Kokkos Tutorial

Jeff Miles [1], Christian Trott [1]

[1]Sandia National Laboratories

Online April 21-24, 2020

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

Using your own ${HOME}

- Git
- GCC 4.8.4 (or newer) *OR* Intel 15 (or newer) *OR* Clang 3.5.2 (or newer)
- CUDA nvcc 9.0 (or newer) *AND* NVIDIA compute capability 3.0 (or newer)
- git clone `https://github.com/kokkos/kokkos`
  into `${HOME}/Kokkos/kokkos`
- git clone `https://github.com/kokkos/kokkos-tutorials`
  into `${HOME}/Kokkos/kokkos-tutorials`

  Slides are in
  `${HOME}/Kokkos/kokkos-tutorials/Intro-Full/Slides`

  Exercises are in
  `${HOME}/Kokkos/kokkos-tutorials/Intro-Full/Exercises`

  Exercises' makefiles look for `${HOME}/Kokkos/kokkos`

Prerequisites for Tutorial Exercises

Online Resources:

- https://github.com/kokkos: Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/blob/master/Intro-Full/Slides/KokkosTutorial_ORNL20.pdf: These slides.
- https://github.com/kokkos/kokkos/wiki: Wiki including API reference
- https://github.com/kokkos/kokkos-tutorials/issues/28: Instructions to get cloud instance with GPU
- https://kokkosteam.slack.com: Slack channel for Kokkos

Tutorial Objectives

Kokkos' basic capabilities:

- Simple 1D data parallel computational patterns
- Deciding where code is run and where data is placed
- Managing data access patterns for performance portability

Kokkos' advanced capabilities:

- Thread safety, thread scalability, and atomic operations
- Hierarchical patterns for maximizing parallelism

Kokkos' advanced capabilities not covered today:

- Multidimensional data parallelism
- Dynamic directed acyclic graph of tasks pattern
- Numerous *plugin* points for extensibility

Tutorial Takeaways

- Kokkos enables **Single Source Performance Portable Codes**
- **Simple things stay simple** - it is not much more complicated than OpenMP
- **Advanced performance optimizing capabilities** easier to use with Kokkos than e.g. CUDA
- Kokkos provides data abstractions critical for performance portability not available in OpenMP or OpenACC
  Controlling data access patterns is key for obtaining performance

Operating assumptions (0)

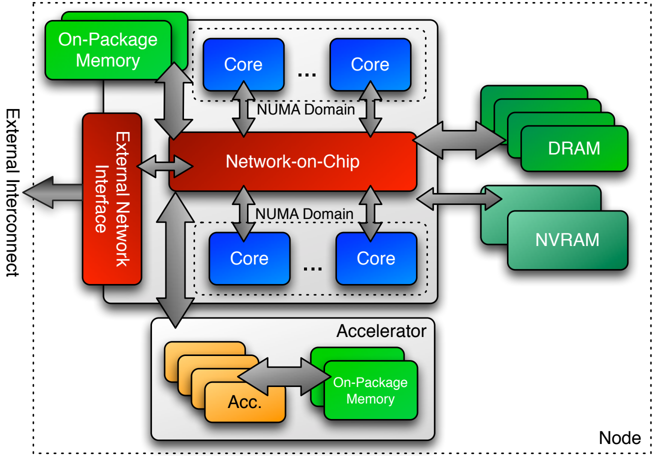
Assume you are here because:

- Want to use **all** HPC node architectures; including GPUs
- Are familiar with **C++**
- Want GPU programming to be **easier**
- Would like **portability**, as long as it doesn't hurt performance

Helpful for understanding nuances:

- Are familiar with **data parallelism**
- Are familiar with **OpenMP**
- Are familiar with **GPU architecture** and **CUDA**

Target machine:



Important Point: Performance Portability

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: Performance Portability

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: Performance Portability

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.

Concepts: Patterns, Policies, and Bodies

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

Concepts: Patterns, Policies, and Bodies

Pattern Policy

Body

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

Terminology:

- **Pattern**: structure of the computations
  for, reduction, scan, task-graph, ...
- **Execution Policy**: how computations are executed
  static scheduling, dynamic scheduling, thread teams, ...
- **Computational Body**: code which performs each unit of work; *e.g.*, the loop body

$\Rightarrow$ 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 PHI *and* NVIDIA GPU *and* AMD GPU *and* ...

“Parallel for” on a GPU via pragmas

Option 1: OpenMP 4.5

```
#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;
}
```

“Parallel for” on a GPU via pragmas

Option 1: OpenMP 4.5

```
#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;
}
```

Portable, but not performance portable

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

But what about performance?

Portable, but not performance portable

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, GPUs, 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.

Using Kokkos for data parallel patterns (0)

Data parallel patterns and work

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

Kokkos maps **work** to cores

Using Kokkos for data parallel patterns (0)

Data parallel patterns and work

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

Kokkos maps **work** to cores

- each iteration of a computational body is a **unit of work**.
- an **iteration index** identifies a particular unit of work.
- an **iteration range** identifies a total amount of work.

Using Kokkos for data parallel patterns (0)

Data parallel patterns and work

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

Kokkos maps **work** to cores

- each iteration of a computational body is a **unit of work**.
- an **iteration index** identifies a particular unit of work.
- an **iteration range** identifies a total amount of work.

Important concept: Work mapping

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

Using Kokkos for data parallel patterns (2)

How are computational bodies given to Kokkos?

Using Kokkos for data parallel patterns (2)

How are computational bodies given to Kokkos?

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

Using Kokkos for data parallel patterns (2)

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 ... */
  ...
};
```

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

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

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

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

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

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

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

```
struct Functor {
  void operator()(const int64_t index) const {...}
}
```

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

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

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

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

```
struct Functor {
  void operator()(const int64_t index) const {...}
}
```

Warning: concurrency and order

Concurrency and ordering of parallel iterations is *not* guaranteed by the Kokkos runtime.

Using Kokkos for data parallel patterns (4)

How is data passed to computational bodies?

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

```
struct AtomForceFunctor {
  ...
  void operator()(const int64_t atomIndex) const {
    atomForces[atomIndex] = calculateForce(...data...);
  }
  ...
}
```

Using Kokkos for data parallel patterns (4)

How is data passed to computational bodies?

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

```
struct AtomForceFunctor {
  ...
  void operator()(const int64_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**.

Using Kokkos for data parallel patterns (5)

Putting it all together: the complete functor:

```
struct AtomForceFunctor {
  ForceType _atomForces;
  AtomDataType _atomData;
  AtomForceFunctor(/* args */) {...}
  void operator()(const int64_t atomIndex) const {
    _atomForces[atomIndex] = calculateForce(_atomData);
  }
};
```

Putting it all together: the complete functor:

```
struct AtomForceFunctor {
  ForceType _atomForces;
  AtomDataType _atomData;
  AtomForceFunctor(/* args */) {...}
  void operator()(const int64_t atomIndex) const {
    _atomForces[atomIndex] = calculateForce(_atomData);
  }
};
```

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

Serial

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

Putting it all together: the complete functor:

```
struct AtomForceFunctor {
  ForceType _atomForces;
  AtomDataType _atomData;
  AtomForceFunctor(/* args */) {...}
  void operator()(const int64_t atomIndex) const {
    _atomForces[atomIndex] = calculateForce(_atomData);
  }
};
```

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

Serial

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

Functor

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

The complete picture (using functors):

1. Defining the functor (operator+data):

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

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

  void operator()(const int64_t atomIndex) const {
    _atomForces[atomIndex] = calculateForce(_atomData);
  }
}
```

2. **Executing** in parallel with Kokkos pattern:

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

Using Kokkos for data parallel patterns (7)

Functors are tedious ⇒ **C++11 Lambdas** are concise

```
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
    [=] (const int64_t atomIndex) {
    atomForces[atomIndex] = calculateForce(data);
  }
);
```

Using Kokkos for data parallel patterns (7)

Functors are tedious ⇒ **C++11 Lambdas** are concise

```
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
    [=] (const int64_t atomIndex) {
    atomForces[atomIndex] = calculateForce(data);
  }
);
```

A lambda is not *magic*, it is the compiler **auto-generating** a **functor** for you.

Using Kokkos for data parallel patterns (7)

Functors are tedious ⇒ **C++11 Lambdas** are concise

```
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
    [=] (const int64_t atomIndex) {
    atomForces[atomIndex] = calculateForce(data);
  }
);
```

A lambda is not *magic*, it is the compiler **auto-generating** a **functor** for you.

Warning: Lambda capture and C++ containers

For portability to GPU a lambda must capture by value `[=]`. Don't capture containers (*e.g.*, std::vector) by value because it will copy the container's entire contents.

How does this compare to OpenMP?

Serial

