

Kokkos Tutorial

H. Carter Edwards¹, Christian R. Trott¹, and
Fernanda Foertter²

¹Sandia National Laboratories

²Oak Ridge National Laboratory

Supercomputing, Nov 12-17, 2017

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2017-11593 C

SOFTWARE FOR LAB

Remote Desktop Software:

- ▶ Download NoMachine now for best performance from **www.nomachine.com/download**
- ▶ Alternatively you may use a VNC client or the provided browser-based VNC option

SSH Access Software (optional):

- ▶ PuTTY for Windows can be downloaded from **www.putty.org**
- ▶ Alternatively you may use a provided browser-based SSH option

CONNECTION INSTRUCTIONS

- ▶ Navigate to nvlabs.qwiklab.com
- ▶ Login or create a new account
- ▶ Select the **Instructor-Led Hands-on Labs** Class
- ▶ Find the lab called **Kokkos**, ..., select it, click Select, and finally click Start
- ▶ After a short wait, lab instance Connection information will be shown
- ▶ Please ask Lab Assistants for help!

Kokkos Tutorial

H. Carter Edwards¹, Christian R. Trott¹, and
Fernanda Foertter²

¹Sandia National Laboratories

²Oak Ridge National Laboratory

Supercomputing, Nov 12-17, 2017

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2017-11593 C

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

Knowledge of C++: class constructors, member variables, member functions, member operators, template arguments

Using NVIDIA's NVLABS

- ▶ Kokkos pre installed in $\${HOME}/kokkos$
- ▶ Exercises pre installed in $\${HOME}/SC2017$

Using your own $\${HOME}$

- ▶ 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-tutorials into $\${HOME}/kokkos-tutorials$

Exercises are in $\${HOME}/kokkos-tutorials/Intro-Short/SC2017$

Exercises' makefiles look for $\${HOME}/kokkos$

Understand Kokkos Programming Model Abstractions

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

Kokkos' basic capabilities covered today:

- ▶ Simple 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:

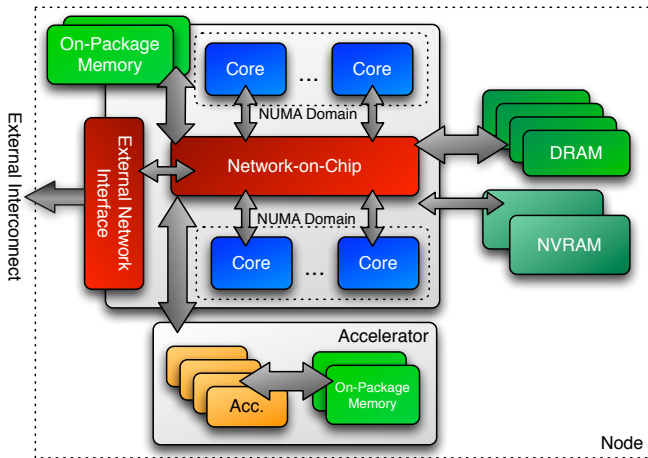
- ▶ Thread safety, thread scalability, and atomic operations
- ▶ Hierarchical patterns for maximizing parallelism
- ▶ Dynamic directed acyclic graph of tasks pattern
- ▶ Numerous *plugin* points for extensibility

- ▶ 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.
 - ▶ *full day tutorial only*

Assume you are here because:

- ▶ Want to use **all** HPC node architectures; including GPUs
- ▶ Are familiar with **C++**
- ▶ Are familiar with **data parallelism**
- ▶ A little familiar with **OpenMP**
- ▶ A little familiar with **NVIDIA GPU architecture**
 Spoiler alert: memory access patterns are important
- ▶ Want GPU programming to be **easier**
- ▶ Would like **portability**, as long as it doesn't hurt performance

Target machine:



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.

Concepts for threaded data parallelism

Learning objectives:

- ▶ Terminology of pattern, policy, and body.
- ▶ The data layout problem.

```
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;  
}
```

Pattern

Policy

Body

```
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 loop?

```
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;  
}
```

What if we want to **thread** the loop?

```
#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 loop?

```
#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

```
#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

```
#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

```
#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.

