, MemoryTraits<Atomic>> > forces(...);
```

Many memory traits exist or are experimental, including Read, Write, ReadWrite, ReadOnce (non-temporal), Contiguous, and RandomAccess.
**Example:** RandomAccess memory trait:

On **GPUs**, there is a special pathway for fast **read-only, random** access, originally designed for textures.
**Example:** RandomAccess memory trait:

On GPUs, there is a special pathway for fast read-only, random access, originally designed for textures.

**How to access texture memory via CUDA:**

```c
cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeLinear;
resDesc.res.linear.devPtr = buffer;
resDesc.res.linear.desc.f = cudaChannelFormatKindFloat;
resDesc.res.linear.desc.x = 32; // bits per channel
resDesc.res.linear.sizeInBytes = N*sizeof(float);

cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.readMode = cudaReadModeElementType;

cudaTextureObject_t tex=0;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
```
**Example:** RandomAccess memory trait:

On GPUs, there is a special pathway for fast read-only, random access, originally designed for textures.

How to access texture memory via **CUDA**:

```c
cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeLinear;
resDesc.res.linear.devPtr = buffer;
resDesc.res.linear.desc.f = cudaChannelFormatKindFloat;
resDesc.res.linear.desc.x = 32; // bits per channel
resDesc.res.linear.sizeInBytes = N*sizeof(float);

cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.readMode = cudaReadModeElementType;

cudaTextureObject_t tex=0;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
```

How to access texture memory via **Kokkos**:

```c
View< const double***, Layout, Space, MemoryTraits<RandomAccess> > name(...);
```
Atomics are the only thread-scalable solution to thread safety.

- Locks or data replication are strongly discouraged

Atomic performance depends on ratio of independent work and atomic operations.

- With more work, there is a lower performance penalty, because of increased opportunity to interleave work and atomic.

The Atomic memory trait can be used to make all accesses to a view atomic.

The cost of atomics can be negligible:

- CPU ideal: contiguous access, integer types
- GPU ideal: scattered access, 32-bit types

Many programs with the scatter-add pattern can be thread-scallably parallelized using atomics without much modification.
Hierarchical parallelism

Finding and exploiting more parallelism in your computations.

Learning objectives:

- Similarities and differences between outer and inner levels of parallelism
- Thread teams (league of teams of threads)
- Performance improvement with well-coordinated teams
(Flat parallel) Kernel:

Kokkos::parallel_reduce(N,
    KOKKOS_LAMBDA (const int row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (int col = 0; col < M; ++col) {
            thisRowsSum += A(row,col) * x(col);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);
(Flat parallel) Kernel:

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

Problem: What if we don’t have enough rows to saturate the GPU?
(Flat parallel) Kernel:

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

Problem: What if we don’t have enough rows to saturate the GPU?

Solutions?
(Flat parallel) Kernel:

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

Problem: What if we don’t have enough rows to saturate the GPU?

Solutions?
  ▶ Atomics
  ▶ Thread teams
**Atomics kernel:**

```cpp
Kokkos::parallel_for(N,
    KOKKOS_LAMBDA (const size_t index) {
    const int row = extractRow(index);
    const int col = extractCol(index);
    atomic_add(&result, A(row,col) * x(col));
});
```

Example: inner product (1)
Example: inner product (1)

Atomics kernel:

Kokkos::parallel_for(N,
    KOKKOS_LAMBDA (const size_t index) {
        const int row = extractRow(index);
        const int col = extractCol(index);
        atomic_add(&result, A(row,col) * x(col));
    });

Problem: Poor performance
Doing each individual row with atomics is like doing scalar integration with atomics.

Instead, you could envision doing a large number of `parallel_reduce` kernels.

```cpp
for each row
    Functor functor(row, ...);
    parallel_reduce(M, functor);
```
Doing each individual row with atomics is like doing scalar integration with atomics.

Instead, you could envision doing a large number of parallel_reduce kernels.