```
for (int64_t i = 0; i < N; ++i) {
  /* loop body */
}
```

OpenMP

```
#pragma omp parallel for
for (int64_t i = 0; i < N; ++i) {
  /* loop body */
}
```

Kokkos

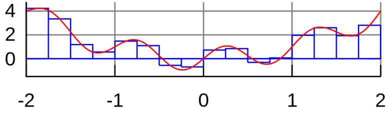
```
parallel_for(N, [=] (const int64_t i) {
  /* loop body */
});
```

Important concept

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

Riemann-sum-style numerical integration:

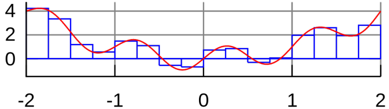
$$y = \int_{lower}^{upper} function(x)\, dx$$



Wikipedia

Riemann-sum-style numerical integration:

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

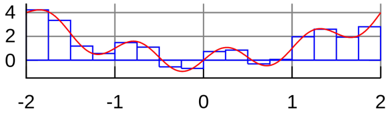


Wikipedia

```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  const double x =
    lower + (i/numberOfIntervals) * (upper - lower);
  const double thisIntervalsContribution = function(x);
  totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

Riemann-sum-style numerical integration:

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



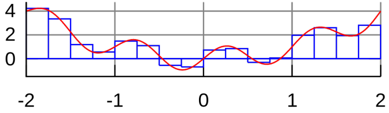
Wikipedia

```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  const double x =
    lower + (i/numberOfIntervals) * (upper - lower);
  const double thisIntervalsContribution = function(x);
  totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

How do we **parallelize** it? *Correctly?*

Riemann-sum-style numerical integration:

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



Wikipedia

Pattern?

Policy?

Body?

```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  const double x =
    lower + (i/numberOfIntervals) * (upper - lower);
  const double thisIntervalsContribution = function(x);
  totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

How do we **parallelize** it? *Correctly?*

Scalar integration (1)

An (incorrect) attempt:

```
double totalIntegral = 0;
Kokkos::parallel_for(numberOfIntervals,
  [=] (const int64_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 int64_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 int64_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

Scalar integration (3)

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

Scalar integration (3)

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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 (int64_t i = 0; i < N; ++i) {
  finalReducedValue += ...
}
```

Scalar integration (3)

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

```
double finalReducedValue = 0;
#pragma omp parallel for reduction(+:finalReducedValue)
for (int64_t i = 0; i < N; ++i) {
  finalReducedValue += ...
}
```

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

```
double finalReducedValue = 0;
parallel_reduce(N, functor, finalReducedValue);
```

Example: Scalar integration

OpenMP

```
double totalIntegral = 0;
#pragma omp parallel for reduction(+:totalIntegral)
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  totalIntegral += function(...);
}
```

Kokkos

```
double totalIntegral = 0;
parallel_reduce(numberOfIntervals,
  [=] (const int64_t i, double & valueToUpdate) {
    valueToUpdate += function(...);
  },
  totalIntegral);
```

- The operator takes **two arguments**: a work index and a value to update.
- The second argument is a **thread-private value** that is managed by Kokkos; it is not the final reduced value.

Scalar integration (5)

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

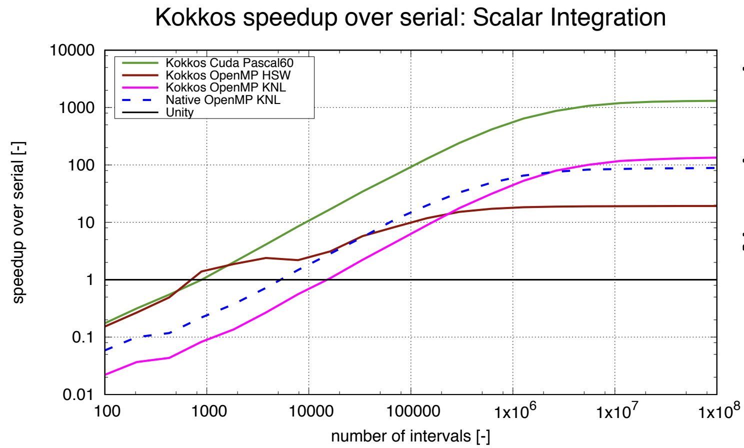
- $\alpha =$ dispatch overhead
- $\beta =$ time for a unit of work
- $N =$ number of units of work
- $P =$ available concurrency

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

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

Scalar integration (6)

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



Note: log scale

Always name your kernels!

Giving unique names to each kernel is immensely helpful for debugging and profiling. You will regret it if you don't!

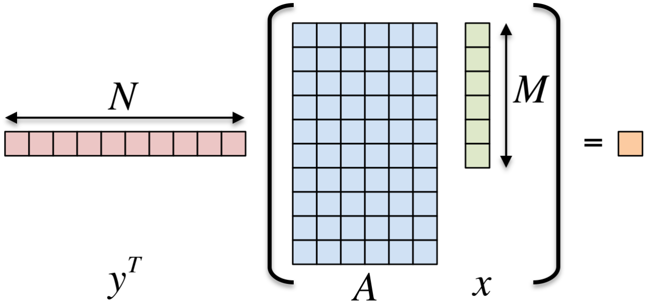
- Non-nested parallel patterns can take an optional string argument.
- The label doesn't need to be unique, but it is helpful.
- Anything convertible to "const std::string"
- Used by profiling and debugging tools (see Profiling Tutorial)

Example:

```
double totalIntegral = 0;
parallel_reduce("Reduction",numberOfIntervals,
  [=] (const int64_t i, double & valueToUpdate) {
    valueToUpdate += function(...);
  },
  totalIntegral);
```

Recurring Exercise: Inner Product

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



Details:

- y is Nx1, A is NxM, x is Mx1
- 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:

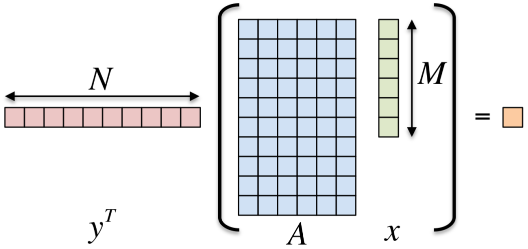
```
#include <Kokkos_Core.hpp>
int main(int argc, char** argv) {
  /* ... do any necessary setup (e.g., initialize MPI) ... */
  Kokkos::initialize(argc, argv);
  {
  /* ... do computations ... */
  }
  Kokkos::finalize();
  return 0;
}
```

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

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



Details:

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

Compiling for CPU

```
# gcc using OpenMP (default) and Serial back-ends,
# (optional) change non-default arch with KOKKOS_ARCH
  make -j KOKKOS_DEVICES=OpenMP,Serial KOKKOS_ARCH=...
```

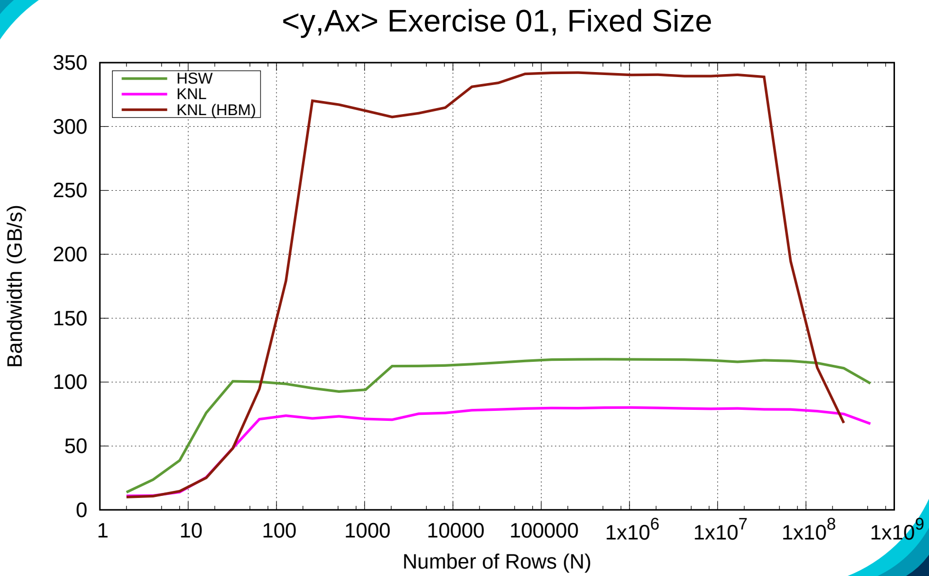
Running on CPU with OpenMP back-end

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

Things to try:

- Vary problem size with cline arg -S s
- Vary number of rows with cline arg -N n
- Num rows $= 2^n$, num cols $= 2^m$, total size $= 2^s == 2^{n+m}$

Exercise #1 results



Basic capabilities we haven't covered

- Customizing `parallel_reduce` data type and reduction operator
  - *e.g.*, minimum, maximum, ...
- `parallel_scan` pattern for exclusive and inclusive prefix sum
- Using *tag dispatch* interface to allow non-trivial functors to have multiple "`operator()`" functions.
  - very useful in large, complex applications

Section Summary

- **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.