Problem: 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.  
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

```
#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 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**

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 **work** to cores

Data parallel patterns and work

```
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

```
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:

```
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,

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

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 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,

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

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

```
struct Functor {  
    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?

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

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

How is data passed to computational bodies?

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

```
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:

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

Putting it all together: the complete functor:

```
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

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

Putting it all together: the complete functor:

```

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

```

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

```

Functor

```

AtomForceFunctor functor(atomForces, data);
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex){
    functor(atomIndex);
}

```

The complete picture (using functors):

1. Defining the functor (operator+data):

```
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:

```
AtomForceFunctor functor(atomForces, data);  
Kokkos::parallel_for(numberOfAtoms, functor);
```

Functors are tedious \Rightarrow **C++11 Lambdas** are concise

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

Functors are tedious \Rightarrow **C++11 Lambdas** are concise

```
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 \Rightarrow **C++11 Lambdas** are concise

```
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

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

OpenMP

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

Kokkos

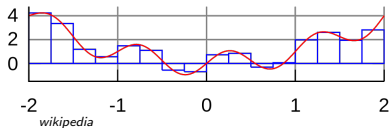
```
parallel_for(N, [=] (const size_t i) {  
    /* loop body */  
});
```

Important concept

Simple Kokkos usage is **no more conceptually difficult** than OpenMP, the annotations just go in different places.

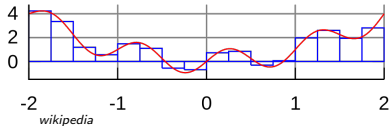
Riemann-sum-style numerical integration:

$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



Riemann-sum-style numerical integration:

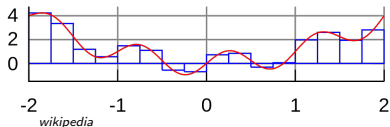
$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



```
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:

$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$

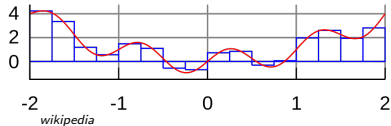


```
double totalIntegral = 0;
for (size_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

How would we **parallelize** it?

Riemann-sum-style numerical integration:

$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



Pattern?

```
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;
```

Policy?

Body?

How would we **parallelize** it?

An (incorrect) attempt:

```
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

step	thread 0	thread 1
0	load	
1	increment	load
2	write	increment
3		write

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 += ...
}
```

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

OpenMP

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

Kokkos

```
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.

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: $\text{Time} = \alpha + \frac{\beta * N}{P}$

- ▶ α = dispatch overhead
- ▶ β = 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: $\text{Time} = \alpha + \frac{\beta * N}{P}$

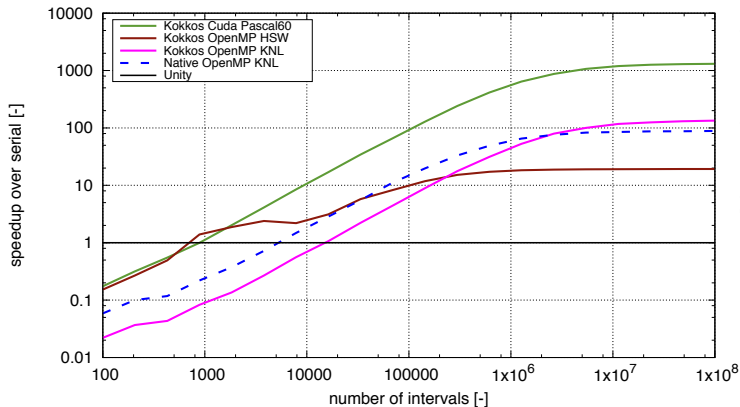
- ▶ α = dispatch overhead
- ▶ β = time for a unit of work
- ▶ N = number of units of work
- ▶ P = available concurrency

$$\text{Speedup} = P \div \left(1 + \frac{\alpha * P}{\beta * N}\right)$$

- ▶ Should have $\alpha * P \ll \beta * N$
- ▶ All runtimes strive to minimize launch overhead α
- ▶ Find more parallelism to increase N
- ▶ Merge (fuse) parallel operations to increase β