```cpp
for each row
    Functor functor(row, ...);
    parallel_reduce(M, functor);
```

This is an example of hierarchical work.

**Important concept: Hierarchical parallelism**

Algorithms that exhibit hierarchical structure can exploit hierarchical parallelism with **thread teams**.
Example: inner product (3)

Important concept: Thread team

A collection of threads which are guaranteed to be executing **concurrently** and **can synchronize**.
Important concept: Thread team
A collection of threads which are guaranteed to be executing **concurrently** and can synchronize.

High-level strategy:
1. Do **one parallel launch** of $N$ teams of $M$ threads.
2. Each thread performs **one** entry in the row.
3. The threads within **teams perform a reduction**.
4. The thread teams **perform a reduction**.
The final hierarchical parallel kernel:

```cpp
parallel_reduce(
    team_policy(N, Kokkos::AUTO),
    KOKKOS_LAMBD(A) (member_type & teamMember, double & update) { 
        int row = teamMember.league_rank();
        double thisRowsSum = 0;
        parallel_reduce(TeamThreadRange(teamMember, M),
            [=] (int col, double & innerUpdate) {
                innerUpdate += A(row, col) * x(col);
            }, thisRowsSum);
        if (teamMember.team_rank() == 0) {
            update += y(row) * thisRowsSum;
        }
    }, result);
```

The **performance** and **flexibility** of teams is *naturally* and *concisely* expressed under the Kokkos model.

Let’s walk through how we got to this *final* answer.
Important point

Using teams is changing the execution \textit{policy}.

\textbf{“Flat parallelism”} uses \texttt{RangePolicy}:

We specify a \textit{total amount of work}.

\begin{verbatim}
// total work = N
parallel_for(RangePolicy<ExecutionSpace>(0,N), functor);
\end{verbatim}
Important point

Using teams is changing the execution policy.

“**Flat** parallelism” uses RangePolicy:

We specify a *total amount of work*.

```cpp
// total work = N
parallel_for(
    RangePolicy<ExecutionSpace>(0,N), functor);
```

“**Hierarchical** parallelism” uses TeamPolicy:

We specify a *team size* and a *number of teams*.

```cpp
// total work = numberOfTeams * teamSize
parallel_for(
    TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize), functor);
```
typedef typename TeamPolicy<ExecSpace>::member_type member_type;

void operator ()(const member_type & teamMember) {
  // Which team am I on?
  const unsigned int leagueRank = teamMember.league_rank();
  // Which thread am I on this team?
  const unsigned int teamRank = teamMember.team_rank();
}
Important point

When using teams, functor operators receive a \textit{team member}.

typedef typename TeamPolicy<ExecSpace>::member_type member_type;

void operator()(const member_type & teamMember) {
    // Which team am I on?
    const unsigned int leagueRank = teamMember.league_rank();
    // Which thread am I on this team?
    const unsigned int teamRank = teamMember.team_rank();
}

Warning

There may be more (or fewer) team members than pieces of your algorithm’s work per team
First attempt at inner product exercise:

```cpp
operator() (const member_type & teamMember ) {
    const unsigned int row = teamMember.league_rank();
    const unsigned int col = teamMember.team_rank();
    atomic_add(&result,y(row) * A(row,col) * x(entry));
}
```
First attempt at inner product exercise:

```cpp
operator() (const member_type & teamMember ) {
    const unsigned int row = teamMember.league_rank();
    const unsigned int col = teamMember.team_rank();
    atomic_add(&result,y(row) * A(row, col) * x(entry));
}
```

- When team size ≠ number of columns, how are units of work mapped to team’s member threads? Is the mapping architecture-dependent?
- `atomic_add` performs badly under high contention, how can team’s member threads performantly cooperate for a nested reduction?
We shouldn’t be hard-coding the work mapping...

```cpp
operator () (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    do a reduction' over M columns',
    [=] (const int col) {
        thisRowsSum += A(row, col) * x(col);
    });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution, we’d use Kokkos::parallel reduce.