View motivation

Example: running daxpy on the GPU:

Lambda

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

Functor

```
struct Functor {
  double *_x, *_y, a;
  void operator()(const int64_t i) {
    _y[i] = _a * _x[i] + _y[i];
  }
};
```

View motivation

Example: running daxpy on the GPU:

Lambda

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

Functor

```
struct Functor {
  double *_x, *_y, a;
  void operator()(const int64_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("DAXPY",N, [=] (const int64_t i) {
    y[i] = a * x[i] + y[i];
  });
```

Functor

```
struct Functor {
  double *_x, *_y, a;
  void operator()(const int64_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).

$\Rightarrow$ **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("DAXPY",N, [=] (const int64_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("DAXPY",N, [=] (const int64_t i) {
    // Views x and y are captured by value (copy)
    y(i) = a * x(i) + y(i);
  });
```

Important point

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

View overview:

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

Views (1)

View overview:

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

Example:

```
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
//Access
data(i,j,k) = 5.3;
```

Note: runtime-sized dimensions must come first.

Views (2)

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`

Views (2)

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

What gets printed?

View life cycle:

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

Example:

```
View<double*[5]> a("a", N0), b("b", N0);
a = b;
View<double**> c(b);
a(0,2) = 1;
b(0,2) = 2;
c(0,2) = 3;
print a(0,2)
```

What gets printed?

```
3.0
```

Views (3)

View Properties:

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

Example:

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

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

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

```
make -j KOKKOS_DEVICES=OpenMP   # CPU-only using OpenMP
make -j KOKKOS_DEVICES=Cuda     # GPU - note UVM in Makefile
# 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

Advanced features we haven't covered

- **Memory space** in which view's data resides; *covered next*.
- **deep_copy** view's data; *covered later*.
  Note: Kokkos *never* hides a deep_copy of data.
- **Layout** of multidimensional array; *covered later*.
- **Memory traits**; *covered later*.
- **Subview**: Generating a view that is a "slice" of other multidimensional array view; *covered later*.

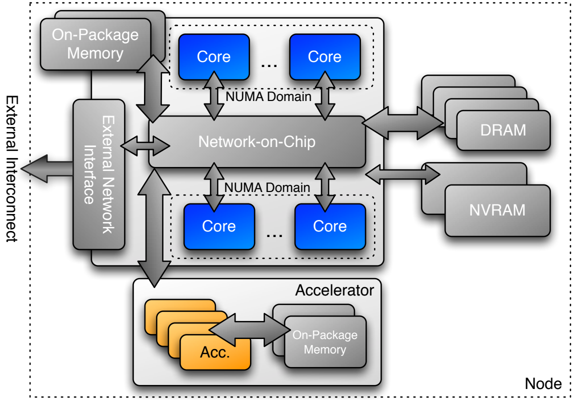
Execution and Memory Spaces

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.

Execution Space

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



Execution spaces: `Serial`, `Threads`, `OpenMP`, `Cuda`, `HIP`, ...

Execution spaces (2)

Host

Parallel

```
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                     [=] (const int64_t somethingIndex) {
                       const double y = ...;
                       // do something interesting
                     }
                     );
```

Execution spaces (2)

Host

```
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                     [=] (const int64_t somethingIndex) {
                       const double y = ...;
                       // do something interesting
                     }
                     );
```

Parallel

- Where will Host code be run? CPU? GPU?

  ⇒ Always in the **host process**

Execution spaces (2)

Host

Parallel

```
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                     [=] (const int64_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**

Execution spaces (2)

Host

Parallel

```
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                     [=] (const int64_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("Label",
  RangePolicy< ExecutionSpace >(0,numberOfIntervals),
  [=] (const int64_t i) {
    /* ... body ... */
  });
```

Default

```
parallel_for("Label",
  numberOfIntervals, // == RangePolicy<>(0,numberOfIntervals)
  [=] (const int64_t i) {
    /* ... body ... */
  });
```

Changing the parallel execution space:

Custom

```
parallel_for("Label",
  RangePolicy< ExecutionSpace >(0,numberOfIntervals),
  [=] (const int64_t i) {
    /* ... body ... */
  });
```

Default

```
parallel_for("Label",
  numberOfIntervals, // == RangePolicy<>(0,numberOfIntervals)
  [=] (const int64_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 int64_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const int64_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 int64_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const int64_t index) const {
    helperFunction(index);
  }
}
// Where kokkos defines:
#define KOKKOS_INLINE_FUNCTION inline                     /* #if CPU-only */
#define KOKKOS_INLINE_FUNCTION inline __device__ __host__ /* #if CPU+Cuda */
```

Lambda annotation with `KOKKOS_LAMBDA` macro (requires CUDA 8.0)

```
Kokkos::parallel_for("Label",numberOfIterations,
  KOKKOS_LAMBDA (const int64_t index) {...});

// Where Kokkos defines:
#define KOKKOS_LAMBDA [=]            /* #if CPU-only */
#define KOKKOS_LAMBDA [=] __device__ /* #if CPU+Cuda */
```

Memory space motivating example: summing an array

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

double sum = 0;
Kokkos::parallel_reduce("Label",
  RangePolicy<SomeExampleExecutionSpace>(0, size),
  KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
    valueToUpdate += data(index);
  },
  sum);
```

Memory space motivating example: summing an array

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

double sum = 0;
Kokkos::parallel_reduce("Label",
  RangePolicy<SomeExampleExecutionSpace>(0, size),
  KOKKOS_LAMBDA (const int64_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 (int64_t i = 0; i < size; ++i) {
  data(i) = ...read from file...
}

double sum = 0;
Kokkos::parallel_reduce("Label",
  RangePolicy<SomeExampleExecutionSpace>(0, size),
  KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
    valueToUpdate += data(index);
  },
  sum);
```

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

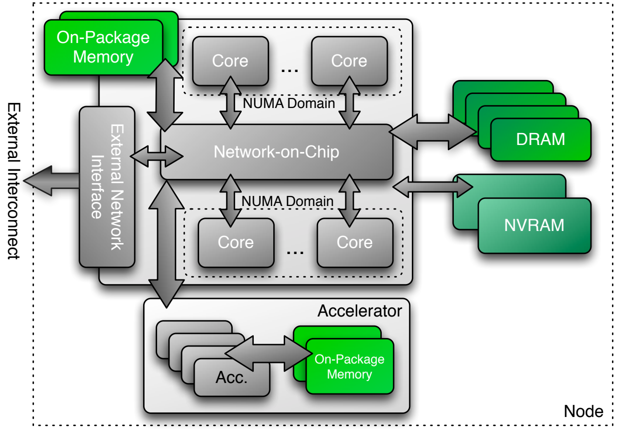
Memory space motivating example: summing an array

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

double sum = 0;
Kokkos::parallel_reduce("Label",
  RangePolicy<SomeExampleExecutionSpace>(0, size),
  KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
    valueToUpdate += data(index);
  },
  sum);
```

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

$\Rightarrow$ **Memory Spaces**

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



Memory spaces (1)

Important concept: Memory spaces

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

Memory spaces (1)

Important concept: Memory spaces

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

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

Important concept: Memory spaces

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

- `View<double***,`*Memory***Space**`> 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***,`*Memory***Space**`> 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

Memory spaces (1)

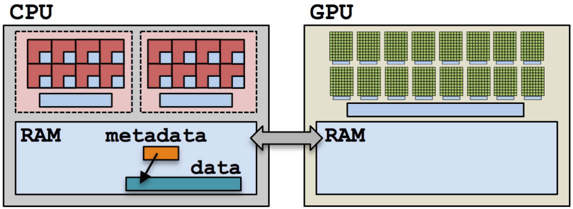
Important concept: Memory spaces

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

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

Example: HostSpace

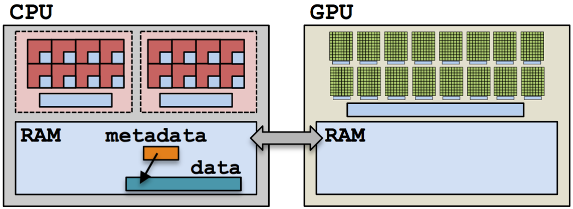
```
View<double**, HostSpace> hostView(...constructor arguments...);
```



Memory spaces (2)

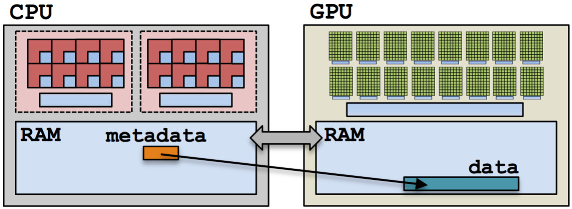
Example: HostSpace

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



Example: CudaSpace

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



Execution and Memory spaces (0)

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.

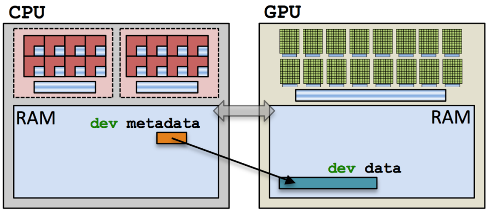
```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev(...);
parallel_for("Label",N,
  KL (int i) {
    dev(i) = ...;
  });
```

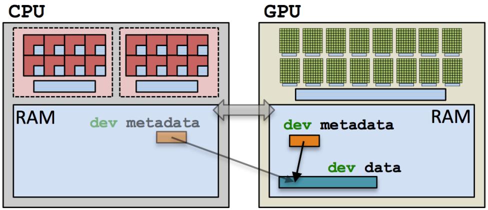
Note: **no deep copies** of array data are performed; *views are like pointers*.

Execution and Memory spaces (1)

Example: one view