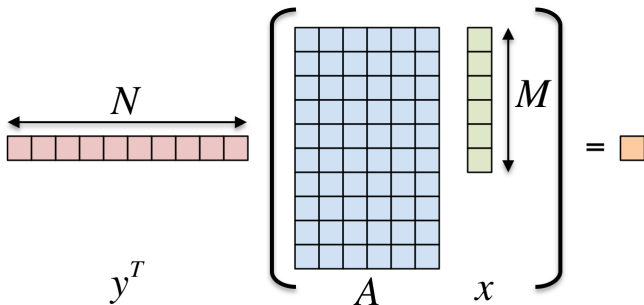
Results: illustrates simple speedup model = $P \div \left(1 + \frac{\alpha * P}{\beta * N}\right)$

Kokkos speedup over serial: Scalar Integration



Note: log scale

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



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:

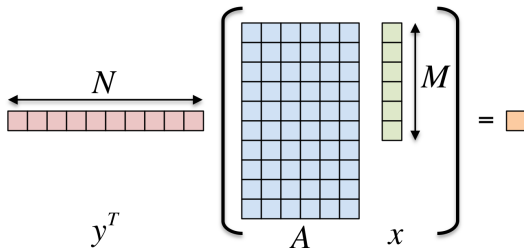
```
#include <Kokkos_Core.hpp>
int main(int argc, char** argv) {
    /* ... do any necessary setup (e.g., initialize MPI) ... */
    Kokkos::initialize(argc, argv);
    /* ... do computations ... */
    Kokkos::finalize();
    return 0;
}
```

(Optional) Command-line arguments:

--kokkos-threads=INT	total number of threads (or threads within NUMA region)
--kokkos-numa=INT	number of NUMA regions
--kokkos-device=INT	device (GPU) ID to use

Exercise #1: Inner Product, Flat Parallelism on the CPU

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



Details:

- ▶ Location: SC2017/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  
make -j [KOKKOS_DEVICES=OpenMP,Serial]
```

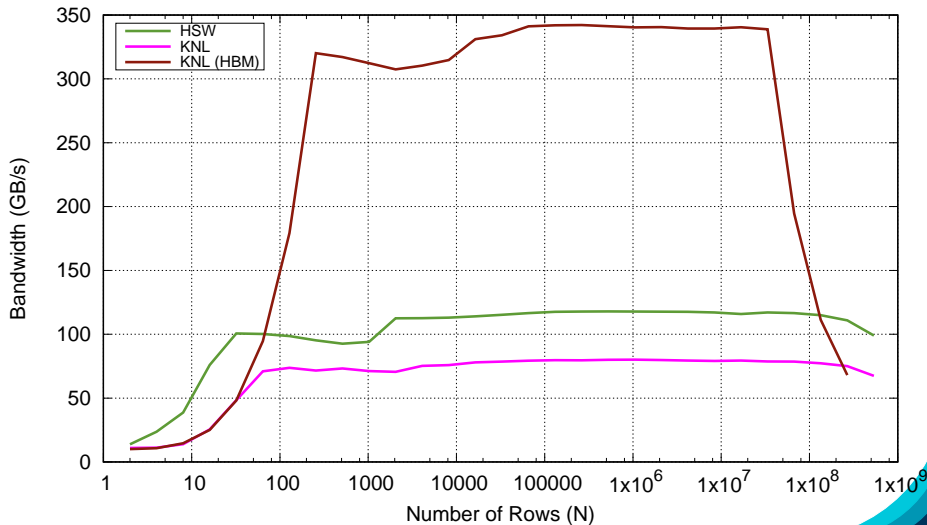
Running on CPU with OpenMP back-end

```
# Set OpenMP affinity  
export OMP_NUM_THREADS=8  
export GOMP_CPU_AFFINITY=0-8  
# 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
- ▶ Vary number of rows (-N ...)

