

# Kokkos Tutorial

H. Carter Edwards <sup>1</sup>, Christian R. Trott <sup>1</sup>, Jeff Amelang <sup>2</sup>

<sup>1</sup>Sandia National Laboratories

<sup>2</sup>Google

Supercomputing'16, November 13, 2016

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.

SAND2016-8015 C

## SOFTWARE FOR LAB

### Remote Desktop Software:

- ▶ Download NoMachine now for best performance from  
[www.nomachine.com/download](http://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](http://www.putty.org)
- ▶ Alternatively you may use a provided browser-based SSH option

## CONNECTION INSTRUCTIONS

- ▶ Navigate to [nvlabs.qwiklab.com](https://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 Edwards <sup>1</sup>, Christian R. Trott <sup>1</sup>, Jeff Amelang <sup>2</sup>

<sup>1</sup>Sandia National Laboratories

<sup>2</sup>Google

Supercomputing'16, November 13, 2016

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.

SAND2016-8015 C

**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}/SC2016

## Using your own \${HOME}

- ▶ Git
- ▶ GCC 4.8.4 (or newer) OR Intel 14 (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](https://github.com/kokkos/kokkos) into \${HOME}/kokkos
- ▶ clone [github.com/kokkos/kokkos-tutorials/SC2016](https://github.com/kokkos/kokkos-tutorials/SC2016) into \${HOME}/SC2016
  - makefiles look for \${HOME}/kokkos*

## 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

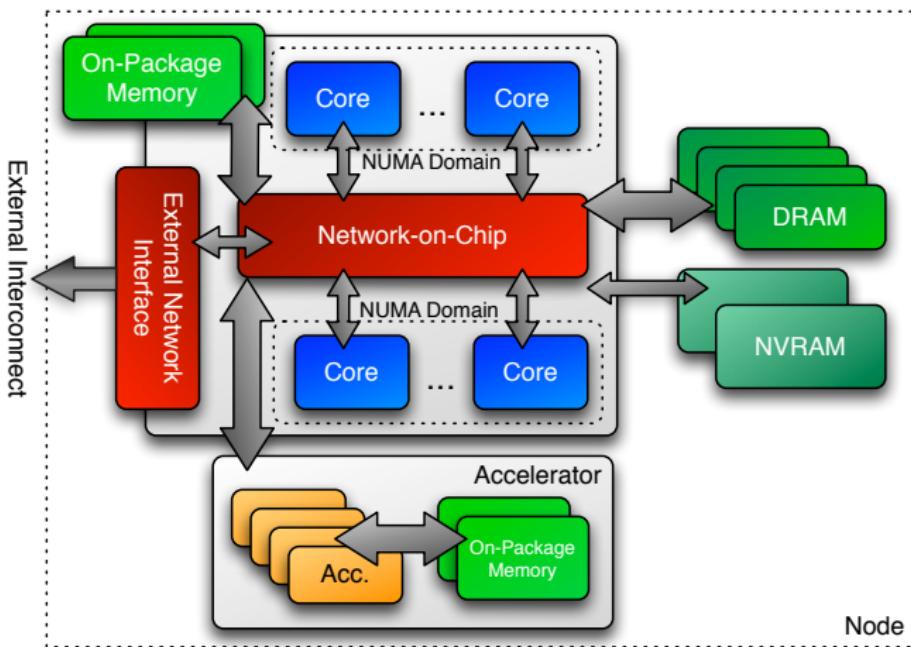
### Part Two:

- ▶ 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.*

**Target audience:**

- ▶ **Wants** to use GPUs (and perhaps other accelerators)
- ▶ Is familiar with **data parallelism**
- ▶ **Has taken** “cuda programming 101”
- ▶ Familiar with **NVIDIA GPU architecture** at a *high* level
  - Aware that **coalesced** memory access is important
- ▶ Some familiarity with **OpenMP**
- ▶ Wants CUDA to be **easier**
- ▶ Would like **portability**, if 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**

```
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;  
}
```

**Body****Policy**

## 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;
}
```

## 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.

## 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 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

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

Kokkos maps **work** to cores

## Data parallel patterns and work

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

Functor

```
AtomForceFunctor functor(atomForces, data);
for (atomIndex = 0; atomIndex < numberAtoms; ++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 verbose ⇒ 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 verbose ⇒ 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 verbose ⇒ 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 (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

```
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

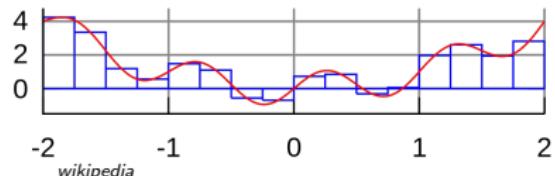
```
parallel_for(N, [=] (const size_t i) {  
    /* loop body */  
});
```

### Important concept

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

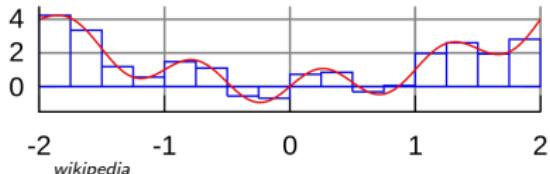
## Riemann-sum-style numerical integration:

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



## Riemann-sum-style numerical integration:

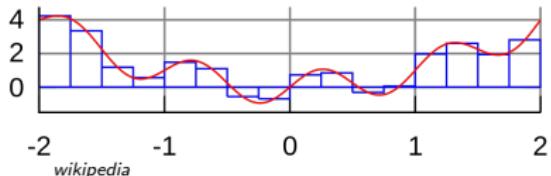
$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



```
double totalIntegral = 0;
for (size_t i = 0; i < number0fIntervals; ++i) {
    const double x =
        lower + (i/number0fIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

## Riemann-sum-style numerical integration:

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

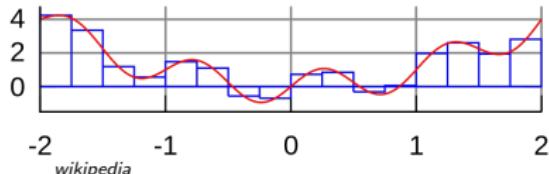


```
double totalIntegral = 0;
for (size_t i = 0; i < number0fIntervals; ++i) {
    const double x =
        lower + (i/number0fIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

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 < number0fIntervals; ++i) {
    const double x =
        lower + (i/number0fIntervals) * (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);
```

```
double totalIntegral = 0;
#pragma omp parallel for reduction(+:totalIntegral)
for (size_t i = 0; i < number0fIntervals; ++i) {
    totalIntegral += function(...);
}
```

```
double totalIntegral = 0;
parallel_reduce(number0fIntervals,
 [=] (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 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:  $\text{Time} = \alpha + \frac{\beta * N}{P}$

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

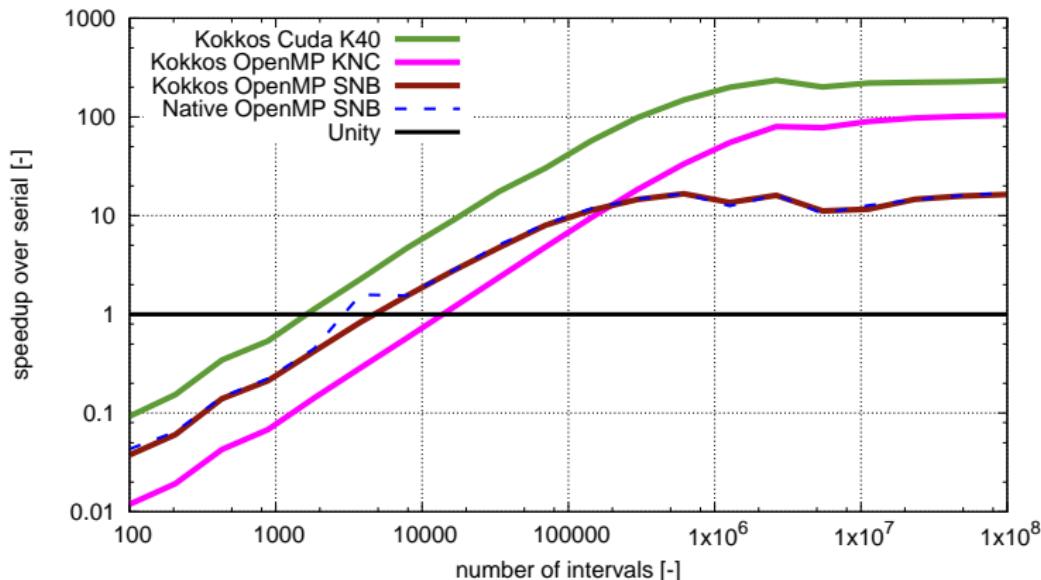
- ▶  $\alpha$  = dispatch overhead
- ▶  $\beta$  = time for a unit of work
- ▶  $N$  = number of units of work
- ▶  $P$  = available concurrency

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

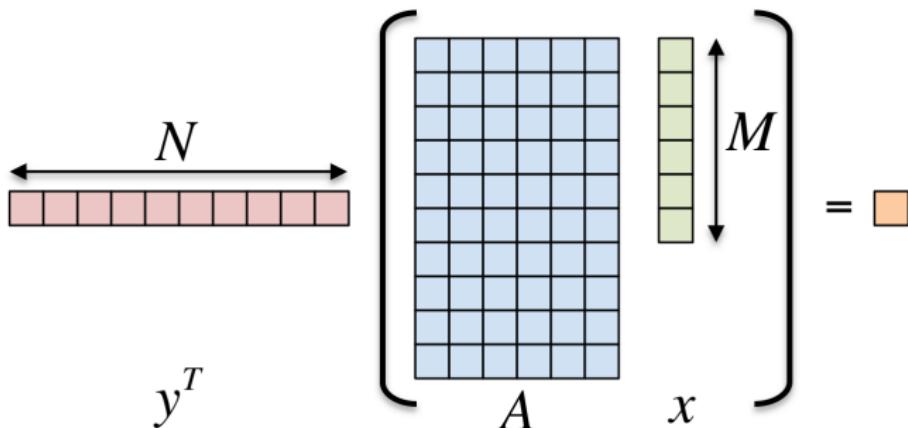
- ▶ Should have  $\alpha * P \ll \beta * N$
- ▶ All runtimes strive to minimize launch overhead  $\alpha$
- ▶ Find more parallelism to increase  $N$
- ▶ Merge (fuse) parallel operations to increase  $\beta$

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

Kokkos speedup over serial: Scalar Integration



**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

## Exercise #1: include, initialize, finalize Kokkos

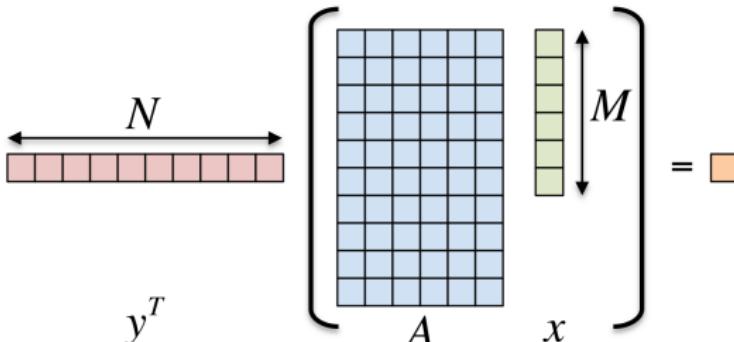
The **first step** in using Kokkos is to include, initialize, and finalize:

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



**Details:**

$$y^T$$

- ▶ Location: `~/SC2016/Exercises/01/`
- ▶ 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

```
cd ~/SC2016/Exercises/01/Begin  
# gcc using OpenMP (default) and Serial back-ends  
make -j [KOKKOS_DEVICES=OpenMP,Serial]
```

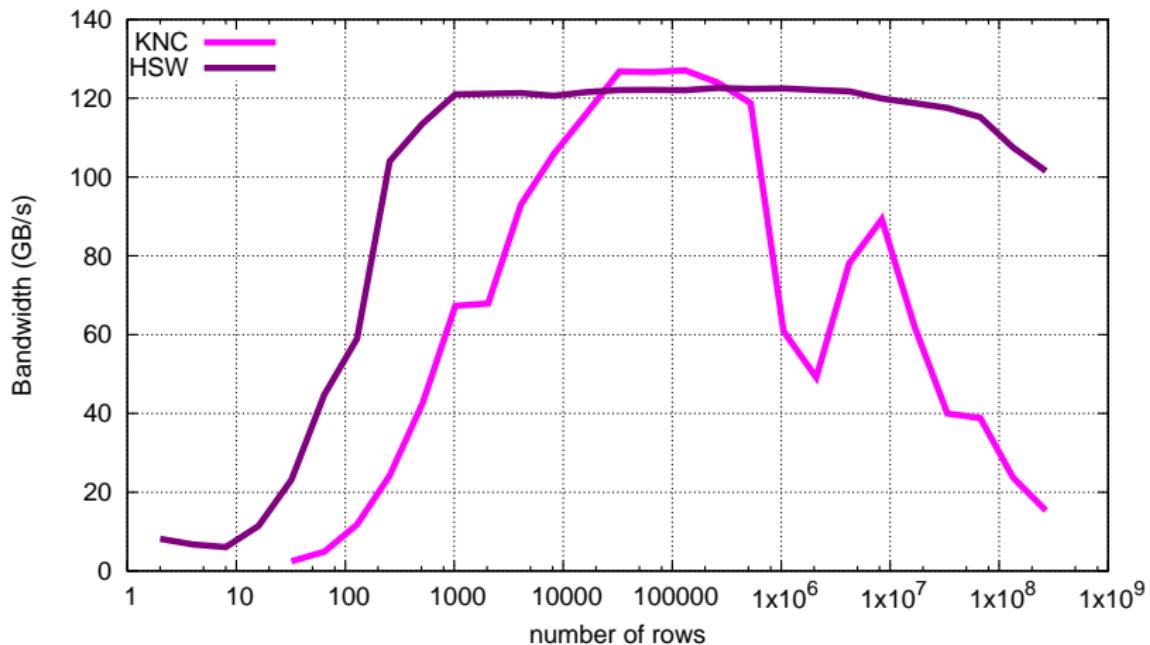
## Running on CPU with OpenMP back-end

```
# 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 ...)