```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
parallel_for("Label",N,
  KL (int i) {
    dev(i) = ...;
  });
```

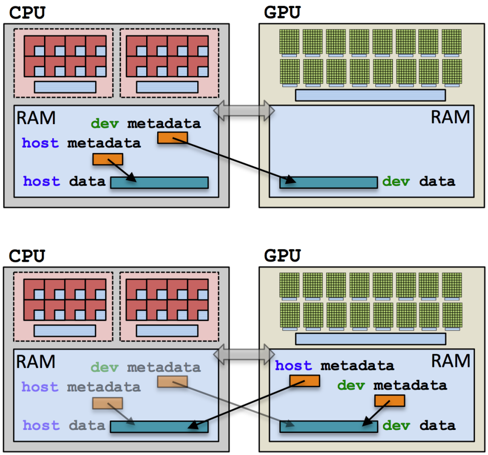




Execution and Memory spaces (2)

Example: two views

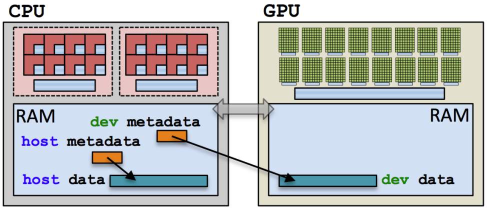
```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for("Label",N,
  KL (int i) {
    dev(i)  = ...;
    host(i) = ...;
  });
```

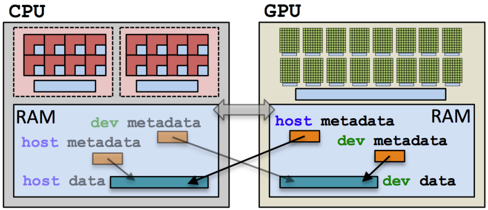


Execution and Memory spaces (2)

Example: two views

```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for("Label",N,
  KL (int i) {
    dev(i)  = ...;
    host(i) = ...;
  });
```





Example (redux): summing an array with the GPU

(failed) Attempt 1: `View` lives in `CudaSpace`

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 1: `View` lives in `CudaSpace`

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2: `View` lives in `HostSpace`

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2: `View` lives in `HostSpace`

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2: `View` lives in `HostSpace`

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

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

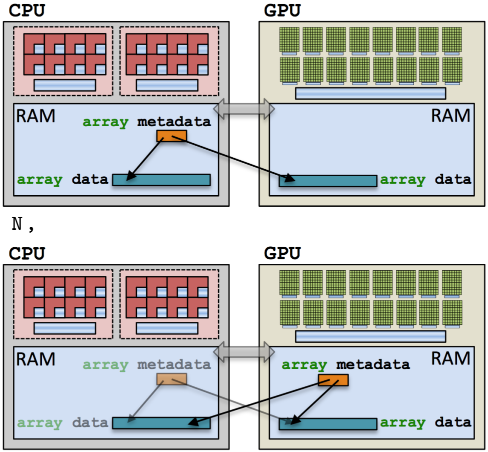
What's the solution?

- CudaUVMSpace
- CudaHostPinnedSpace (skipping)
- Mirroring

Execution and Memory spaces (5)

CudaUVMSpace

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



Cuda runtime automatically handles data movement, at a **performance hit**.

Important concept: Mirrors

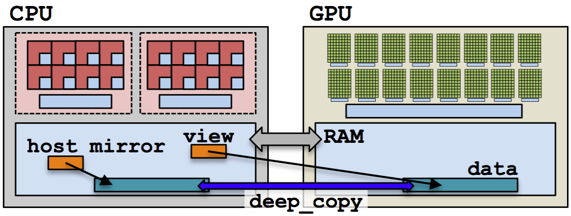
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

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);
```



Mirroring pattern

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);
```

Mirroring pattern

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.).

Mirroring pattern

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("Label",
     RangePolicy< Space>(0, size),
     KOKKOS_LAMBDA (...) { use and change view });
   ```