<y,Ax> Exercise 01, Fixed Size



- ▶ 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

- ▶ **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**, 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:

Lambda

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

Functor

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

Example: running daxpy on the GPU:

Lambda

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

Functor

```
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:

Lambda

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

Functor

```
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:

```
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:

```
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`

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:

```
View<double*> a("a", NO), b("b", NO);  
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:

```
View<double*> a("a", NO), b("b", NO);  
a = b;  
View<double*> c(b);  
a(0) = 1;  
b(0) = 2;  
c(0) = 3;  
print a(0)
```

What gets printed?
3.0

Exercise #2: Inner Product, Flat Parallelism on the CPU, with Views

- ▶ Location: SC2017/Exercises/03/Begin/
- ▶ Assignment: Change data storage from arrays to Views.
- ▶ Compile and run on CPU, and then on GPU with UVM

```
make -j KOKKOS_DEVICES=OpenMP # CPU-only using OpenMP
make -j KOKKOS_DEVICES=Cuda \
    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

- ▶ **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 data movement.
- ▶ The need for `Kokkos::initialize` and `finalize`.
- ▶ Where to use Kokkos annotation macros for portability.

Thought experiment: Consider this code:

section 1
section 2

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

Thought experiment: Consider this code:

```
section 1 MPI_Reduce(...);  
          FILE * file = fopen(...);  
          runANormalFunction(...data...);  
section 2 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:

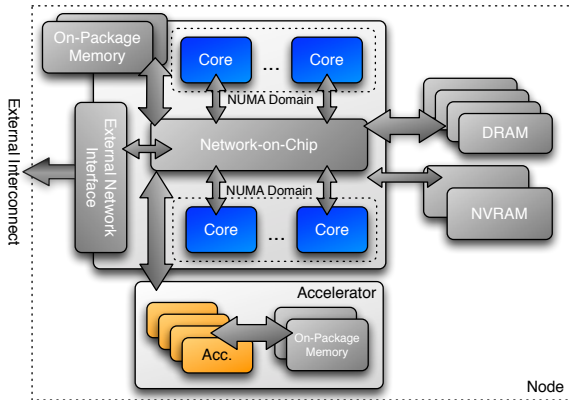
```
section 1 MPI_Reduce(...);  
          FILE * file = fopen(...);  
          runANormalFunction(...data...);  
section 2 Kokkos::parallel_for(numberOfSomethings ,  
                               [=] (const size_t somethingIndex) {  
                                   const double y = ...;  
                                   // 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, ...

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

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

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

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

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

- ▶ 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 ... */  
    });
```

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.

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

```
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)

```
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<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

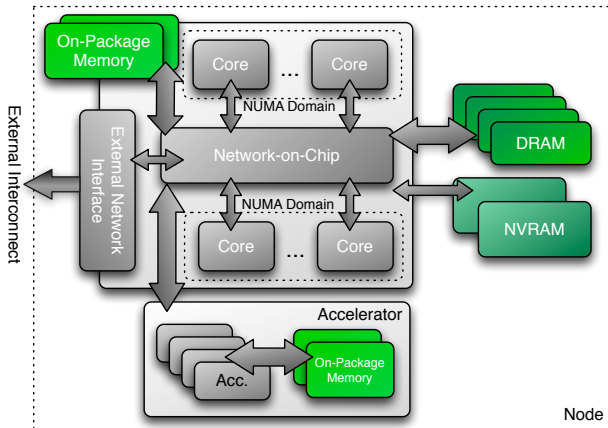
```
View<double*> data("data", size);
for (size_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

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(...);`

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`

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

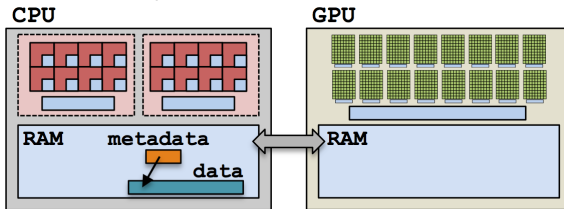
Important concept: Memory spaces

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**.