## &lt;y,Ax&gt; Exercise01, fixed problem size



- ▶ Customized reduction type and operator
- ▶ Exclusive and inclusive **prefix scan** with the parallel\_scan pattern.
- ▶ Using *tag dispatch* interface to allow non-trivial functors to have multiple “operator()” functions.
- ▶ **NEW:** Directed acyclic graph (DAG) of tasks pattern.
- ▶ **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**, 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 when passing.

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

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

- ▶ Location: ~/SC2016/Exercises/02/
- ▶ 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:

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

Kokkos::parallel_for(numberOfSomethings,
                     [=] (const size_t somethingIndex) {
    const double y = ...;
    // do something interesting
});
);
```

section 1

section 2

**Thought experiment:** Consider this code:

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

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

```

section 2

- ▶ 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:

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

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

```

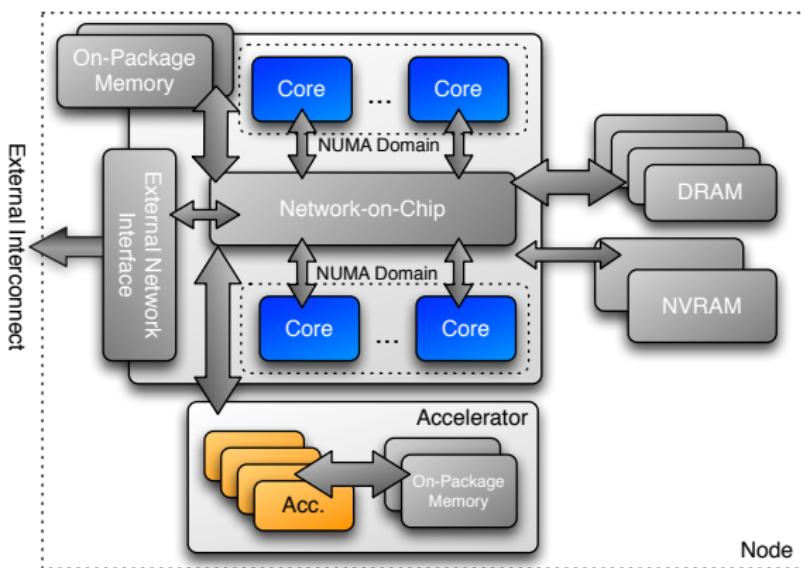
section 2

- ▶ 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

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

Kokkos::parallel_for(numberOfSomethings,
                     [=] (const size_t somethingIndex) {
    const double y = ...;
    // do something interesting
}
);
```

Parallel

Host

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

Kokkos::parallel_for(numberOfSomethings,
                     [=] (const size_t somethingIndex) {
    const double y = ...;
    // do something interesting
}
);
```

Parallel

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

Host

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

Parallel

- ▶ 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 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**
- ▶ 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 ... */
    });
}
```

## 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 7.5)

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

// Where kokkos defines:
#define KOKKOS_LAMBDA [=] /* #if CPU-only */
#define KOKKOS_LAMBDA [=] __device__ __host__ /* #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

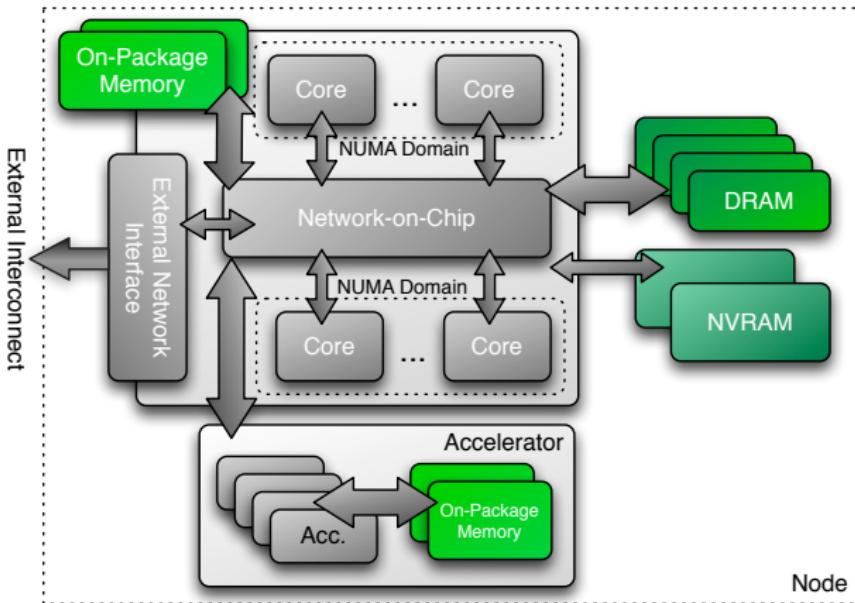
```
View<double*> data("data", size);
for (size_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

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 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

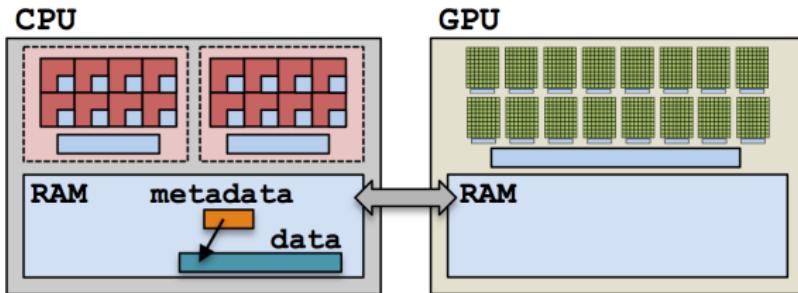
## Important concept: Memory spaces

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**.

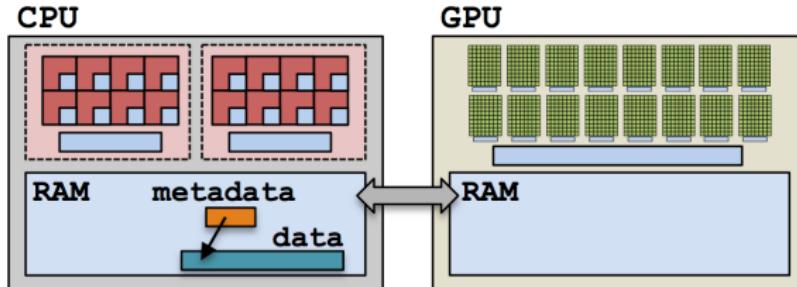
## Example: HostSpace

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



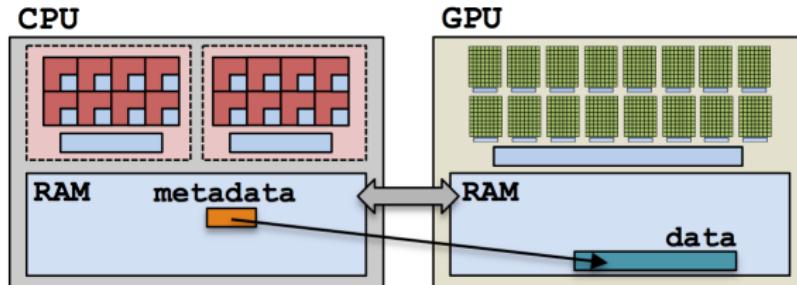
## Example: HostSpace

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



## Example: CudaSpace

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



## Anatomy of a kernel launch:

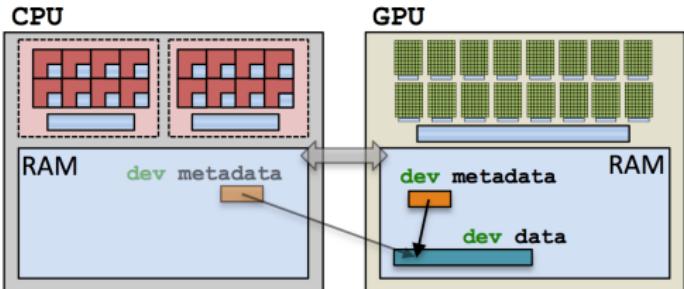
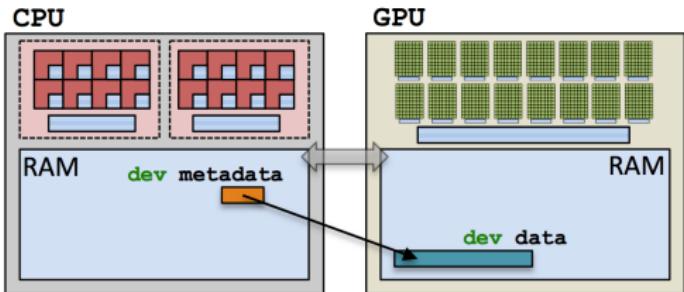
1. User declares views, allocating.
2. User instantiates a functor with views.
3. User launches parallel\_something:
  - ▶ Functor is copied to the device.
  - ▶ Kernel is run.
  - ▶ Copy of functor on the device is released.

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

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

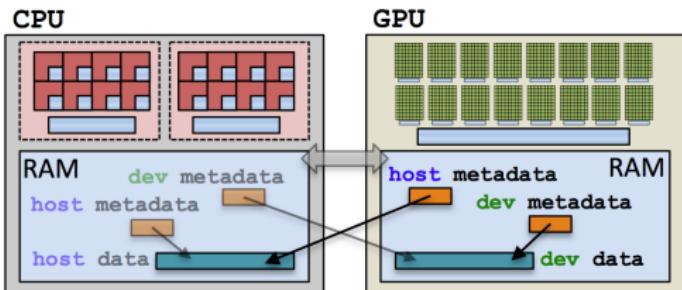
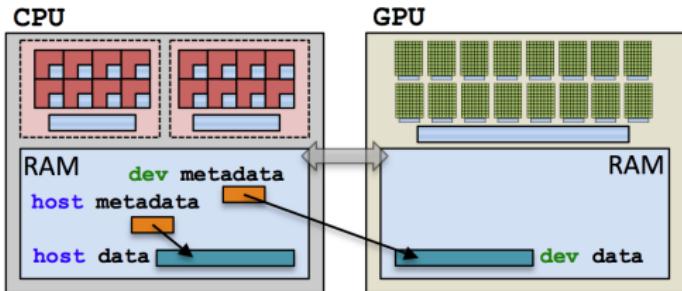
## Example: one view

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



## Example: two views

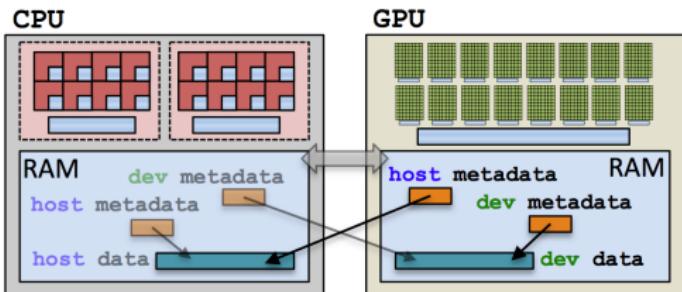
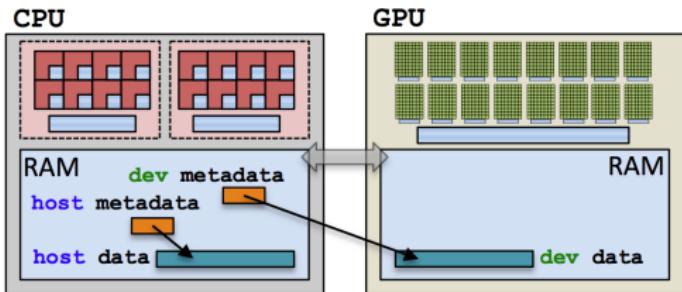
```
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for(N,
 [=] (int i) {
    dev(i) = ...;
    host(i) = ...;
});
```



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

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<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<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<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?

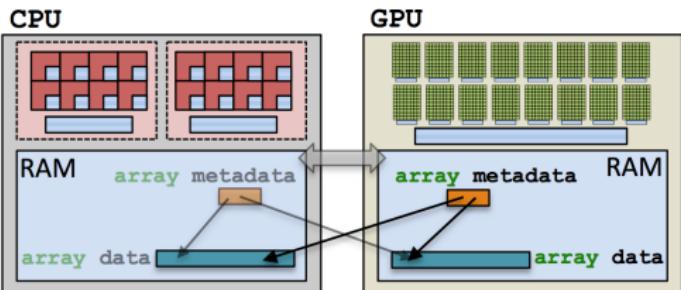
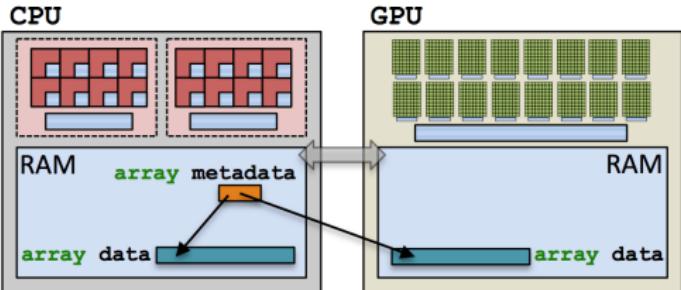
- ▶ CudaUVMSpace
- ▶ CudaHostPinnedSpace (skipping)
- ▶ Mirroring

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

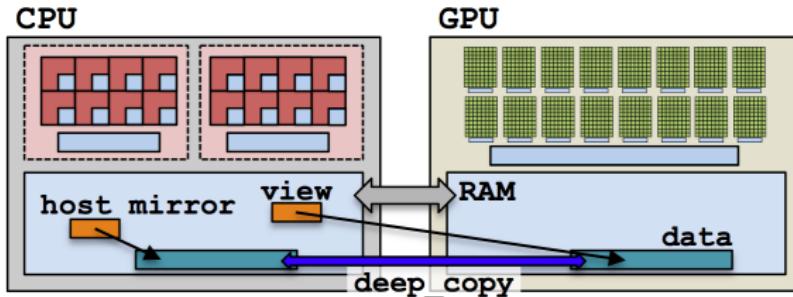
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

## 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.
- ▶ Kokkos *never* performs a **hidden deep copy**.

### Details:

- ▶ Location: ~ /SC2016/Exercises/03/
- ▶ 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.

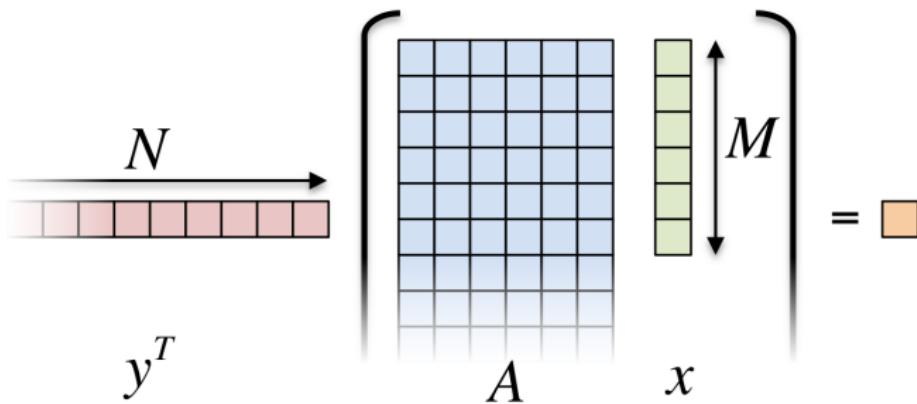
# Managing memory access patterns for performance portability

## 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.

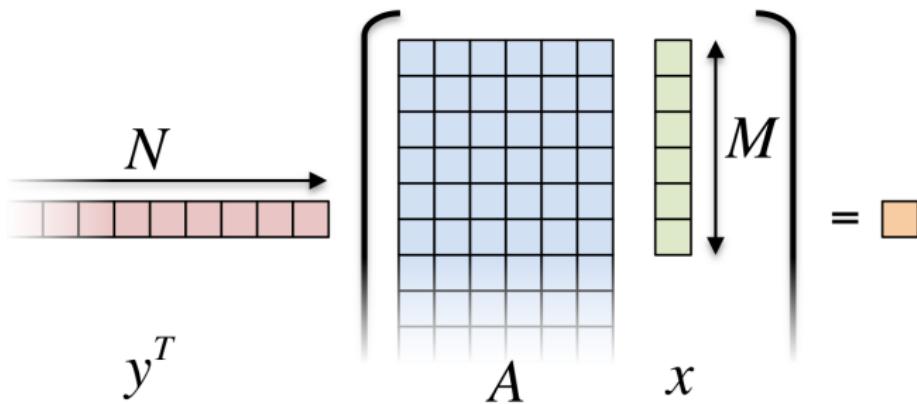
## Example: inner product (0)

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



## Example: inner product (0)

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

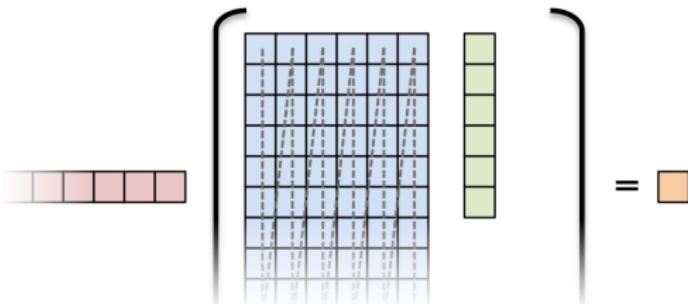


How should  $A$  be laid out in memory?

Layout is the mapping of multi-index to memory:

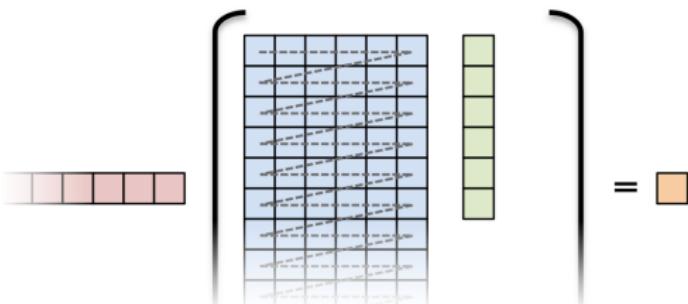
### LayoutLeft

in 2D, “column-major”



### LayoutRight

in 2D, “row-major”



## Important concept: Layout

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