Mirroring pattern

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("Label",
     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);
   ```

Mirrors of Views in HostSpace

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

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

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

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

Details:

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

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

Things to try:

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

View and Spaces Section Summary

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

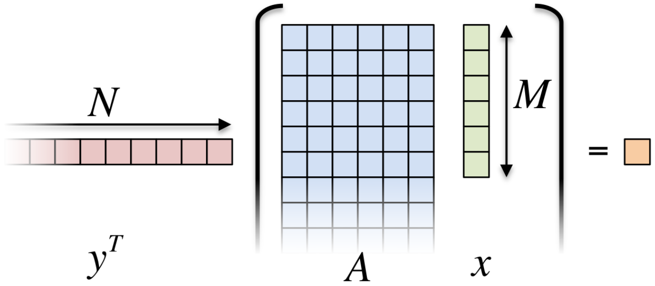
Managing memory access patterns for performance portability

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.

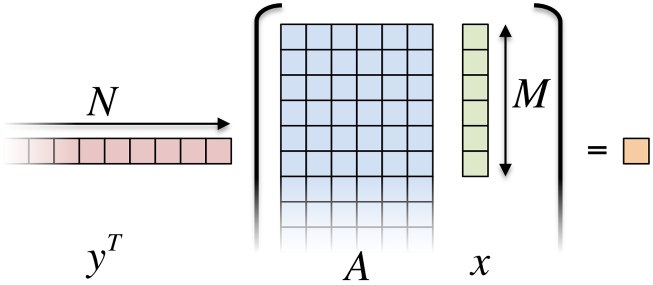
Example: inner product (0)

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



Example: inner product (0)

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



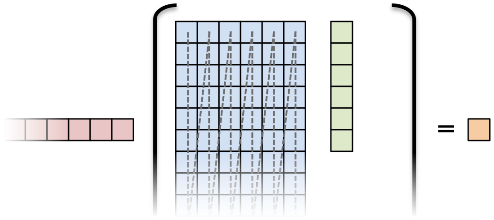
Driving question: How should A be laid out in memory?

Example: inner product (1)

Layout is the mapping of multi-index to memory:

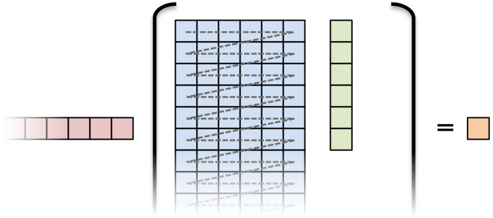
LayoutLeft

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: $\approx$ 50 lines
- Advanced layouts: `LayoutStride`, `LayoutTiled`, ...

Exercise #4: Inner Product, Flat Parallelism

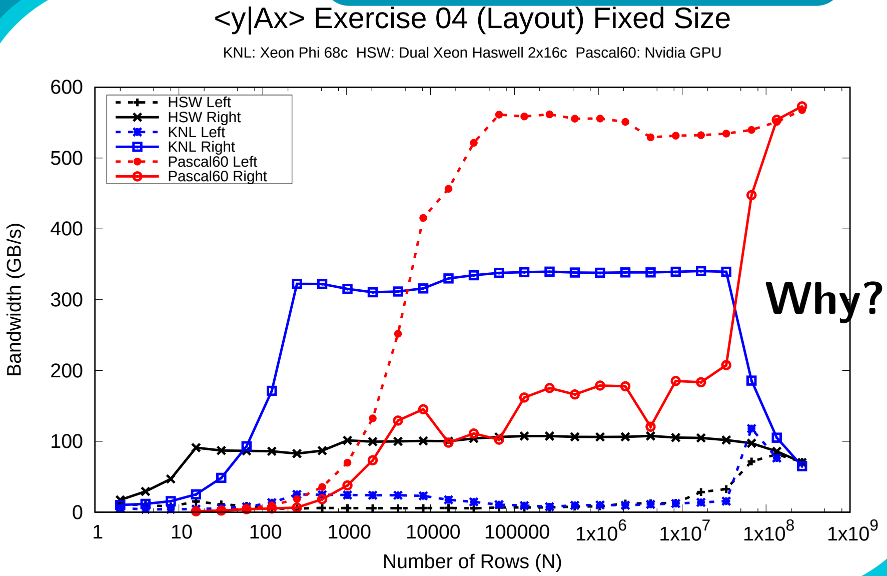
Details:

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

Things to try:

- Vary problem size and number of rows (-S ...; -N ...)
- Change number of repeats (-nrepeat ...)
- Compare behavior of CPU vs GPU
- Compare using UVM vs not using UVM on GPUs
- Check what happens if `MemSpace` and `ExecSpace` do not match.

Exercise #4: Inner Product, Flat Parallelism



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 benefit (NVIDIA Volta) or must synchronize (AMD) in groups.
  i.e., threads in groups can/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 benefit (NVIDIA Volta) or must synchronize (AMD) in groups.
  i.e., threads in groups can/must execute instructions together.

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

Thread independence:

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

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

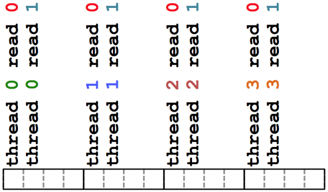
- **CPU** threads are independent.
  i.e., threads may execute at any rate.
- **GPU** threads benefit (NVIDIA Volta) or must synchronize (AMD) in groups.
  i.e., threads in groups can/must execute instructions together.

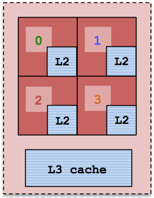
In particular, all threads in a group (*warp* or *wavefront*) must finished their loads before *any* thread can move on.

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

Caching and coalescing (1)

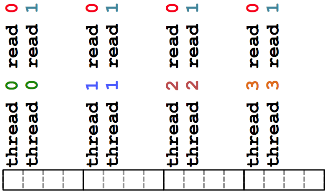
CPUs: few (independent) cores with separate caches:

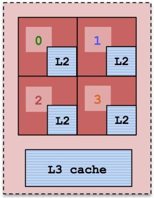




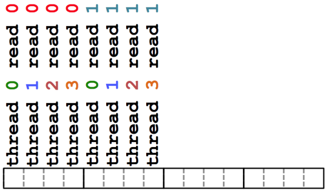
Caching and coalescing (1)

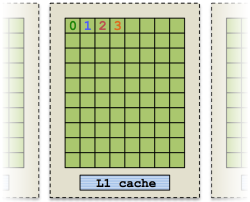
CPUs: few (independent) cores with separate caches:





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





Caching and coalescing (2)

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`.

Caching and coalescing (2)

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 on GPUs and non-cached loads on CPUs *greatly* reduces performance (can be ¿10X)

Mapping indices to cores (0)

Consider the array summation example:

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

double sum = 0;
Kokkos::parallel_reduce("Label",
  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`)?

Mapping indices to cores (0)

Consider the array summation example:

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

double sum = 0;
Kokkos::parallel_reduce("Label",
  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:	Strided:
`0, 1, 2, ..., N/P`	`0, N/P, 2*N/P, ...`

Mapping indices to cores (0)

Consider the array summation example:

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

double sum = 0;
Kokkos::parallel_reduce("Label",
  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:	Strided:
`0, 1, 2, ..., N/P`	`0, N/P, 2*N/P, ...`
CPU	**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?

Mapping indices to cores (1)

Iterating for the execution space:

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

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

Important point

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

Rule of Thumb

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

Example:

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

Example: inner product (2)

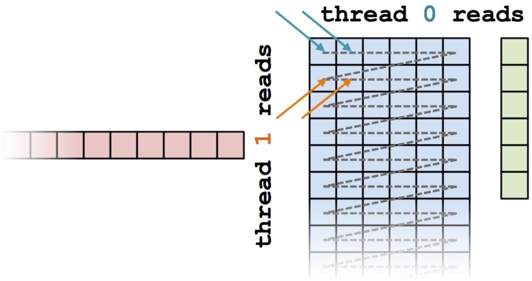
Important point

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

Important point

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

Analysis: row-major (`LayoutRight`)

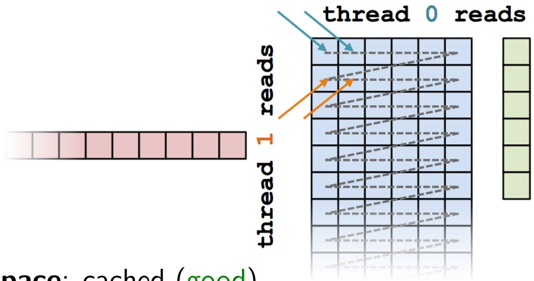


Example: inner product (2)

Important point

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

Analysis: row-major (`LayoutRight`)



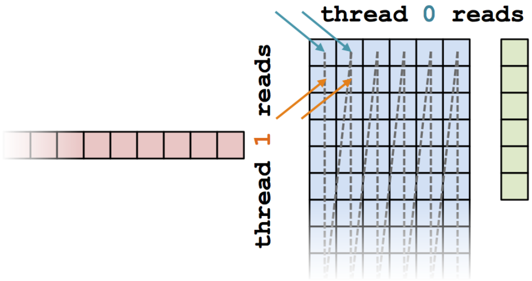
- **HostSpace**: cached (good)
- **CudaSpace**: uncoalesced (bad)

Example: inner product (3)

Important point

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

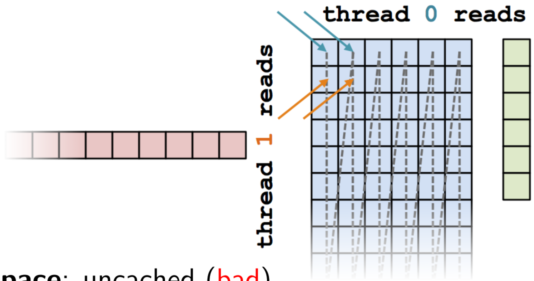
Analysis: column-major (`LayoutLeft`)



Important point

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

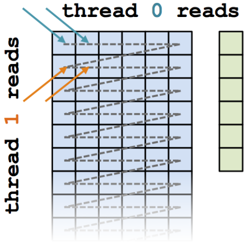
Analysis: column-major (`LayoutLeft`)



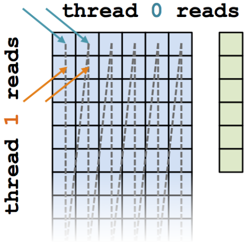
- **HostSpace**: uncached (bad)
- **CudaSpace**: coalesced (good)

Analysis: Kokkos architecture-dependent

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



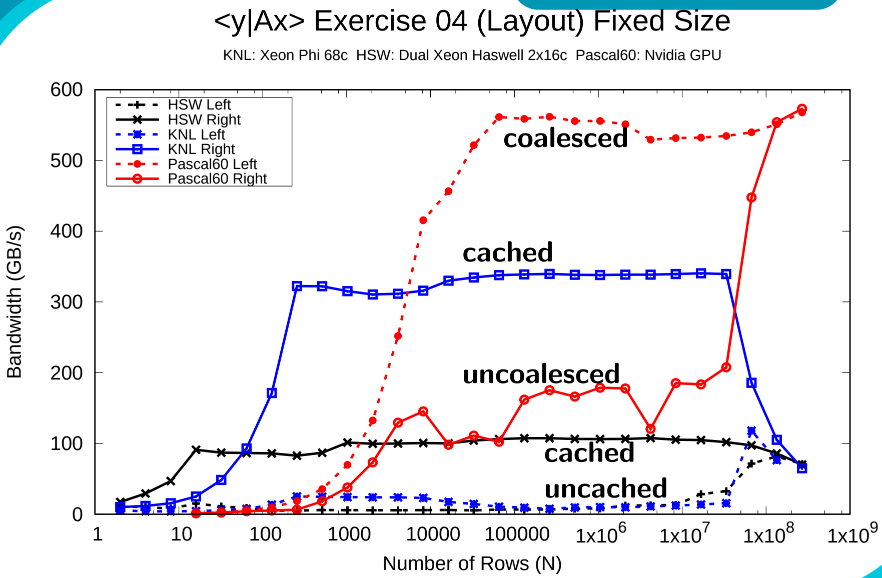
(a) OpenMP



(b) Cuda

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

Example: inner product (5)



Memory Access Pattern Summary

- 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.
  $\Rightarrow$ You'll need multiple versions of code or pay the performance penalty.

DualView

Learning objectives:

- Motivation and Value Added.
- Usage.
- Exercises.

Motivation and Value-added

- DualView was designed to help transition codes to Kokkos.

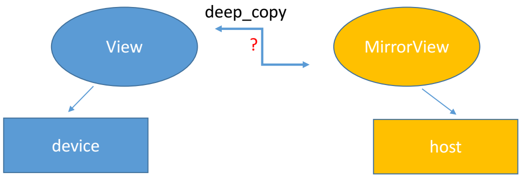
Motivation and Value-added

- DualView was designed to help transition codes to Kokkos.
- DualView simplifies the task of managing data movement between memory spaces, e.g., host and device.

Motivation and Value-added

- DualView was designed to help transition codes to Kokkos.
- DualView simplifies the task of managing data movement between memory spaces, e.g., host and device.
- When converting a typical app to use Kokkos, there is usually no holistic view of such data transfers.

DualView(1)

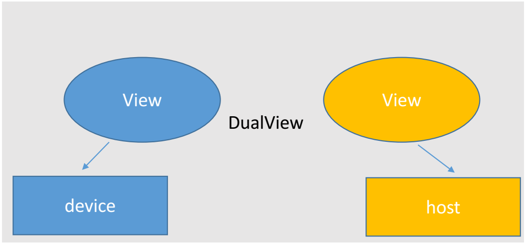


Without DualView, could use MirrorViews, but

- deep copies are expensive, use sparingly
- do I need a deep copy here?
- where is the most recent data?
- is data on the host or device stale?
- was code modified upstream? is data here now stale, but not in previous version?

DualView: Usage

DualView bundles two views, a Host View and a Device View



There is no automatic tracking of data freshness:

- you must tell Kokkos when data has been modified on a memory space.
- If you mark data as modified when you modify it, then Kokkos will know if it needs to move data

DualView bundles two views, a Host View and a Device View

- Data members for the two views