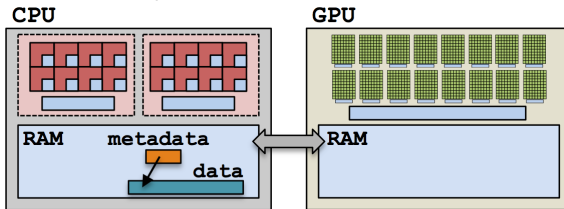
Example: HostSpace

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



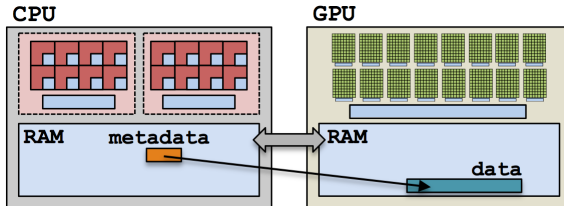
Example: HostSpace

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



Example: CudaSpace

```
View<double**, CudaSpace> view(...constructor arguments...);
```



Anatomy of a kernel launch:

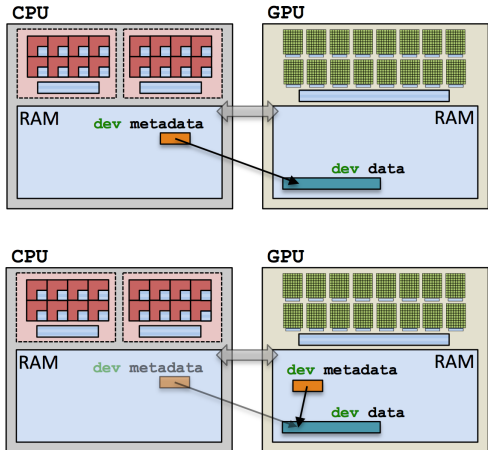
1. User declares views, allocating.
2. User instantiates a functor with views.
3. User launches `parallel_something`:
 - ▶ Functor is copied to the device.
 - ▶ Kernel is run.
 - ▶ Copy of functor on the device is released.

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

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) = ...;
  });
```

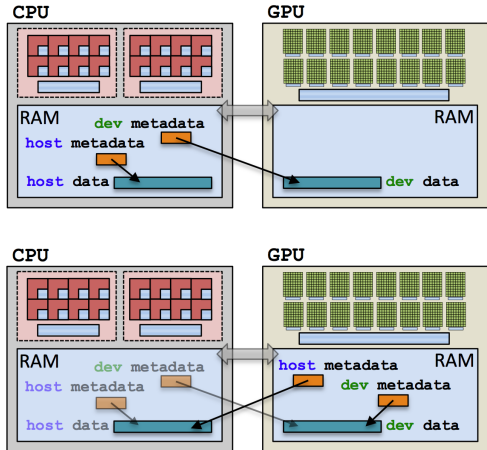


Example: two views

```

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

```

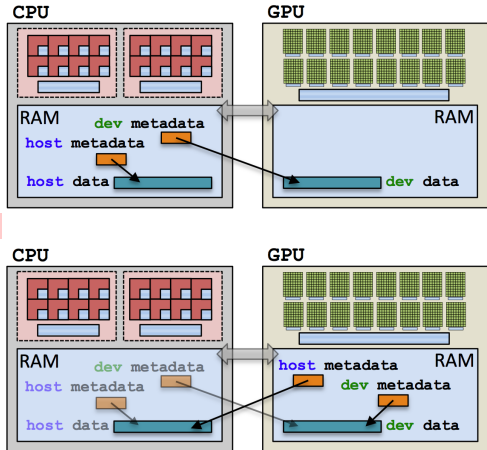


Example: two views

```

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 lives in CudaSpace

```
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

```
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);
```

fault

Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

```
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: View lives in HostSpace

```
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

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

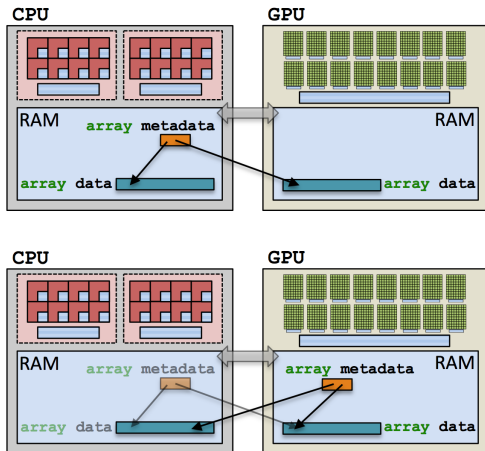
double sum = 0;
Kokkos::parallel_reduce(
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const size_t index, double & valueToUpdate) {
        valueToUpdate += array(index);      illegal access
    },
    sum);
```

What's the solution?