```
View<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:

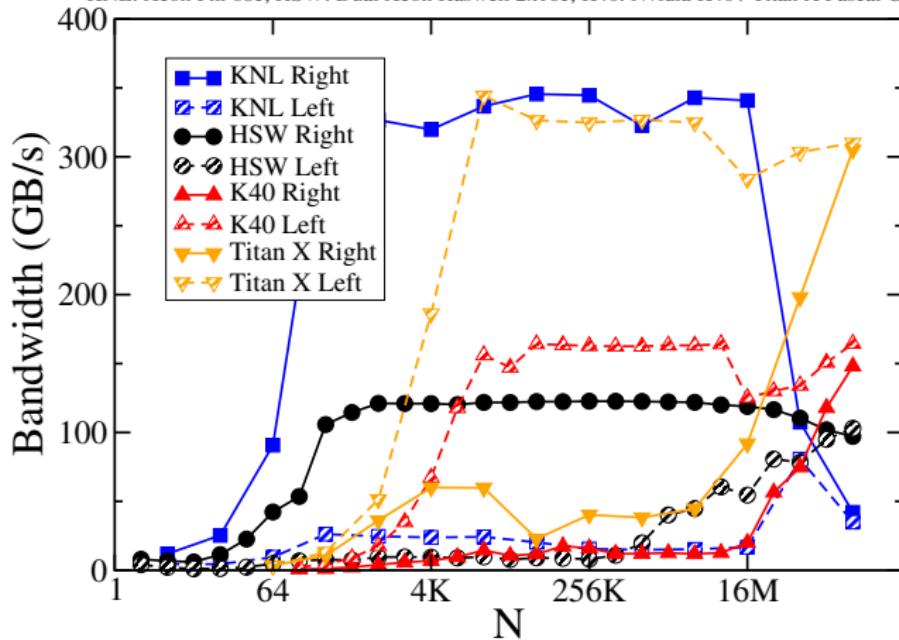
- ▶ Location: ~/SC2016/Exercises/04/
- ▶ Replace ‘‘N’’ in parallel dispatch with RangePolicy<ExecSpace>
- ▶ Add MemSpace to all Views and Layout to A
- ▶ Experiment with the combinations of ExecSpace, Layout to view performance

### 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.

## <ylAx> Exercise 04 (Layout)

KNL: Xeon Phi 68c; HSW: Dual Xeon Haswell 2x16c; K40: Nvidia K40 / Titan X Pascal GPU



Why?