```
DualView::t_host  h_view
DualView::t_dev   d_view
```

- Retrieve data members

```
t_host  view_host();
t_dev   view_device();
```

- Mark data as modified

```
void modify_host();
void modify_device();
```

DualView bundles two views, a Host View and a Device View

- Sync data in a direction if not in sync

```
void sync_host();
void sync_device();
```

- Check sync status

```
void need_sync_host();
void need_sync_device();
```

DualView has templated functions for generic use in templated code

- Retrieve data members

```
template<class Space>
auto view();
```

- Mark data as modified

```
template<class Space>
void modify();
```

- Sync data in a direction if not in sync

```
template<class Space>
void sync();
```

- Check sync status

```
template<class Space>
void need_sync();
```

Details:

- Location: `Intro-Full/Exercises/dualview/Begin/`
- Modify or create a new compute_enthalpy function in dual_view_exercise.cpp to:
  - 1. Take (dual)views as arguments
  - 2. Call **modify()** and/or **sync()** when appropriate for the dual views
  - 3. Runs the kernel on host or device execution spaces

```
# 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
./dualview.cuda -S 26
```

Tightly Nested Loops with MDRangePolicy

Learning objectives:

- Demonstrate usage of the MDRangePolicy with tightly nested loops.
- Syntax - Required and optional settings
- Code demo and example

Motivating example: Consider the nested for loops:

```
for ( int i = 0; i < Ni; ++i )
for ( int j = 0; j < Nj; ++j )
for ( int k = 0; k < Nk; ++k )
  some_init_fcn(i, j, k);
```

Based on Kokkos lessons thus far, you might parallelize this as

```
Kokkos::parallel_for(Ni,
                     KOKKOS_LAMBDA (const i) {
                       for ( int j = 0; j < Nj; ++j )
                       for ( int k = 0; k < Nk; ++k )
                        some_init_fcn(i, j, k);
                     }
                     );
```

- This only parallelizes along one dimension, leaving potential parallelism unexploited.
- What if Ni is too small to amortize the cost of constructing a parallel region, but Ni*Nj*Nk makes it worthwhile?

MDRangePolicy (1)

Solution: Use an MDRangePolicy

```
for ( int i = 0; i < Ni; ++i )
for ( int j = 0; j < Nj; ++j )
for ( int k = 0; k < Nk; ++k )
  some_init_fcn(i, j, k);
```

Instead, use the MDRangePolicy with the parallel for

```
Kokkos::parallel_for(Kokkos::MDRangePolicy<Kokkos::Rank<3>>
                                ({0,0,0}, {Ni,Nj,Nk}),
                     KOKKOS_LAMBDA (int i, int j, int k) {
                        some_init_fcn(i, j, k);
                     }
                     );
```

Required Template Parameters to MDRangePolicy

```
Kokkos::Rank< N, IterateOuter, IterateInner >
```

- **N: (Required)** the rank of the index space (limited from 2 to 6)
- **IterateOuter (Optional)** iteration pattern between tiles
  - **Options:** Iterate::Left, Iterate::Right, Iterate::Default
- **IterateInner (Optional)** iteration pattern within tiles
  - **Options:** Iterate::Left, Iterate::Right, Iterate::Default

MDRangePolicy API(1)

Optional Template Parameters

```
ExecutionSpace
```

- **Options:** Serial, OpenMP, Threads, Cuda

```
Schedule < Options >
```

- **Options:** Static, Dynamic

```
IndexType < Options >
```

- **Options:** int, long, etc

```
WorkTag
```

- **Options:** SomeClass

```
MDRangePolicy< Rank<2,OP,IP>, OpenMP, Schedule<Static>,
                IndexType<int> > mdrpolicy;
```

Policy Arguments

```
BeginList
```

- **Initializer List or Kokkos::Array (Required):** rank arguments for starts of index space
  - **Example** Rank 2: {b0,b1}

```
EndList
```

- **Initializer List or Kokkos::Array (Required):** rank arguments for ends of index space
  - **Example** Rank 2: {e0,e1}

```
TileDimList
```

- **Initializer List or Kokkos::Array (Optional):** rank arguments for dimension of tiles
  - **Example** Rank 2: {t0,t1}

```
mdrpolicy( {b0,b1}, {e0,e1}, {t0,t1} );
```

Exercise - mdrange: Initialize multi-dim views with MDRangePolicy

Details:

- Location: `Intro-Full/Exercises/mdrange/Begin/`
- This begins with the `Solution` of 02
- Initialize the device Views x and y directly on the device using a parallel for and RangePolicy
- Initialize the device View matrix A directly on the device using a parallel for and MDRangePolicy

```
# 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
./mdrange_exercise.cuda -S 26
```

Exercise - mdrange: Initialize multi-dim views with MDRangePolicy

Things to try:

- Name the kernels - pass a string as the first argument of the parallel pattern
- Try changing the iteration patterns for the tiles in the MDRangePolicy, notice differences in performance

Subviews: Taking 'slices' of Views

Learning objectives:

- Introduce Kokkos::subview - basic capabilities and syntax
- Suggested usage and practices

Subview description:

- A subview is a 'slice' of a View that behaves as a View

Subview description:

- A subview is a 'slice' of a View that behaves as a View
  - Same syntax as a View - access data using (multi-)index entries

Subview description:

- A subview is a 'slice' of a View that behaves as a View
  - Same syntax as a View - access data using (multi-)index entries
  - The 'slice' and original View point to the same data - no extra memory allocation or copying

Subview description:

- A subview is a 'slice' of a View that behaves as a View
  - Same syntax as a View - access data using (multi-)index entries
  - The 'slice' and original View point to the same data - no extra memory allocation or copying
- Can be constructed on host or within a kernel (no allocation of memory occurs)

Subview description:

- A subview is a 'slice' of a View that behaves as a View
  - Same syntax as a View - access data using (multi-)index entries
  - The 'slice' and original View point to the same data - no extra memory allocation or copying
- Can be constructed on host or within a kernel (no allocation of memory occurs)
- Similar capability as provided by Matlab, Fortran, Python, etc. using 'colon' notation

Subviews (1)

Introductory Usage Demo:

Begin with a View:

```
Kokkos::View< double*** > v("v", N0, N1, N2);
```

Subviews (1)

Introductory Usage Demo:

Begin with a View:

```
Kokkos::View< double*** > v("v", N0, N1, N2);
```

Say we want a 2-dimensional slice at an index i0 in the first dimension - that is, in Matlab/Fortran/Python notation:

```
slicei0 = v(i0, :, :);
```

Introductory Usage Demo:

Begin with a View:

```
Kokkos::View< double*** > v("v", N0, N1, N2);
```

Say we want a 2-dimensional slice at an index i0 in the first dimension - that is, in Matlab/Fortran/Python notation:

```
slicei0 = v(i0, :, :);
```

This can be accomplished in Kokkos using a subview as follows:

```
auto slicei0 =
  Kokkos::subview(v, i0, Kokkos::ALL, Kokkos::ALL);

auto slicei0 =
  Kokkos::subview(v, i0, std::make_pair(0,v.extent(1)),
                         std::make_pair(0,v.extent(2)));
// extent(N) returns the size of dimension N of the View
```

Syntax:

```
Kokkos::subview( Kokkos::View<...> view,
                 arg0,
                 ...)
```

- **view:** First argument to the subview is the view of which a slice will be taken
- **argN:** Slice info for rank N - provide same number of arguments as rank
- **Options for argN:**
  - **index:** integral type single value
  - **partial-range:** std::pair or Kokkos::pair of integral types to provide sub-range of a rank's range [0,N)
  - **full-range:** use Kokkos::ALL rather than providing the full range as a pair

Subviews (3)

Suggested usage:

- Use 'auto' to determine the return type of a subview
- A subview can help with encapsulation - e.g. can pass into functions expecting a lower-dimensional View
- Use Kokkos::pair for partial ranges if subview created within a kernel
- Avoid usage if very few data accesses will be made to the subview
  - Construction of subview costs 20-40 operations

Exercise - Subviews: Basic usage

Details:

- Location: `Intro-Full/Exercises/subview/Begin/`
- This begins with the `Solution` of 04
- In the parallel reduce kernel, create a subview for row j of view A
- Use this subview when computing A(j,:)*x(:) rather than the matrix A

```
# 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
./subview_exercise.cuda -S 26
```

Thread safety and atomic operations

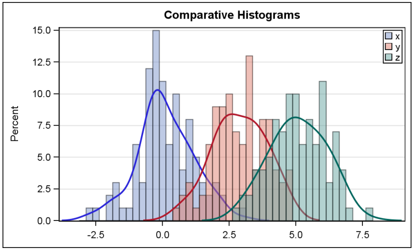
Learning objectives:

- Understand that coordination techniques for low-count CPU threading are not scalable.
- Understand how atomics can parallelize the **scatter-add** pattern.
- Gain **performance intuition** for atomics on the CPU and GPU, for different data types and contention rates.

Examples: Histogram

Histogram kernel:

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

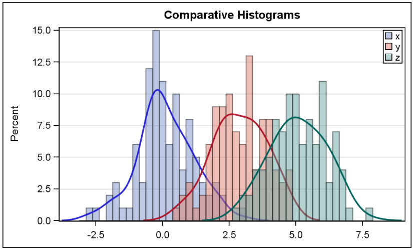


http://www.farmaceuticas.com.br/tag/graficos/

Histogram kernel:

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

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



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

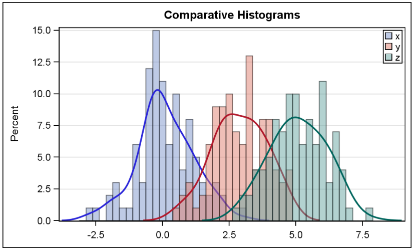
Histogram kernel:

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const size_t 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 are **not portable**.

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 are **not portable**.
- Data replication is **not thread scalable**.

Performance of atomics (0)

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?

Performance of atomics (1)

Two costs: (independent) work and coordination.

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

Performance of atomics (1)

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);
}
```

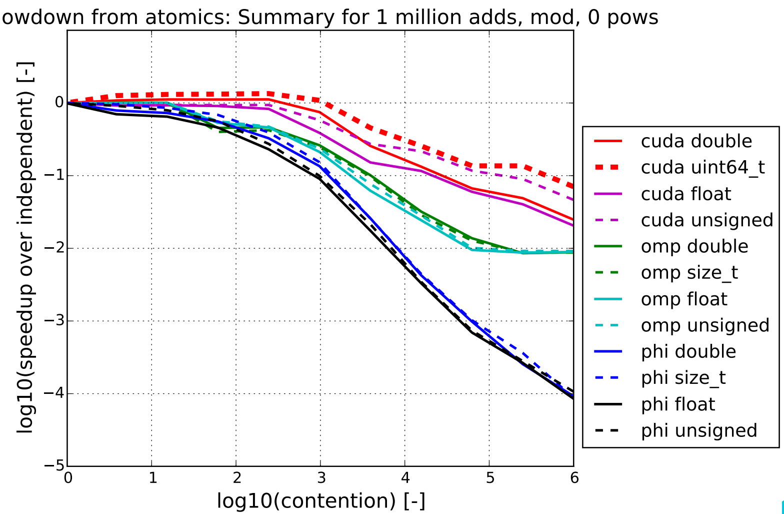
- This is the most extreme case: all coordination and no work.
- Contention is captured by the `atomicStride`.

  `atomicStride` $\rightarrow$ 1 $\Rightarrow$ Scalar integration (bad)

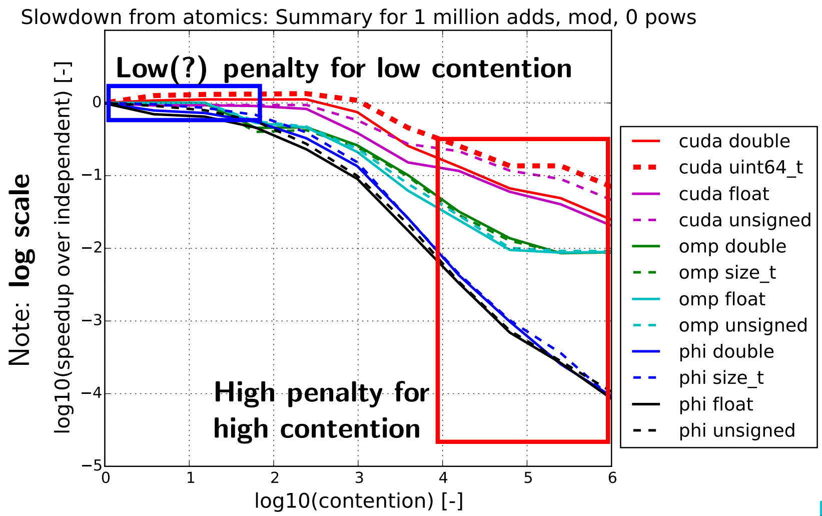
  `atomicStride` $\rightarrow$ large $\Rightarrow$ Independent (good)

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

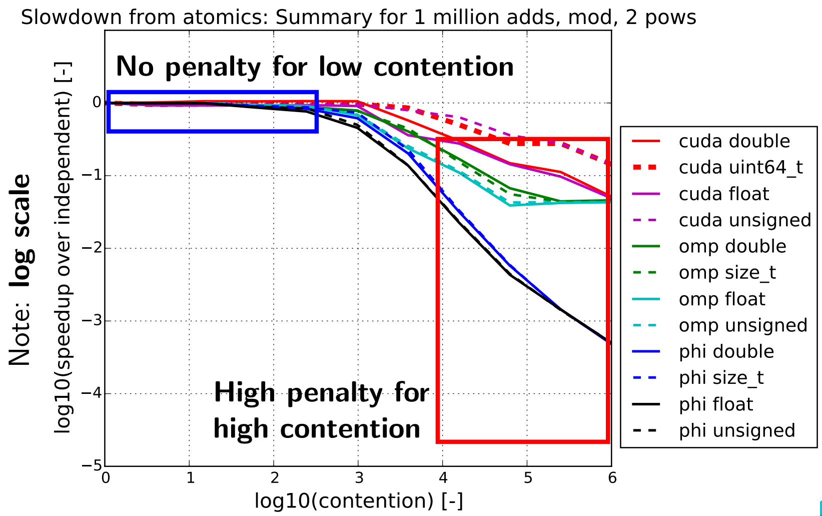
Note: **log scale**



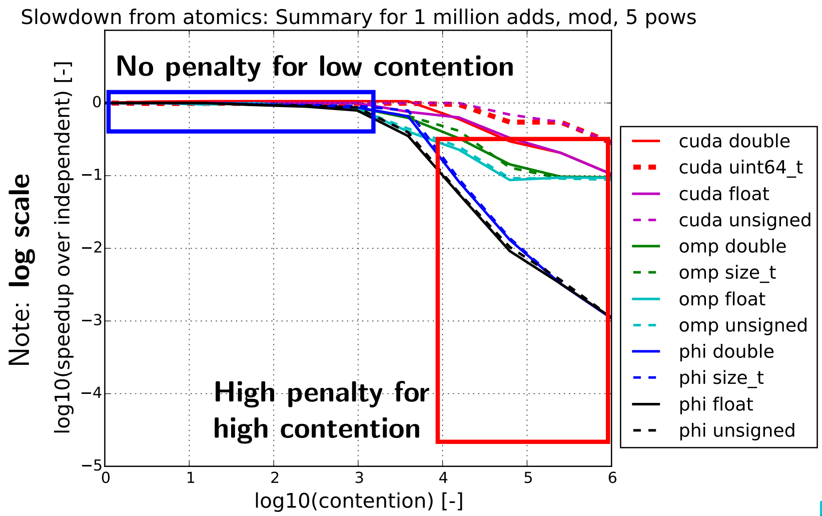
Atomics performance: 1 million adds, **no** work per kernel