Key idea: this is a parallel execution. ⇒ Nested parallel patterns
We shouldn’t be hard-coding the work mapping...

```cpp
operator() (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    "do a reduction"("over M columns",
    [=] (const int col) {
        thisRowsSum += A(row,col) * x(col);
    });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution, 
we’d use Kokkos::parallel_reduce.
We shouldn’t be hard-coding the work mapping...

```cpp
operator () (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    "do a reduction" ("over M columns",
        [=] (const int col) {
            thisRowsSum += A(row, col) * x(col);
        });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution,
    we’d use Kokkos::parallel_reduce.

**Key idea:** this is a parallel execution.
We shouldn’t be hard-coding the work mapping...

```cpp
operator() (member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    "'do a reduction'"("'over M columns'",
        [=] (const int col) {
            thisRowsSum += A(row, col) * x(col);
        });
    if (teamMember.team_rank() == 0) {
        update += (row) * thisRowsSum;
    }
}
```

If this were a parallel execution,
we’d use Kokkos::parallel_reduce.

**Key idea:** this is a parallel execution.

⇒ **Nested parallel patterns**
TeamThreadRange:

operator() (const member_type & teamMember, double & update) {
    const int row = teamMember.league_rank();
    double thisRowsSum;
    parallel_reduce(TeamThreadRange(teamMember, M),
                    [=] (const int col, double & rowUpdate) {
                        rowUpdate += A(row, col) * x(col);
                        thisRowsSum += A(row, col) * x(col);
                    }, thisRowsSum);
    if (teamMember.team_rank() == 0) {
        update += y(row) * thisRowsSum;
    }
}

The mapping of work indices to threads is architecture-dependent.
The amount of work given to the TeamThreadRange need not be a multiple of the team size.
Intra-team reduction handled for you.
TeamThreadRange:

```cpp
operator() (const member_type & teamMember, double & update) {
  const int row = teamMember.league_rank();
  double thisRowsSum;
  parallel_reduce(TeamThreadRange(teamMember, M),
    [=] (const int col, double & rowUpdate) {
      rowUpdate += A(row, col) * x(col);
    }, thisRowsSum);
  if (teamMember.team_rank() == 0) {
    update += y(row) * thisRowsSum;
  }
}
```

- The mapping of work indices to threads is architecture-dependent.
- The amount of work given to the TeamThreadRange need not be a multiple of the team_size.
- Intra-team reduction handled for you.
**Anatomy of nested parallelism:**

```cpp
def parallel_outer(
    TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize),
    KOKKOS_LAMBDA (const member_type & teamMember[ , . . . ]) {
        /* beginning of outer body */
        parallel_inner(
            TeamThreadRange(teamMember, thisTeamsRangeSize),
            [=] (const unsigned int indexWithinBatch[ , . . . ]) {
                /* inner body */
            }[ , . . . ]);
        /* end of outer body */
    }[ , . . . ];
```

- `parallel_outer` and `parallel_inner` may be any combination of `for`, `reduce`, or `scan`.
- The inner lambda may capture by reference, but capture-by-value is recommended.
- The policy of the inner lambda is always a `TeamThreadRange`.
- `TeamThreadRange` cannot be nested.
What should the team size be?

In practice, you can let Kokkos decide:

```c++
parallel_something(
    TeamPolicy<ExecutionSpace>(numberOfTeams, Kokkos::AUTO),
    /* functor */);
```
In practice, you can **let Kokkos decide**: 

```cpp
parallel_something(
    TeamPolicy<ExecutionSpace>(numberOfTeams, Kokkos::AUTO),
    /* functor */);
```

**NVIDIA GPU:**

- Special hardware available for coordination within a team.
- Within a team 32 threads (*warp*) execute “lock step.”
- Maximum team size: **1024**; Recommended team size: **256**
In practice, you can let Kokkos decide:

```cpp
parallel_something(
    TeamPolicy<ExecutionSpace>(numberOfTeams, Kokkos::AUTO),
    /* functor */);
```