## 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;  
}
```

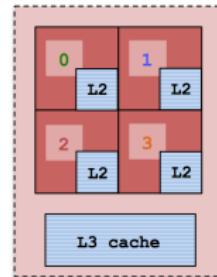
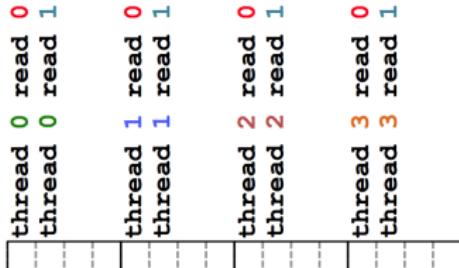
Question: once a thread reads  $d$ , does it need to wait?

- ▶ **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.

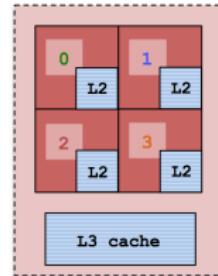
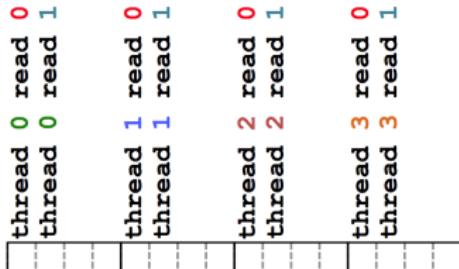
In particular, all threads in a group (*warp*) must finished their loads before *any* thread can move on.

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

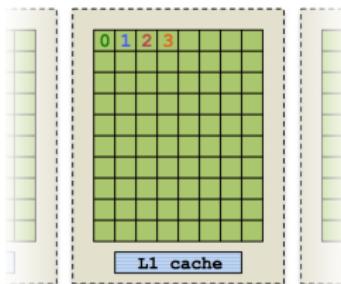
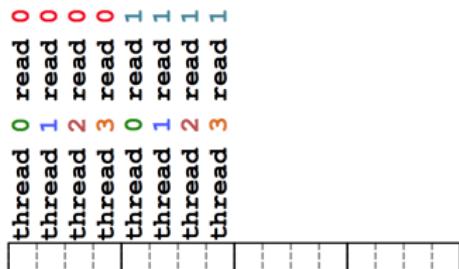
**CPUs:** few (independent) cores with separate caches:



**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.

## 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.

### Important point

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

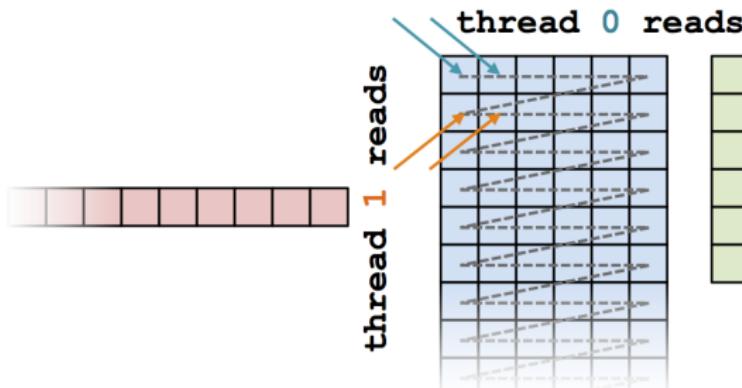
## Important point

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

## Important point

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

### Analysis: row-major (LayoutRight)



## Important point

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

### Analysis: row-major (LayoutRight)

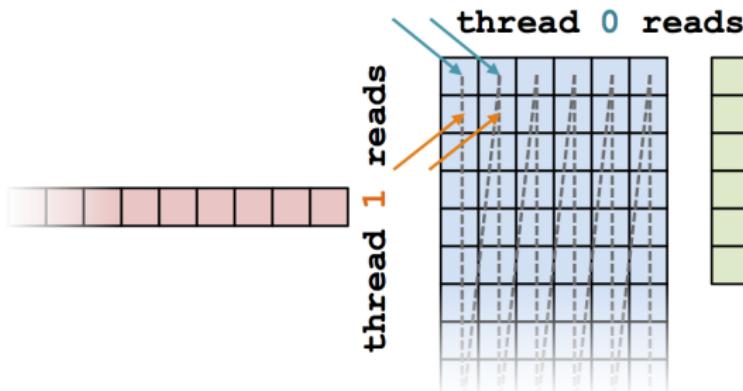


- ▶ **HostSpace:** cached (good)
- ▶ **CudaSpace:** uncoalesced (bad)

## Important point

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

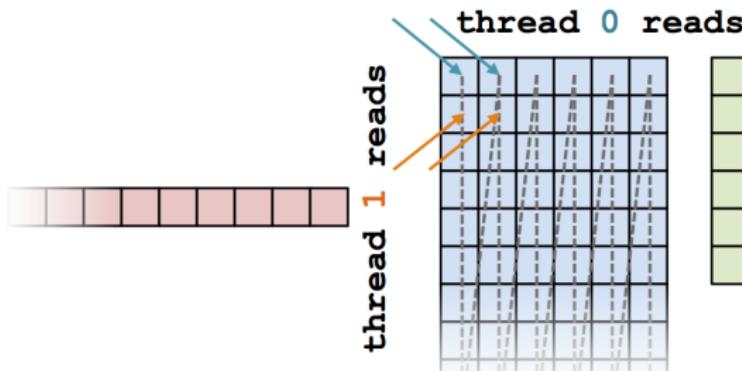
### Analysis: column-major (LayoutLeft)



## Important point

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

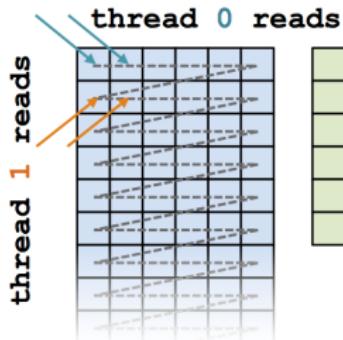
### Analysis: column-major (LayoutLeft)



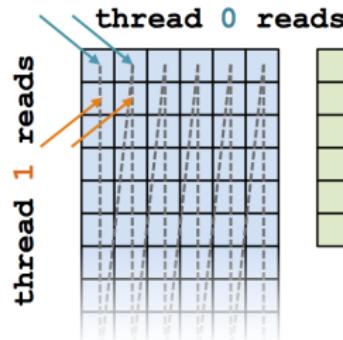
- ▶ **HostSpace:** uncached (**bad**)
- ▶ **CudaSpace:** coalesced (**good**)

## Analysis: Kokkos architecture-dependent

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



(a) OpenMP

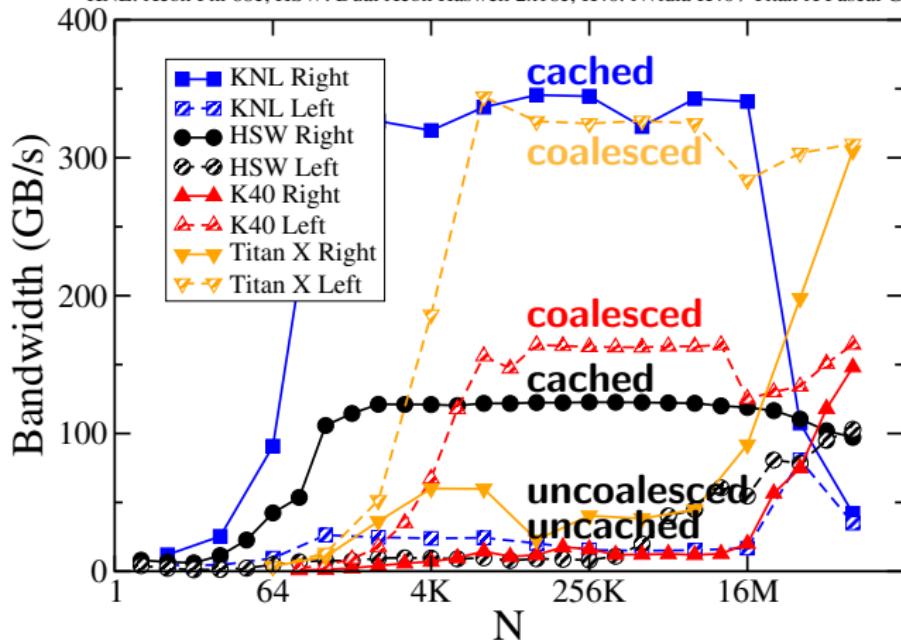


(b) Cuda

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

## &lt;ylAx&gt; Exercise 04 (Layout)

KNL: Xeon Phi 68c; HSW: Dual Xeon Haswell 2x16c; K40: Nvidia K40 / Titan X Pascal 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.

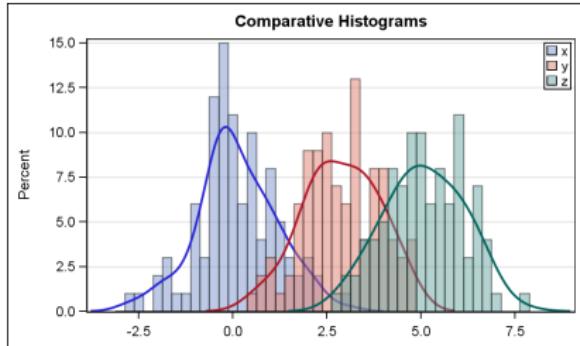
# Thread safety and atomic operations

## 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.

## Histogram kernel:

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

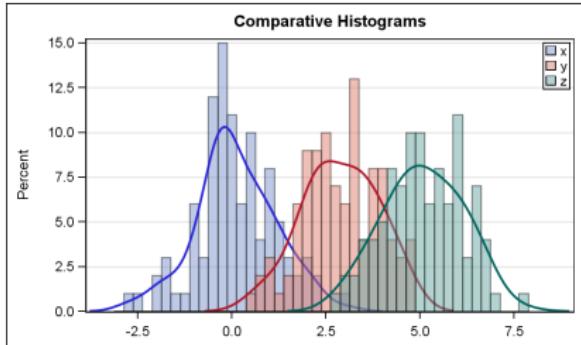


<http://www.farmaceuticas.com.br/tag/graficos/>

## Histogram kernel:

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {  
    const Something 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/>

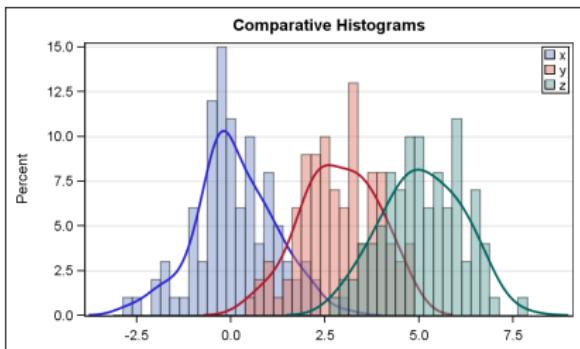
## Histogram kernel:

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

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

## Solution strategies:

- ▶ Locks: not feasible on GPU
- ▶ Thread-private copies:  
not thread-scalable
- ▶ Atomics



<http://www.farmaceuticas.com.br/tag/graficos/>

## Atomics: the portable and thread-scalable solution

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

## Atomics: the portable and thread-scalable solution

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

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

## Atomics: the portable and thread-scalable solution

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {  
    const Something 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

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

## How expensive are atomics?

Thought experiment: scalar integration

```
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?

```
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.

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

**Two costs:** (independent) work and coordination.

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

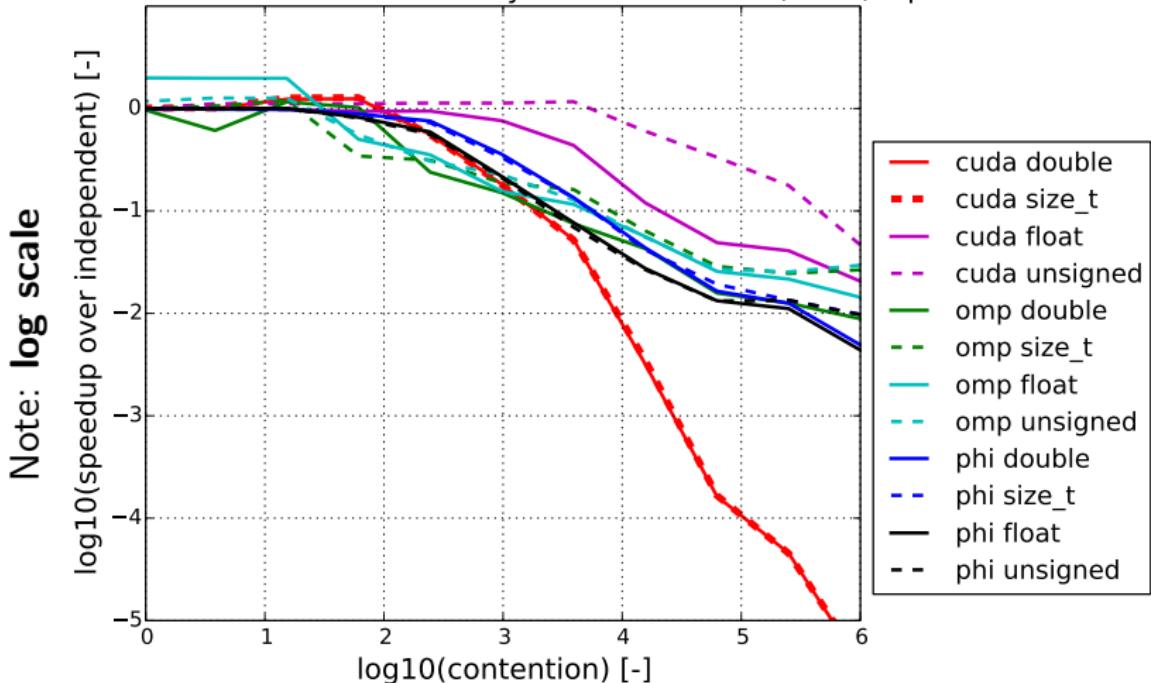
## Experimental setup

```
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`.
  - $\text{atomicStride} \rightarrow 1 \Rightarrow \text{Scalar integration}$
  - $\text{atomicStride} \rightarrow \text{large} \Rightarrow \text{Independent}$

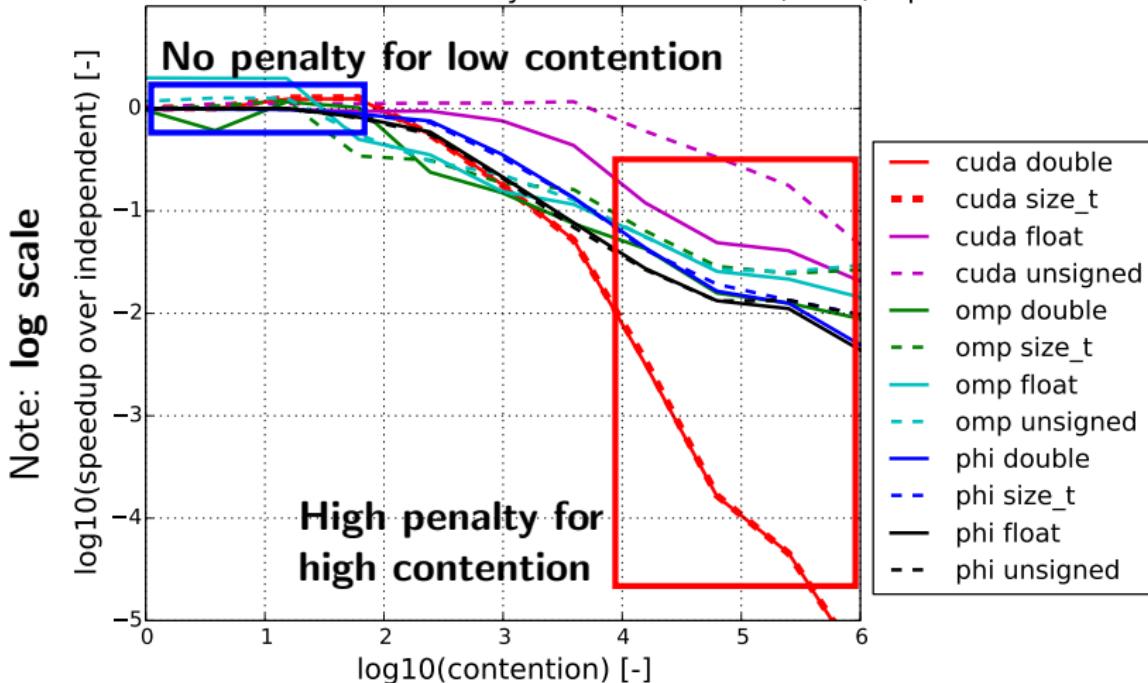
## Atomics performance: 1 million adds, **no** work per kernel