- ▶ CudaUVMSpace
- ▶ CudaHostPinnedSpace (skipping)
- ▶ 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 a **performance hit**.

Important concept: Mirrors

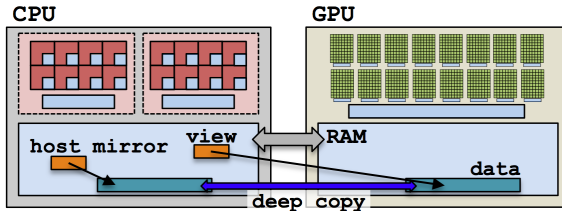
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

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.

```
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.

```
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);
```

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

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);
```

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

4. **Deep copy** **hostView**'s array to **view**'s array.

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

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);
```

3. **Populate** **hostView** on the host (from file, etc.).
4. **Deep copy** **hostView**'s array to **view**'s array.
`Kokkos::deep_copy(view, hostView);`

5. **Launch** a kernel processing the **view**'s array.

```
Kokkos::parallel_for(  
    RangePolicy< Space>(0, size),  
    KOKKOS_LAMBDA (...) { use and change 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);
```

3. **Populate** **hostView** on the host (from file, etc.).
4. **Deep copy** **hostView**'s array to **view**'s array.

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

5. **Launch** a kernel processing the **view**'s array.

```
Kokkos::parallel_for(  
    RangePolicy< Space>(0, size),  
    KOKKOS_LAMBDA (...) { use and change view });
```

6. If needed, **deep copy** the **view**'s updated array back to the **hostView**'s array to write file, etc.

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

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

```
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**.

Exercise #3: Flat Parallelism on the GPU, Views and Host Mirrors

Details:

- ▶ Location: SC2017/Exercises/03/Begin/
- ▶ Add HostMirror Views and deep copy
- ▶ Make sure you use the correct view in initialization and Kernel

```
# 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

- ▶ 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

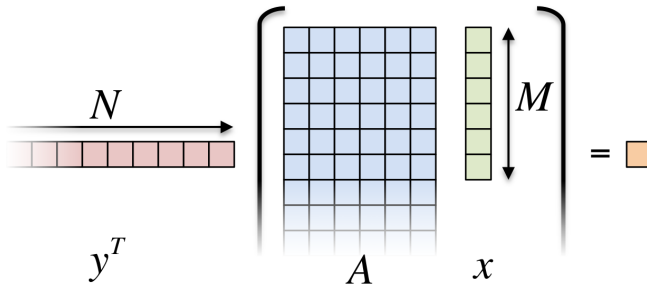
Learning objectives:

- ▶ 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);

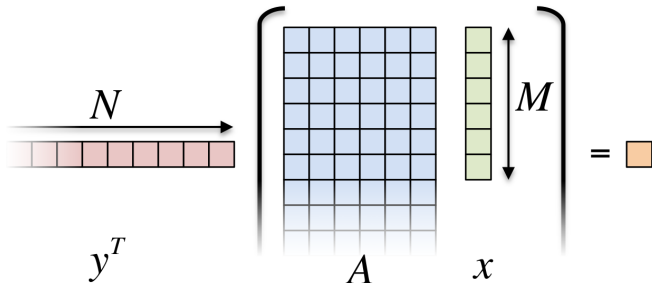
```



```

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);

```

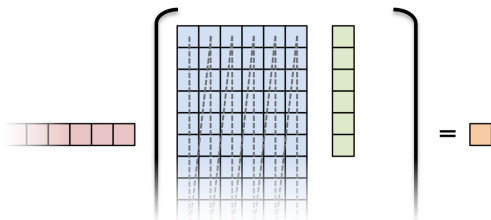


Driving question: How should A be laid out in memory?