**NVIDIA GPU:**
- Special hardware available for coordination within a team.
- Within a team 32 threads (warp) execute “lock step.”
- Maximum team size: 1024; Recommended team size: 256

**Intel Xeon Phi:**
- Recommended team size: # hyperthreads per core
- Hyperthreads share entire cache hierarchy
  - a well-coordinated team avoids cache-thrashing
**Exercise**: Inner product $< y, A \ast x >$

**Details:**
- Location: ~/kokkos-tutorials/SC15/Exercises/03/
- Use lambdas instead of functors for computational bodies.
- Replace RangePolicy<Space> with TeamPolicy<Space>
- Experiment with the combinations of Layout, Space, N to view performance
- Hint: what should the layout of A be?
Exercise #4: Inner Product, Hierarchical Parallelism

<y,Ax> Exercise04, fixed problem size

Bandwidth (GB/s) vs. number of rows for different architectures:
- K40 Right
- K40 Left
- KNC Right
- KNC Left
- HSW Right
- HSW Left

The graph shows the performance of different hardware configurations as the number of rows increases. The x-axis represents the number of rows, while the y-axis shows the bandwidth in GB/s. The performance varies significantly across different architectures and configurations.
Shared memory

Learning objectives:

▶ Understand how shared memory can reduce global memory accesses
▶ Recognize when to use shared memory
▶ Understand how to use shared memory and why barriers are necessary
Each team has access to a “scratch pad”.

![Diagram showing shared memory access](image)
Shared memory (scratch pad) details:

- Accessing data is shared memory is (usually) **much faster** than global memory.
- **GPUs** have separate, dedicated, small, low-latency shared memories (**NOT subject to coalescing requirements**).
- **CPUs** don't have special hardware, but programming with shared memory results in cache-aware memory access patterns.
- Roughly, it's like a *user-managed* L1 cache.
Shared memory (scratch pad) details:

- Accessing data is shared memory is (usually) much faster than global memory.
- **GPUs** have separate, dedicated, small, low-latency shared memories (*NOT subject to coalescing requirements*).
- **CPUs** don't have special hardware, but programming with shared memory results in cache-aware memory access patterns.
- Roughly, it's like a *user-managed L1 cache*.

**Important concept**

When members of a team read the same data multiple times, it's better to load the data into shared memory and read from there.
**Main idea:** Load global data into shared memory and reuse

```cpp
operator()(member_type teamMember) const {
    // Declare team-shared tile of memory
    View< double***
        , execution_space::scratch_memory_space
    > tile( teamMember.team_shared(), ... );

    // copy subgrid data into tile
    teamMember.team_barrier();

    // Compute stencil using tile
}
```
There is a **third level** in the hierarchy below TeamThreadRange: ThreadVectorRange

- Just like for TeamThreadRange, you can perform `parallel_for`, `parallel_reduce`, or `parallel_scan`.
- Important for full performance of Xeon Phi and GPUs

Restricting execution to a **single member**:
- `PerTeam`: one thread per team
- `PerThread`: one vector lane per thread

**Multiple shared views** can be made in shared memory.
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 policy.

Teams can be used to reduce contention for global resources even in “flat” algorithms.

Teams have access to “scratch pad” shared memory.
High performance computers are increasingly \textit{heterogenous}.
\textit{MPI-only is no longer sufficient.}

For \textbf{portability}: OpenMP, OpenACC, ... or Kokkos.

Only Kokkos obtains performant memory access patterns via \textbf{architecture-aware} arrays and work mapping.
\textit{i.e., not just portable, performance portable.}

With Kokkos, \textbf{simple things stay simple} (parallel-for, etc.).
\textit{i.e., it’s no more difficult} than OpenMP.

\textbf{Advanced performance-optimizing patterns} are simpler with Kokkos than with native versions.
\textit{i.e., you’re not missing out} on advanced features.