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



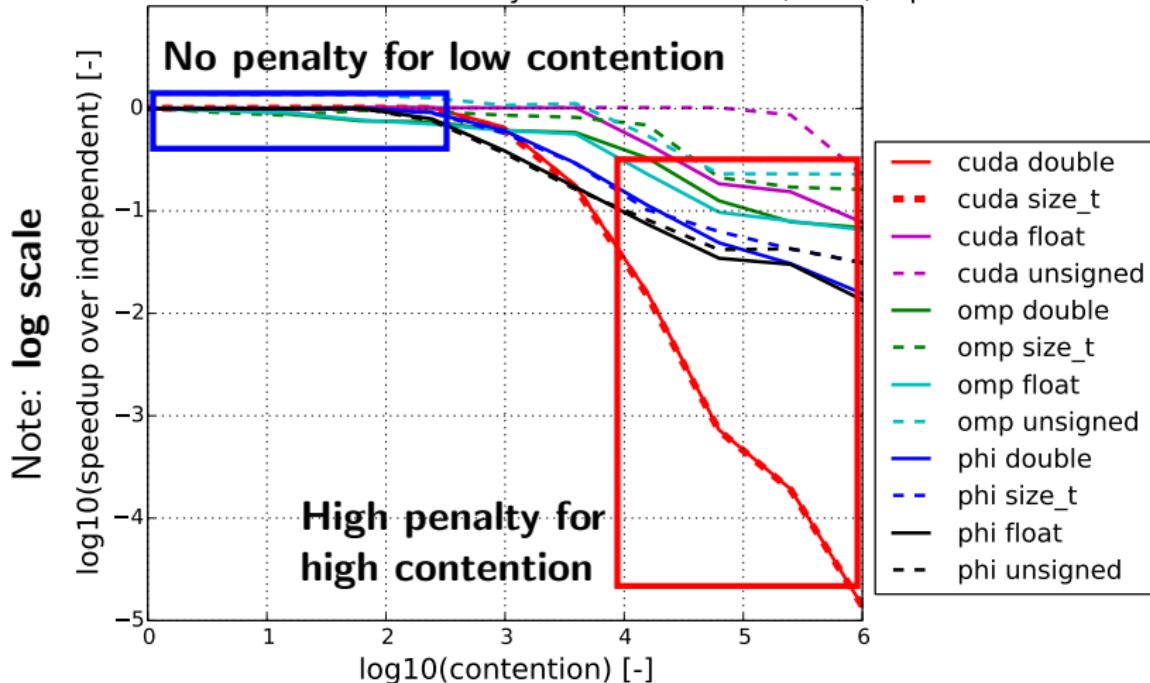
## Atomics performance: 1 million adds, **no** work per kernel

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



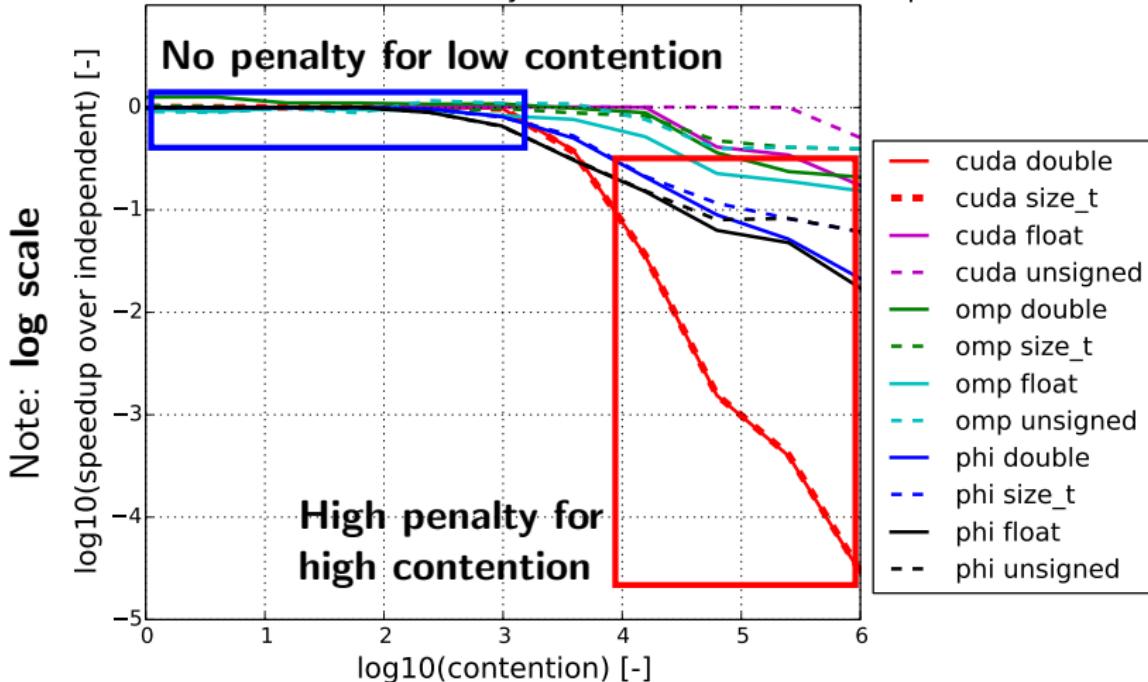
## Atomics performance: 1 million adds, **some** work per kernel

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



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

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



## 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.

```
// 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);
```

## Slight detour: 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:

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

## Slight detour: 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:

```
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**:

```
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**:

```
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**:

```
View< const double***, Layout, Space,
      MemoryTraits<RandomAccess> > name(...);
```

- ▶ Atomics are the only thread-scalable solution to thread safety.
  - ▶ Locks or data replication are **not portable or scalable**
- ▶ 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-scalably parallelized using atomics without much modification.

# Hierarchical parallelism

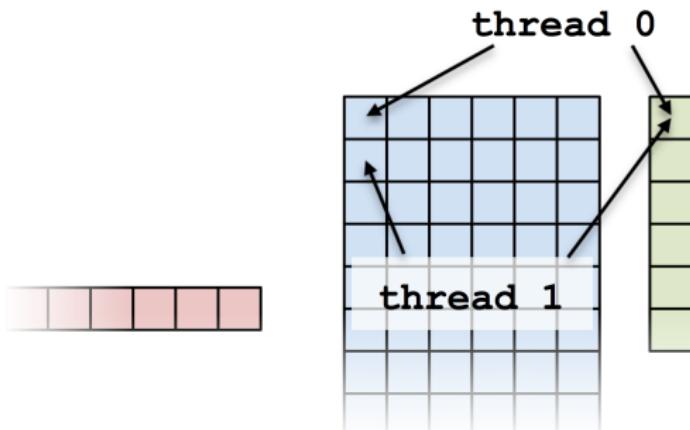
Finding and exploiting more parallelism in your computations.

## **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:

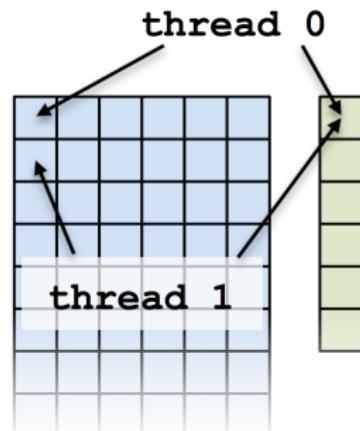
```
Kokkos::parallel_reduce(N,
    KOKKOS_LAMBDA (const int row, double & valueToUpdate) {
        double thisRowsSum = 0;
        for (int col = 0; col < M; ++col) {
            thisRowsSum += A(row,col) * x(col);
        }
        valueToUpdate += y(row) * thisRowsSum;
    }, result);
```



(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?

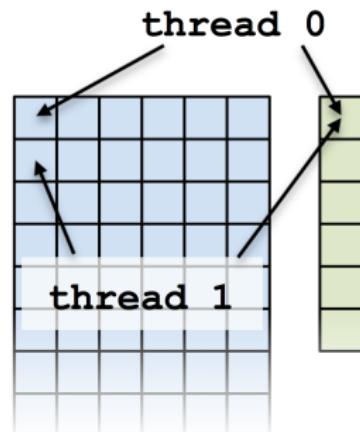


## (Flat parallel) Kernel:

```
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?**



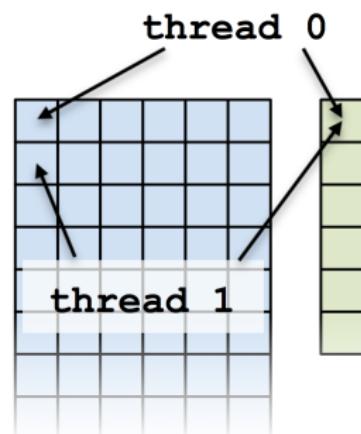
## (Flat parallel) Kernel:

```
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?

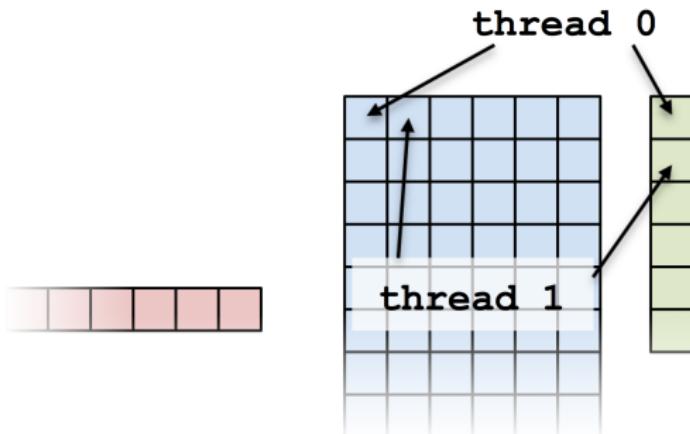
### Solutions?

- ▶ Atomics
- ▶ Thread teams



## Atomics kernel:

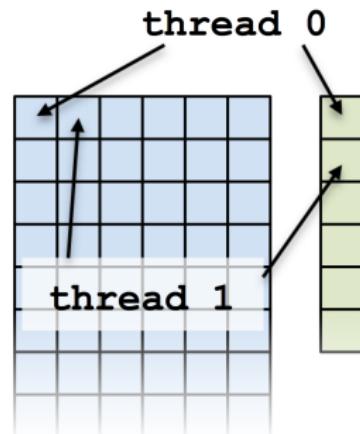
```
Kokkos::parallel_for(N,
    KOKKOS_LAMBDA (const size_t index) {
        const int row = extractRow(index);
        const int col = extractCol(index);
        atomic_add(&result, A(row,col) * x(col));
    });
}
```



## 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.

```
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.

```
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**.

## Important concept: Thread team

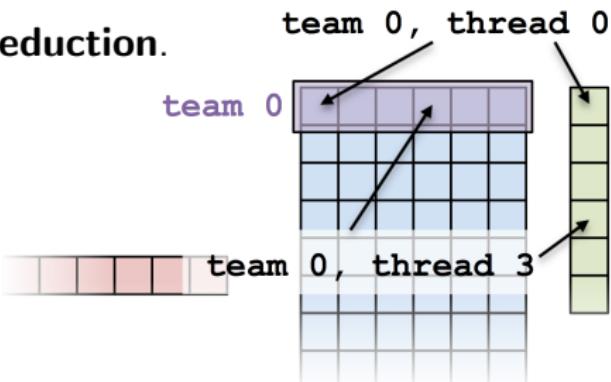
A collection of threads which are guaranteed to be executing **concurrently** and **can synchronize**.

## 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:

```
parallel_reduce(
    team_policy(N, Kokkos::AUTO),
    KOKKOS_LAMBDA (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);
```

## Important point

Using teams is changing the execution *policy*.

“Flat parallelism” uses RangePolicy:

We specify a *total amount of work*.

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

## Important point

Using teams is changing the execution *policy*.

“**Flat** parallelism” uses RangePolicy:

We specify a *total amount of work*.

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

“**Hierarchical** parallelism” uses TeamPolicy:

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

```
// total work = numberOfWorks * teamSize
parallel_for(
    TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize), functor);
```

## Important point

When using teams, functor operators receive a *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();
}
```

## Important point

When using teams, functor operators receive a *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:

```
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:

```
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  $\neq$  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...

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

We shouldn't be hard-coding the work mapping...

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

If this were a parallel execution,  
we'd use Kokkos::parallel\_reduce.

We shouldn't be hard-coding the work mapping...

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

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...

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

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

## TeamThreadRange:

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

- ▶ 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 by Kokkos.

## Anatomy of nested parallelism:

```
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.

In practice, you can **let Kokkos decide**:

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

In practice, you can **let Kokkos decide**:

```
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**:

```
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

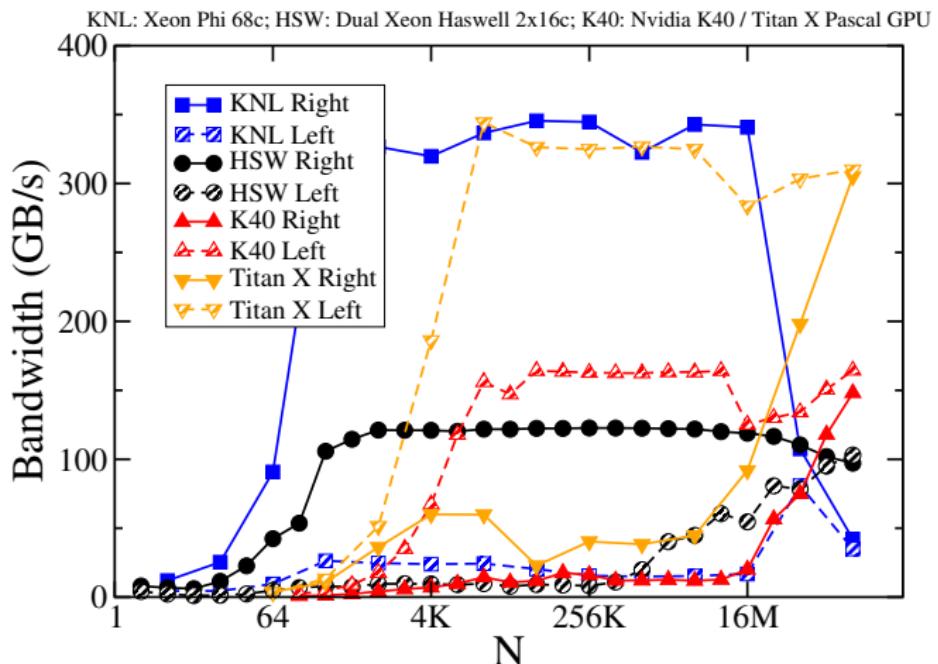
### Details:

- ▶ Location: ~/SC2016/Exercises/05/
- ▶ Replace RangePolicy<Space> with TeamPolicy<Space>
- ▶ Use AUTO for team\_size
- ▶ Make the inner loop a parallel\_reduce with TeamThreadRange policy
- ▶ Experiment with the combinations of Layout, Space, N to view performance
- ▶ Hint: what should the layout of A be?

### Things to try:

- ▶ Vary problem size and number of rows (-S ...; -N ...)
- ▶ Compare behaviour with Exercise 4 for very non-square matrices
- ▶ Compare behavior of CPU vs GPU

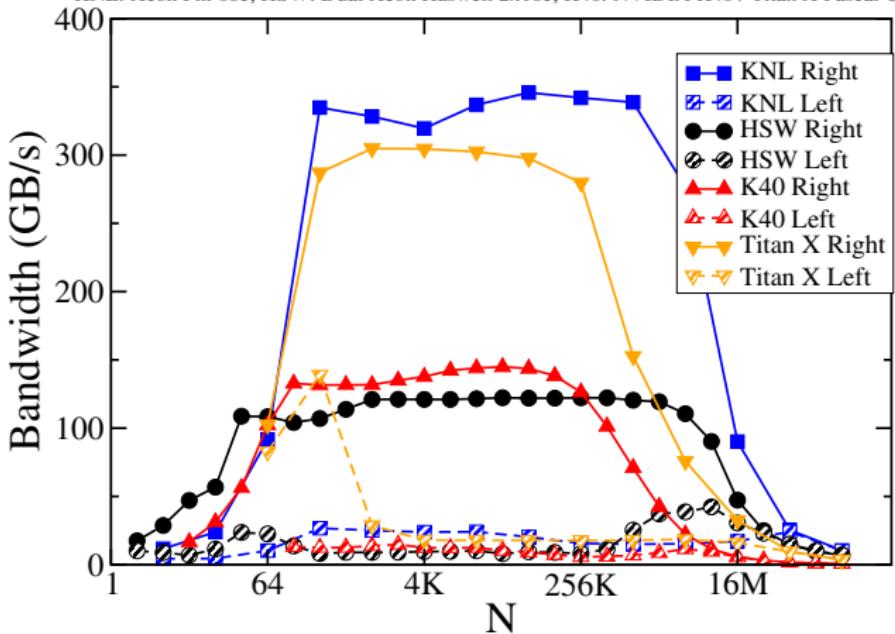
**Reminder:** Exercise 4 results (flat parallelism)  
 $\langle y | Ax \rangle$  Exercise 04 (Layout)



New: Exercise 5 results (hierarchical parallelism)

<ylAx> Exercise 05 (Layouts/Teams)

KNL: Xeon Phi 68c; HSW: Dual Xeon Haswell 2x16c; K40: NVIDIA K40 / Titan X Pascal GPU



## Exposing Vector Level Parallelism

- ▶ Optional **third level** in the hierarchy: ThreadVectorRange
  - ▶ Can be used for parallel\_for, parallel\_reduce, or parallel\_scan.
- ▶ Maps to vectorizable loop on CPUs or (sub-)warp level parallelism on GPUs.
- ▶ Enabled with a **runtime** vector length argument to TeamPolicy
- ▶ There is **no** explicit access to a vector lane ID.
- ▶ Depending on the backend the full global parallel region has active vector lanes.

## Anatomy of nested parallelism:

```
parallel_outer(
    TeamPolicy<>(numberOfTeams, teamSize, vectorLength),
    KOKKOS_LAMBDA (const member_type & teamMember[, ...]) {
        /* beginning of outer body */
        parallel_middle(
            TeamThreadRange(teamMember, thisTeamsRangeSize),
            [=] (const int indexWithinBatch[, ...]) {
                /* begin middle body */
                parallel_inner(
                    ThreadVectorRange(teamMember, thisVectorRangeSize),
                    [=] (const int indexVectorRange[, ...]) {
                        /* inner body */
                        }[, ...];
                    /* end middle body */
                }[, ...]);
            /* end of outer body */
        }[, ...]);
    }
```

A team execution is like a parallel region - all threads are active.

```
// Expected Answer?
parallel_reduce(TeamPolicy<>(N,team_size),
    KOKKOS_LAMBDA (const member_type& teamMember, int& lsum) {
        int s = 0;
        for(int i = 0; i<10; i++) s++;
        lsum += s;
    },sum);

// Expected Answer?
parallel_reduce(TeamPolicy<>(N,team_size),
    KOKKOS_LAMBDA (const member_type& teamMember, int& lsum) {
        int s = 0;
        parallel_reduce(TeamThreadRange(teamMember, team_size),
            [=] (const int k, int & inner_lsum) {
                int inner_s = 0;
                for(int i = 0; i<10; i++) inner_s++;
                inner_lsum += inner_s;
            },s);
        sum += s;
    },sum);
```

A team execution is like a parallel region - all threads are active.

```
// Expected Answer?
parallel_reduce(TeamPolicy<>(N,team_size),
    KOKKOS_LAMBDA (const member_type& teamMember, int& lsum) {
        int s = 0;
        for(int i = 0; i<10; i++) s++;
        lsum += s;      s = 10; sum = N * team_size * 10;
},sum);

// Expected Answer?
parallel_reduce(TeamPolicy<>(N,team_size),
    KOKKOS_LAMBDA (const member_type& teamMember, int& lsum) {
        int s = 0;
        parallel_reduce(TeamThreadRange(teamMember, team_size),
            [=] (const int k, int & inner_lsum) {
                int inner_s = 0;
                for(int i = 0; i<10; i++) inner_s++;
                inner_lsum += inner_s;
            },s);
        sum += s;
},sum);
```

A team execution is like a parallel region - all threads are active.

```
// Expected Answer?
parallel_reduce(TeamPolicy<>(N,team_size),
    KOKKOS_LAMBDA (const member_type& teamMember, int& lsum) {
        int s = 0;
        for(int i = 0; i<10; i++) s++;
        lsum += s;      s = 10; sum = N * team_size * 10;
},sum);

// Expected Answer?
parallel_reduce(TeamPolicy<>(N,team_size),
    KOKKOS_LAMBDA (const member_type& teamMember, int& lsum) {
        int s = 0;
        parallel_reduce(TeamThreadRange(teamMember, team_size),
            [=] (const int k, int & inner_lsum) {
                int inner_s = 0;
                for(int i = 0; i<10; i++) inner_s++;
                inner_lsum += inner_s;
            },s);
        sum += s;      inner_s = 10; s = team_size * 10;
},sum);          sum = N * team_size * team_size * 10;
```

The **single** pattern can be used to restrict execution

- ▶ Like parallel patterns it takes a policy, a lambda, and optionally a broadcast argument.
- ▶ Two policies: **PerTeam** and **PerThread**.
- ▶ Equivalent to OpenMP **single** directive with **nowait**

```
// Restrict to once per thread
single(PerThread(teamMember), [&] () {
    // code
});

// Restrict to once per team with broadcast
int offset = 0;
single(PerTeam(teamMember), [&] (int& offset_val) {
    offset_val = atomic_fetch_add(&counter(), team_N);
}, offset);
```

The previous example was extended with an outer loop over “Elements” to expose a third natural layer of parallelism.

### Details:

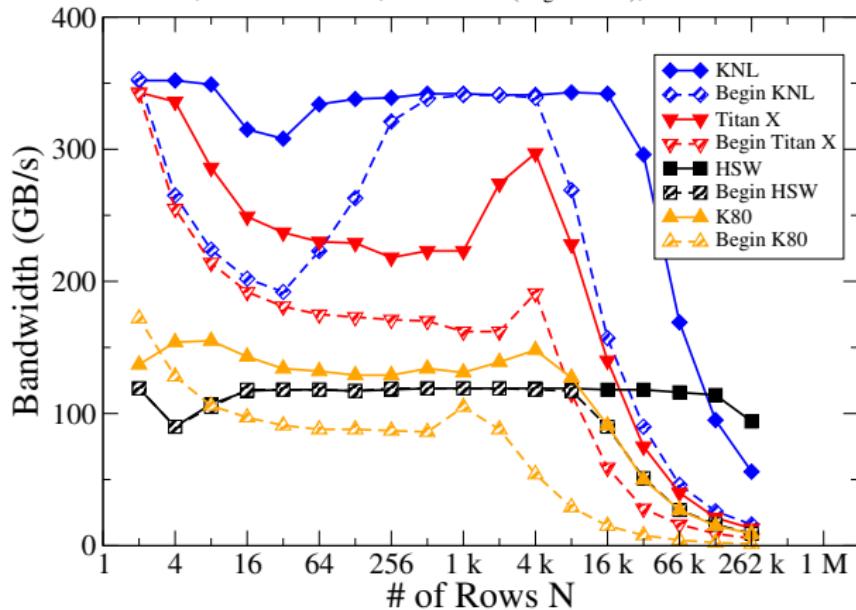
- ▶ Location: `~/SC2016/Exercises/06/`
- ▶ Use the `single` policy instead of checking team rank
- ▶ Parallelize all three loop levels.

### Things to try:

- ▶ Vary problem size and number of rows (`-S ...; -N ...`)
- ▶ Compare behaviour with Exercise 5 for very non-square matrices
- ▶ Compare behavior of CPU vs GPU

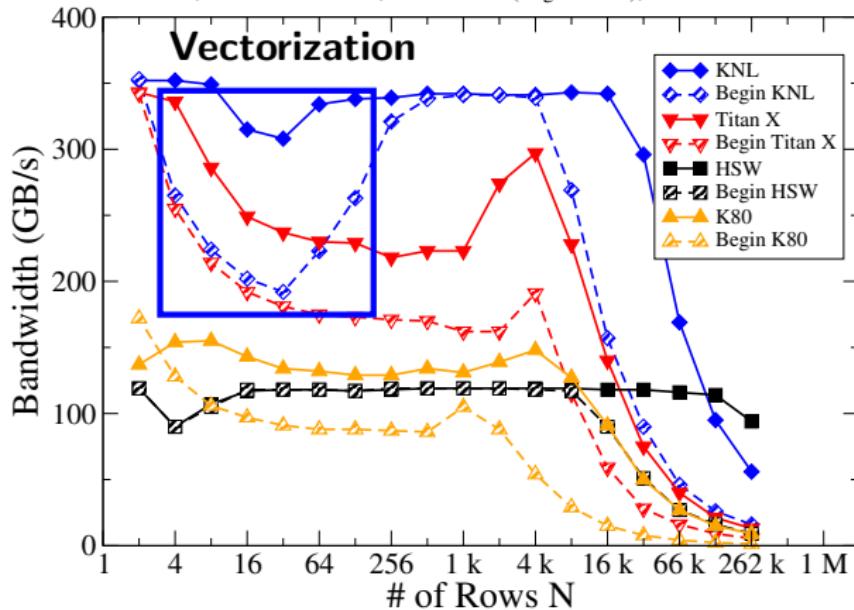
## Exercise 06 (Three Level Parallelism)

Xeon Phi KNL 68c; Dual Haswell 2x16c; NVIDIA K80 (single device); NVIDIA GeForce Titan X Pascal



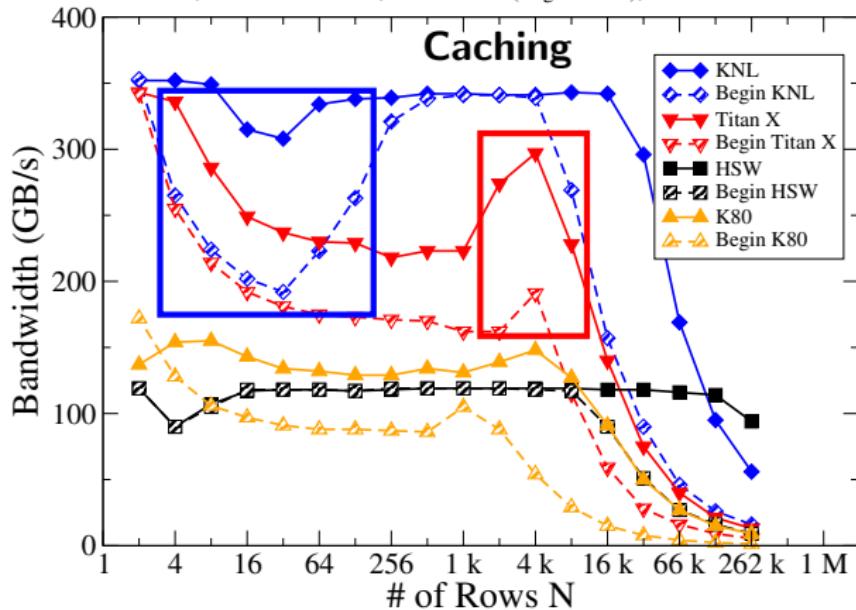
## Exercise 06 (Three Level Parallelism)

Xeon Phi KNL 68c; Dual Haswell 2x16c; NVIDIA K80 (single device); NVIDIA GeForce Titan X Pascal



## Exercise 06 (Three Level Parallelism)

Xeon Phi KNL 68c; Dual Haswell 2x16c; NVIDIA K80 (single device); NVIDIA GeForce Titan X Pascal



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

# Scratch memory

## Learning objectives:

- ▶ Understand concept of team and thread private scratch pads
- ▶ Understand how scratch memory can reduce global memory accesses
- ▶ Recognize when to use scratch memory
- ▶ Understand how to use scratch memory and when barriers are necessary

## Two Levels of Scratch Space

- ▶ Level 0 is limited in size but fast.
- ▶ Level 1 allows larger allocations but is equivalent to High Bandwidth Memory in latency and bandwidth.

## Team or Thread private memory

- ▶ Typically used for per work-item temporary storage.
- ▶ Advantage over pre allocated memory is aggregate size scales with number of threads, not number of work-items.

## Manually Managed Cache

- ▶ Explicitly cache frequently used data.
- ▶ Exposes hardware specific on-core scratch space (e.g. NVIDIA GPU Shared Memory).

## Two Levels of Scratch Space

- ▶ Level 0 is limited in size but fast.
- ▶ Level 1 allows larger allocations but is equivalent to High Bandwidth Memory in latency and bandwidth.

## Team or Thread private memory

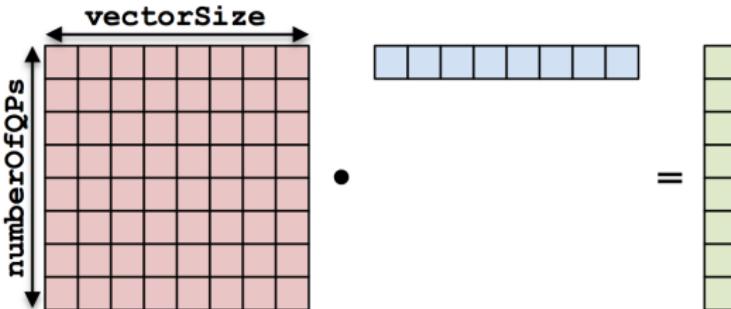
- ▶ Typically used for per work-item temporary storage.
- ▶ Advantage over pre allocated memory is aggregate size scales with number of threads, not number of work-items.

## Manually Managed Cache

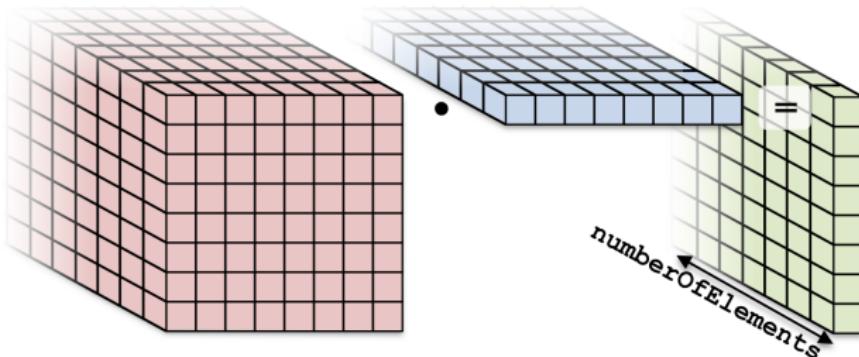
- ▶ Explicitly cache frequently used data.
- ▶ Exposes hardware specific on-core scratch space (e.g. NVIDIA GPU Shared Memory).

**Now: Discuss Manually Managed Cache Use Case.**

## One slice of contractDataFieldScalar:



```
for (qp = 0; qp < numberOfQPs; ++qp) {  
    total = 0;  
    for (i = 0; i < vectorSize; ++i) {  
        total += A(qp, i) * B(i);  
    }  
    result(qp) = total;  
}
```

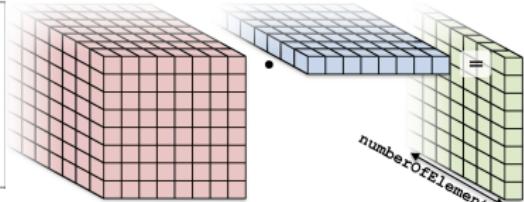
contractDataFieldScalar:

```

for (element = 0; element < number_of_elements; ++element) {
    for (qp = 0; qp < number_of_QPs; ++qp) {
        total = 0;
        for (i = 0; i < vector_size; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```

```
for (element = 0; element < numberElements; ++element) {  
    for (qp = 0; qp < numberQPs; ++qp) {  
        total = 0;  
        for (i = 0; i < vectorSize; ++i) {  
            total += A(element, qp, i) * B(element, i);  
        }  
        result(element, qp) = total;  
    }  
}
```



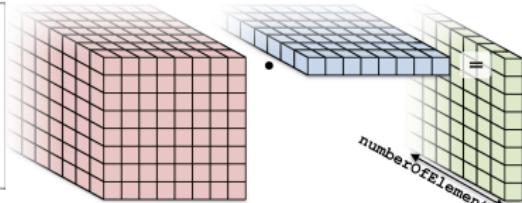
## Parallelization approaches:

- ▶ Each thread handles an element.  
Threads: numberElements

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



## Parallelization approaches:

- ▶ Each thread handles an element.

Threads:  $\text{numberElements}$

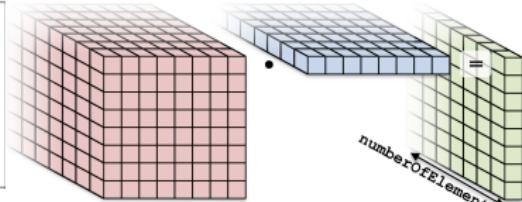
- ▶ Each thread handles a qp.

Threads:  $\text{numberElements} * \text{numberQPs}$

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



## Parallelization approaches:

- ▶ Each thread handles an element.

Threads:  $\text{numberElements}$

- ▶ Each thread handles a qp.

Threads:  $\text{numberElements} * \text{numberQPs}$

- ▶ Each thread handles an i.

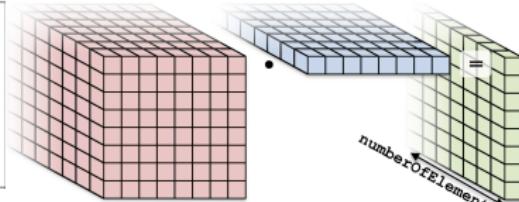
Threads:  $\text{numElements} * \text{numQPs} * \text{vectorSize}$

*Requires a parallel\_reduce.*

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



## Parallelization approaches:

- ▶ Each thread handles an element.

Threads: numberElements

- ▶ Each thread handles a qp.

Threads: numberElements \* numQPs

- ▶ Each thread handles an i.

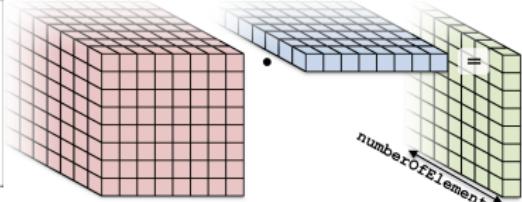
Threads: numElements \* numQPs \* vectorSize

*Requires a parallel\_reduce.*

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



## Flat kernel: Each thread handles a quadrature point

```

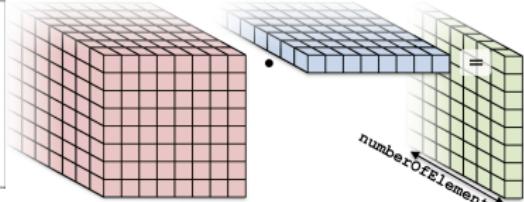
operator()(int index) {
    int element = extractElementFromIndex(index);
    int qp = extractQPFromIndex(index);
    double total = 0;
    for (int i = 0; i < _vectorSize; ++i) {
        total += _A(element, qp, i) * _B(element, i);
    }
    _result(element, qp) = total;
}

```

```

for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



## Teams kernel: Each team handles an element

```

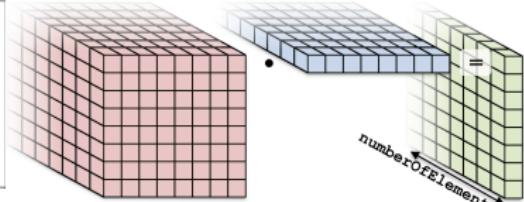
operator()(member_type teamMember) {
    int element = teamMember.league_rank();
    parallel_for(
        TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * _B(element, i);
            }
            _result(element, qp) = total;
        });
}

```

```

for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



## Teams kernel: Each team handles an element

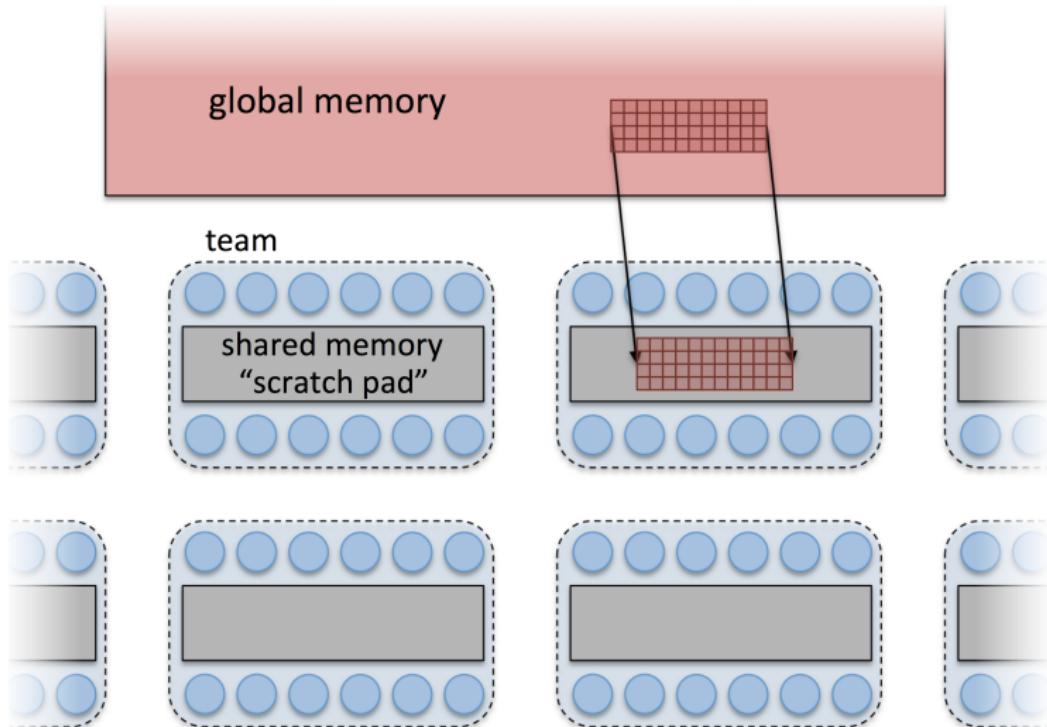
```

operator()(member_type teamMember) {
    int element = teamMember.league_rank();
    parallel_for(
        TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * _B(element, i);
            }
            _result(element, qp) = total;
        });
}

```

No real advantage (yet)

Each team has access to a “scratch pad”.



## Scratch memory (scratch pad) as manual cache:

- ▶ Accessing data in (level 0) scratch memory is (usually) **much faster** than global memory.
- ▶ **GPUs** have separate, dedicated, small, low-latency scratch memories (*NOT subject to coalescing requirements*).
- ▶ **CPUs** don't have special hardware, but programming with scratch memory results in cache-aware memory access patterns.
- ▶ Roughly, it's like a *user-managed* L1 cache.

## Scratch memory (scratch pad) as manual cache:

- ▶ Accessing data in (level 0) scratch memory is (usually) **much faster** than global memory.
- ▶ **GPUs** have separate, dedicated, small, low-latency scratch memories (*NOT subject to coalescing requirements*).
- ▶ **CPUs** don't have special hardware, but programming with scratch 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 scratch memory and read from there.

## Scratch memory for temporary per work-item storage:

- ▶ Scenario: Algorithm requires temporary workspace of size  $W$ .
- ▶ **Without scratch memory:** pre-allocate space for  $N$  work-items of size  $N \times W$ .
- ▶ **With scratch memory:** Kokkos pre-allocates space for each Team or Thread of size  $T \times W$ .
- ▶ PerThread and PerTeam scratch can be used concurrently.
- ▶ Level 0 and Level 1 scratch memory can be used concurrently.

## Scratch memory for temporary per work-item storage:

- ▶ Scenario: Algorithm requires temporary workspace of size  $W$ .
- ▶ **Without scratch memory:** pre-allocate space for  $N$  work-items of size  $N \times W$ .
- ▶ **With scratch memory:** Kokkos pre-allocates space for each Team or Thread of size  $T \times W$ .
- ▶ PerThread and PerTeam scratch can be used concurrently.
- ▶ Level 0 and Level 1 scratch memory can be used concurrently.

### Important concept

If an algorithm requires temporary workspace for each work-item  
use Kokkos's scratch memory.

To use scratch memory, you need to:

1. **Tell Kokkos how much** scratch memory you'll need.
2. **Make** scratch memory **views** inside your kernels.