Layout is the mapping of multi-index to memory:

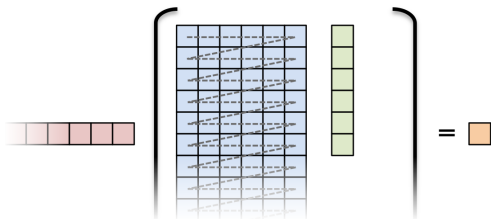
LayoutLeft

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`, ...

Details:

- ▶ Location: SC2017/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

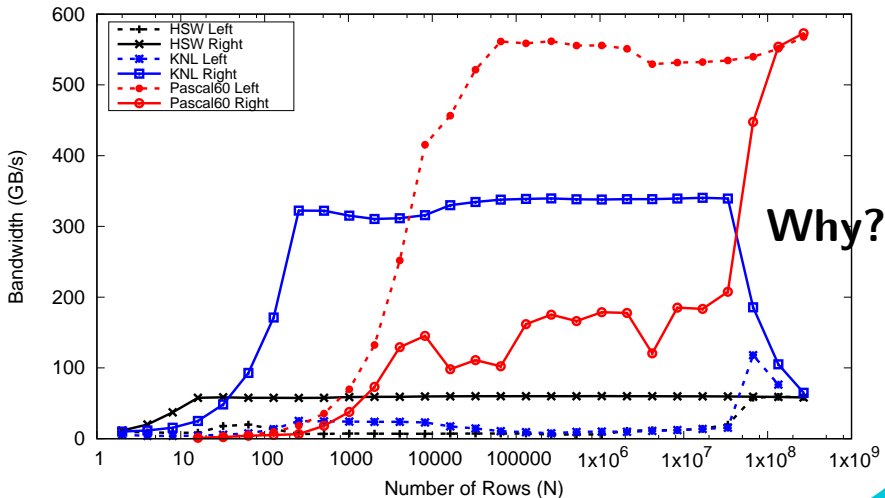
Things to try:

- ▶ 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



Thread independence:

```
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:

```
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:

```
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:

```
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 finish their loads before *any* thread can move on.

Thread independence:

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

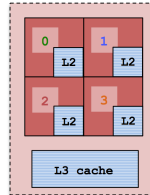
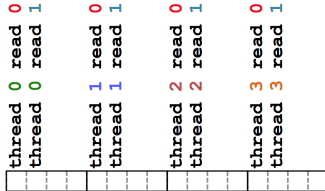
Question: once a thread reads *d*, does it need to wait?

- ▶ **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.

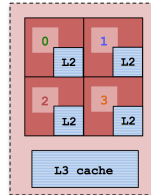
In particular, all threads in a group (*warp*) must finished their loads before *any* thread can move on.

So, **how many cache lines** must be fetched before threads can move on?

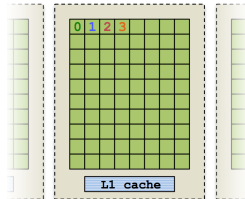
CPUs: few (independent) cores with separate caches:



CPUs: few (independent) cores with separate caches:



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 (more later).

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)?

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:

```
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?

Iterating for the execution space:

```
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:

```
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:

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

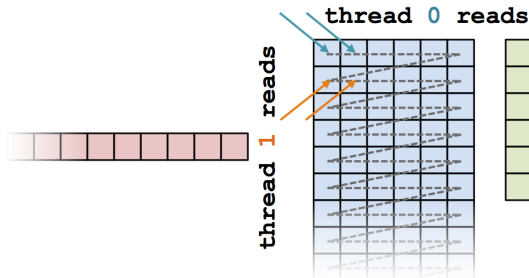
Important point

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

Important point

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

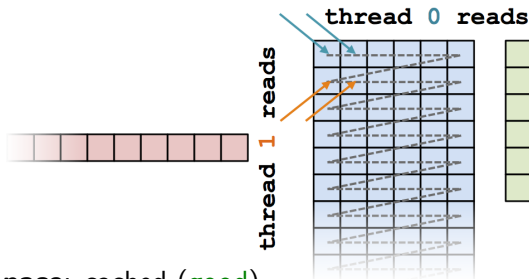
Analysis: row-major (LayoutRight)



Important point

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

Analysis: row-major (LayoutRight)

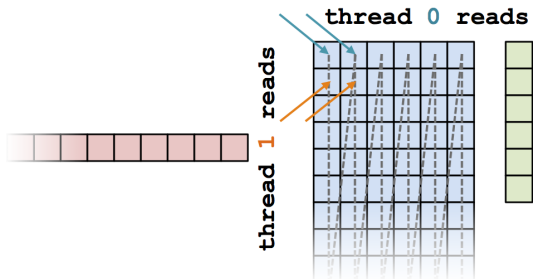


- ▶ **HostSpace**: cached (good)
- ▶ **CudaSpace**: uncoalesced (bad)

Important point

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

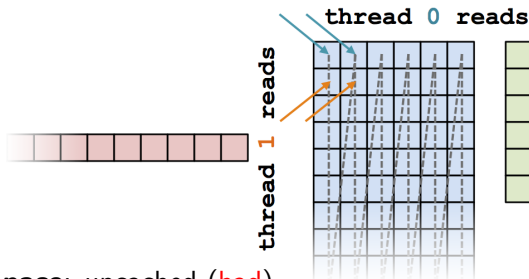
Analysis: column-major (LayoutLeft)



Important point

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

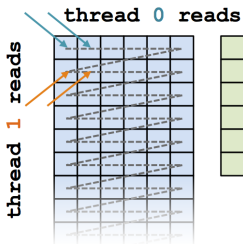
Analysis: column-major (LayoutLeft)



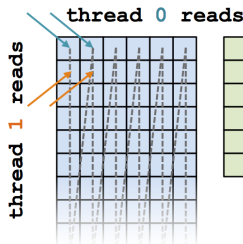
- ▶ **HostSpace**: uncached (**bad**)
- ▶ **CudaSpace**: coalesced (**good**)

Analysis: Kokkos architecture-dependent

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



(a) OpenMP

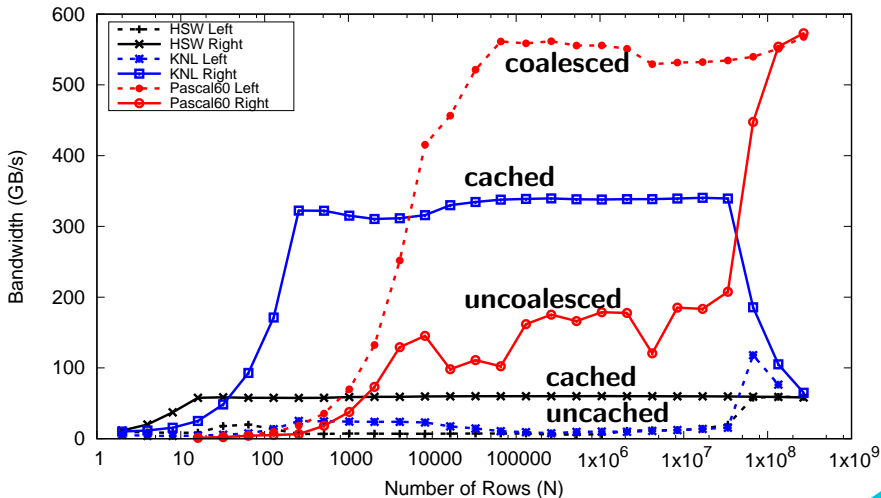


(b) Cuda

- **HostSpace**: cached (good)
- **CudaSpace**: coalesced (good)

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

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



- ▶ 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.

Kokkos capabilities NOT covered today, only 2 hours

- ▶ Atomic operations and their scalability
- ▶ Multidimensional range policy (heirarchical pattern)
- ▶ Thread-team policy (hierarchical pattern) with intra-team shared memory
 - CUDA grid-block parallelism, but easier and portable
- ▶ Dynamic directed acyclic graph (DAG) of tasks pattern
- ▶ Plugging in customized multidimensional array data layout
e.g., arbitrarily strided, heirarchical tiling

- ▶ 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.
 - ▶ *full day tutorial only*