To use scratch memory, you need to:

1. Tell Kokkos how much scratch memory you'll need.
2. Make scratch memory **views** inside your kernels.

```
TeamPolicy<execution_space> policy(N, TeamSize);

// scratch memory view type
typedef View< double*, execution_space::scratch_memory_space
            , MemoryUnmanaged > SharedViewType ;
// compute how much scratch memory (in bytes) is needed
size_t bytes = SharedViewType::shmem_size( _vectorSize );

// tell the policy how much scratch memory is needed
parallel_for(policy.set_scratch_size(0, PerTeam(bytes),
    KOKKOS_LAMBDA (const member_type& teamMember) {

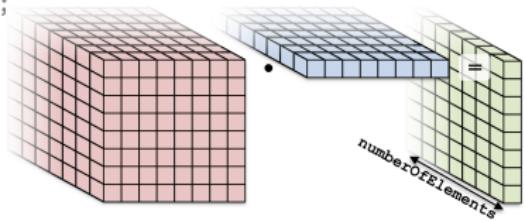
        // this creates a view from the pre-existing scratch memory
        SharedViewType scratch(teamMember.team_scratch(0), _vectorSize);
});
```

## Kernel outline for teams with scratch memory:

```
operator()(member_type teamMember) {
    SharedViewType scratch(teamMember.team_scratch(0), _vectorSize);

    // load slice of _B into scratch

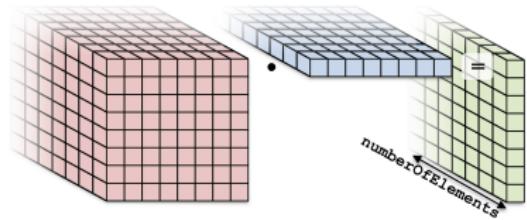
    parallel_for(
        TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * scratch(i);
            }
            _result(element, qp) = total;
        });
}
```



## How to populate the scratch memory?

- ▶ One thread loads it all?

```
if (teamMember.team_rank() == 0) {  
    for (int i = 0; i < _vectorSize; ++i) {  
        scratch(i) = B(element, i);  
    }  
}
```



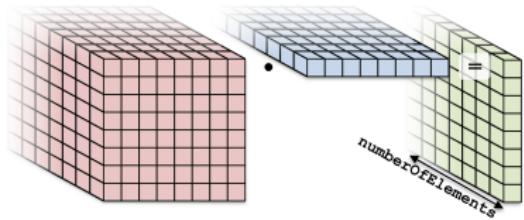
## How to populate the scratch memory?

- ▶ ~~One thread loads it all?~~      **Serial**

```
if (teamMember.team_rank() == 0) {  
    for (int i = 0; i < _vectorSize; ++i) {  
        scratch(i) = B(element, i);  
    }  
}
```

- ▶ Each thread loads one entry?

```
scratch(team_rank) = B(element, team_rank);
```



## How to populate the scratch memory?

- ▶ ~~One thread loads it all?~~      **Serial**

```
if (teamMember.team_rank() == 0) {
    for (int i = 0; i < _vectorSize; ++i) {
        scratch(i) = B(element, i);
    }
}
```

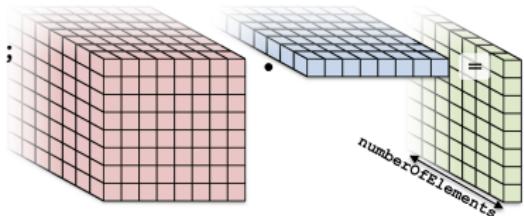
- ▶ ~~Each thread loads one entry?~~      **teamSize ≠ \_vectorSize**

```
scratch(team_rank) = B(element, team_rank);
```

- ▶ **TeamThreadRange**

```
parallel_for(
    TeamThreadRange(teamMember, _vectorSize),
    [=] (int i) {
        scratch(i) = B(element, i);
    });

```



## How to populate the scratch memory?

- ▶ ~~One thread loads it all?~~      **Serial**

```
if (teamMember.team_rank() == 0) {
    for (int i = 0; i < _vectorSize; ++i) {
        scratch(i) = B(element, i);
    }
}
```

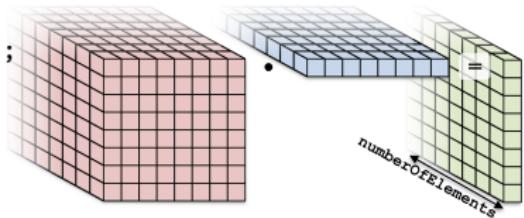
- ▶ ~~Each thread loads one entry?~~      **teamSize  $\neq$  \_vectorSize**

```
scratch(team_rank) = B(element, team_rank);
```

- ▶ **TeamThreadRange**

```
parallel_for(
    TeamThreadRange(teamMember, _vectorSize),
    [=] (int i) {
        scratch(i) = B(element, i);
    });

```



**(incomplete) Kernel for teams with scratch memory:**

```
operator()(member_type teamMember) {
    SharedViewType scratch(...);

    parallel_for(TeamThreadRange(teamMember, _vectorSize),
        [=] (int i) {
            scratch(i) = _B(element, i);
        });
}

parallel_for(TeamThreadRange(teamMember, _numberOfQPs),
    [=] (int qp) {
        double total = 0;
        for (int i = 0; i < _vectorSize; ++i) {
            total += _A(element, qp, i) * scratch(i);
        }
        _result(element, qp) = total;
    });
}
```

## (incomplete) Kernel for teams with scratch memory:

```
operator()(member_type teamMember) {
    SharedViewType scratch(...);

    parallel_for(TeamThreadRange(teamMember, _vectorSize),
        [=] (int i) {
            scratch(i) = _B(element, i);
        });
}

parallel_for(TeamThreadRange(teamMember, _numberOfQPs),
    [=] (int qp) {
        double total = 0;
        for (int i = 0; i < _vectorSize; ++i) {
            total += _A(element, qp, i) * scratch(i);
        }
        _result(element, qp) = total;
    });
}
```

**Problem:** threads may start to use **scratch** before all threads are done loading.

## Kernel for teams with scratch memory:

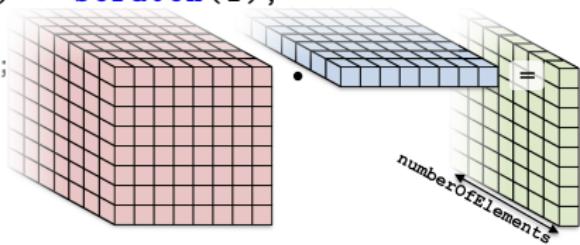
```

operator()(member_type teamMember) {
    SharedViewType scratch(...);

    parallel_for(TeamThreadRange(teamMember, _vectorSize),
        [=] (int i) {
            scratch(i) = _B(element, i);
        });
    teamMember.team_barrier();

    parallel_for(TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * scratch(i);
            }
            _result(element, qp) = total;
        });
}

```



Use Scratch Memory to explicitly cache the x-vector for each element.

### Details:

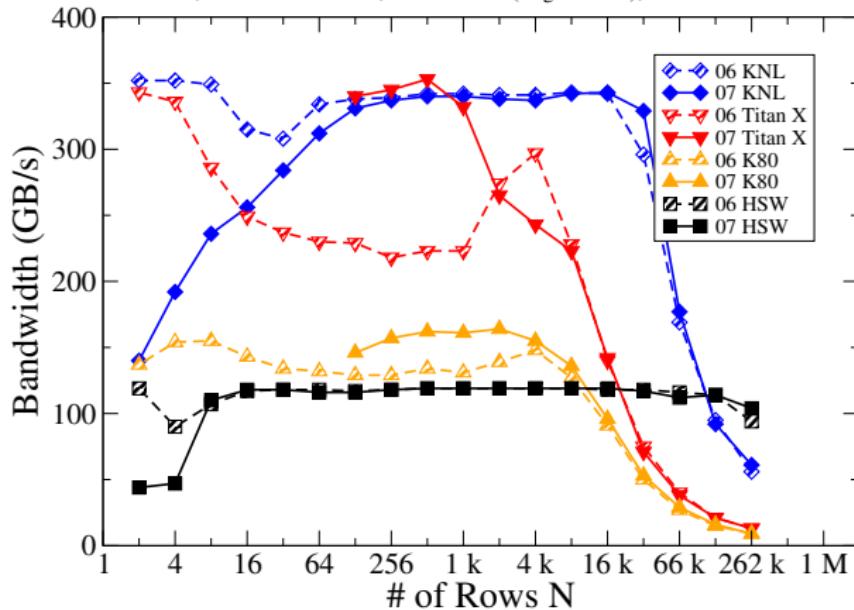
- ▶ Location: ~/SC2016/Exercises/07/
- ▶ Use the single policy instead of checking team rank
- ▶ Parallelize all three loop levels.

### Things to try:

- ▶ Vary problem size and number of rows (-S ...; -N ...)
- ▶ Compare behaviour with Exercise 6
- ▶ Compare behavior of CPU vs GPU

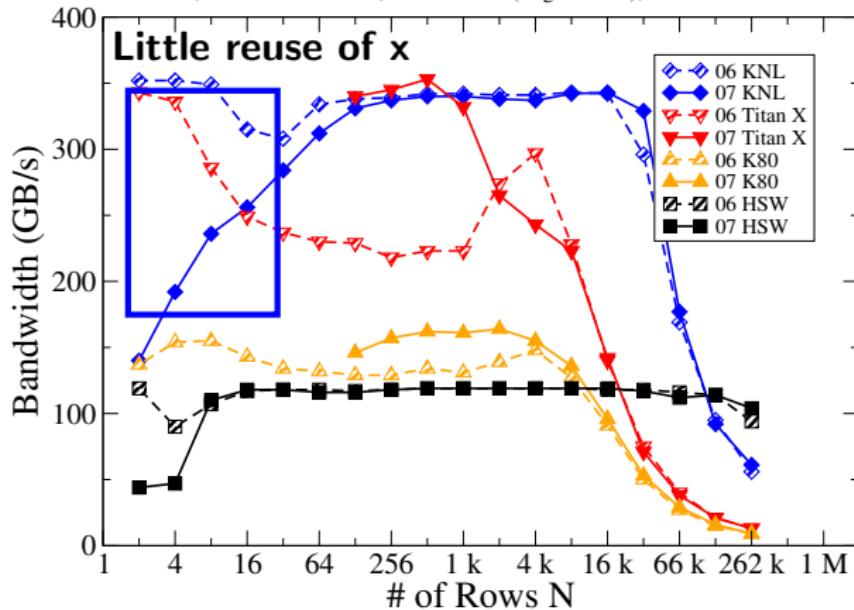
## Exercise 07 (Scratch Memory)

Xeon Phi KNL 68c; Dual Haswell 2x16c; NVIDIA K80 (single device); NVIDIA GeForce Titan X Pascal



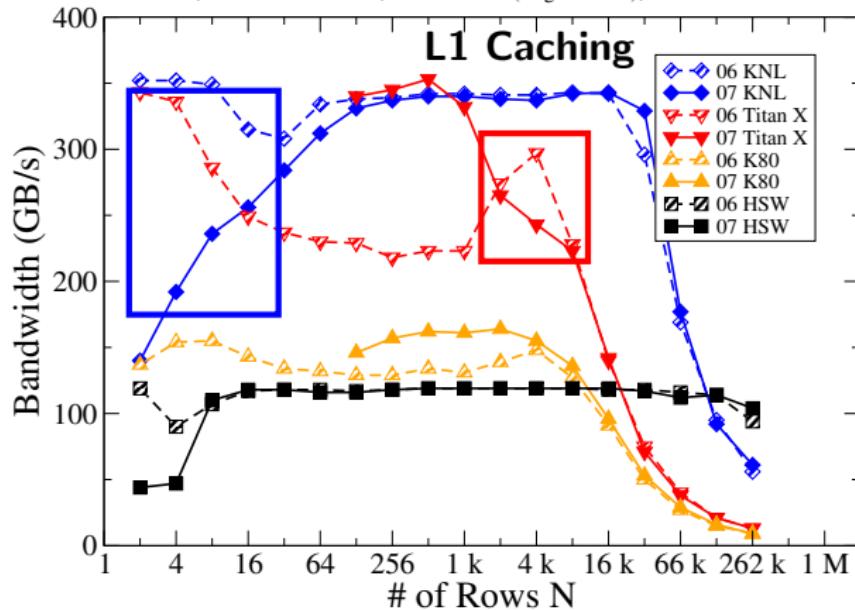
## Exercise 07 (Scratch Memory)

Xeon Phi KNL 68c; Dual Haswell 2x16c; NVIDIA K80 (single device); NVIDIA GeForce Titan X Pascal



## Exercise 07 (Scratch Memory)

Xeon Phi KNL 68c; Dual Haswell 2x16c; NVIDIA K80 (single device); NVIDIA GeForce Titan X Pascal



Allocating scratch in different levels:

```
int level = 1; // valid values 0,1  
policy.set_scratch_size(level,PerTeam(bytes));
```

Using PerThread, PerTeam or both:

```
policy.set_scratch_size(level,PerTeam(bytes));  
policy.set_scratch_size(level,PerThread(bytes));  
policy.set_scratch_size(level,PerTeam(bytes1),  
                        PerThread(bytes2));
```

Using both levels of scratch:

```
policy.set_scratch_size(0,PerTeam(bytes0))  
    .set_scratch_size(1,PerThread(bytes1));
```

Note: `set_scratch_size()` returns a new policy instance, it doesn't modify the existing one.

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

- ▶ 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.*