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



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



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(...);
```

Scatter Contribute (1)

Histogram generation is an example of the **Scatter Contribute** pattern.

- Like a reduction but with many results.
- Number of results scales with number of inputs.
- Each results gets contributions from a small number of inputs/iterations.
- Uses an inputs-to-results map not inverse.

Examples:

- Particles contributing to neighbors forces.
- Cells contributing forces to nodes.
- Computing histograms.
- Computing a density grid from point source contributions.

Scatter Contribute (2)

There are two useful algorithms:.

- **Atomics:** thread-scalable but depends on atomic performance.
- **Data Replication:** every thread owns a copy of the output, not thread-scalable but good for low (< 16) threads count architectures.

Scatter Contribute (2)

There are two useful algorithms:.

- **Atomics:** thread-scalable but depends on atomic performance.
- **Data Replication:** every thread owns a copy of the output, not thread-scalable but good for low (< 16) threads count architectures.

Important Capability: ScatterView

ScatterView can transparently switch between **Atomic** and **Data Replication** based scatter algorithms.

Scatter Contribute (2)

There are two useful algorithms:.

- **Atomics:** thread-scalable but depends on atomic performance.
- **Data Replication:** every thread owns a copy of the output, not thread-scalable but good for low (< 16) threads count architectures.

Important Capability: ScatterView

ScatterView can transparently switch between **Atomic** and **Data Replication** based scatter algorithms.

- Abstracts over scatter contribute algorithms.
- Compile time choice with backend-specific defaults.
- Only limited number of operations are supported.
- Part of Kokkos Containers.

Scatter Contribute (3)

Example:

```
// Begin with a normal View
Kokkos::View<double*> results("results",N);
// Create a scatter view wrapping the original view
Kokkos::Experimental::ScatterView<double*> scatter(results);
// Reset contributions if necessary
scatter.reset();
// Start parallel operation
Kokkos::parallel_for("ScatterAlg", M,
  KOKKOS_LAMBDA(int i) {
  // Get the accessor - e.g. the thread specific copy
  // or an atomic view of the data.
  auto access = scatter.access();

  for(int j=0; j<num_neighs(i); j++) {
    // Get the destination index
    int neigh = neighbors(i,j);
    // Add the contribution
    access(neigh) += contribution(i,neigh);
  }
});
// Combine the results - no op if ScatterView was using atomics in
Kokkos::Experimental::contribute(results,scatter);
```

Exercise ScatterView

- Location: `Intro-Full/Exercises/scatter_view/Begin/`
- Assignment: Convert scatter_view_loop to use `ScatterView`.
- Compile and run on both CPU and GPU

```
make -j KOKKOS_DEVICES=OpenMP  # CPU-only using OpenMP
make -j KOKKOS_DEVICES=Cuda    # GPU - note UVM in Makefile
# Run exercise
./scatterview.host
./scatterview.cuda
# Note the warnings, set appropriate environment variables
```

- Compare performance on CPU of the three variants
- Compare performance on GPU of the two variants
- Vary problem size: first and second optional argument

Section Summary

- 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.

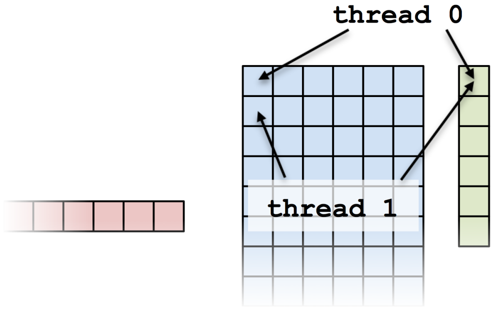
Learning objectives:

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

Example: inner product (0)

(Flat parallel) Kernel:

```
Kokkos::parallel_reduce("yAx",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);
```

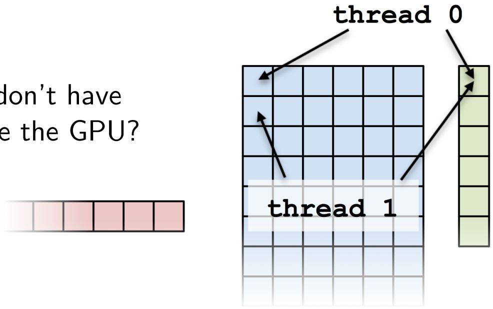


Example: inner product (0)

(Flat parallel) Kernel:

```
Kokkos::parallel_reduce("yAx",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);
```

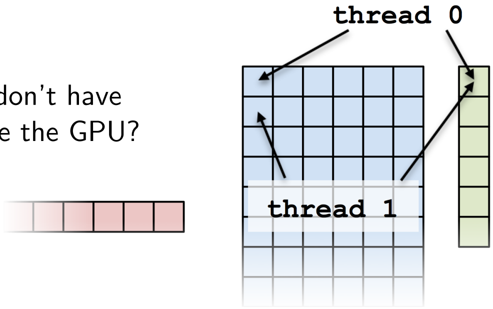
Problem: What if we don't have enough rows to saturate the GPU?



Example: inner product (0)

(Flat parallel) Kernel:

```
Kokkos::parallel_reduce("yAx",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);
```



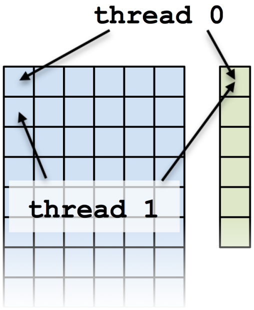
Problem: What if we don't have enough rows to saturate the GPU?

Solutions?

Example: inner product (0)

(Flat parallel) Kernel:

```
Kokkos::parallel_reduce("yAx",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?

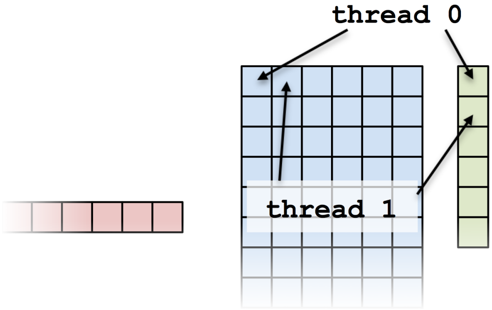
Solutions?

- Atomics
- Thread teams

Example: inner product (1)

Atomics kernel:

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

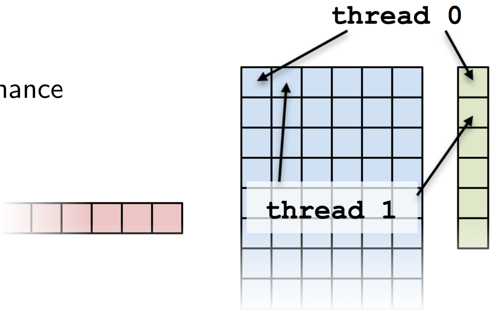


Example: inner product (1)

Atomics kernel:

```
Kokkos::parallel_for("yAx", 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



Example: inner product (2)

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);
}
```

Example: inner product (2)

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**.

Example: inner product (3)

Important concept: Thread team

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

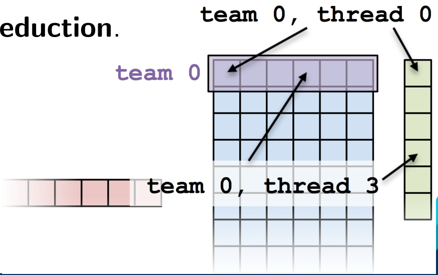
Example: inner product (3)

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**.



Example: inner product (4)

The final hierarchical parallel kernel:

```
parallel_reduce("yAx",
  team_policy(N, Kokkos::AUTO),

  KOKKOS_LAMBDA (const 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);
```

TeamPolicy (0)

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("Label",
  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("Label",
  RangePolicy<ExecutionSpace>(0,N), functor);
```

"**Hierarchical** parallelism" uses `TeamPolicy`:

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

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

TeamPolicy (1)

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();
}
```

TeamPolicy (1)

Important point

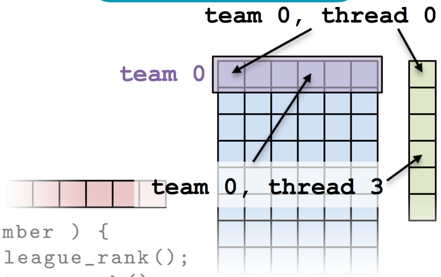
When using teams, functor operators receive a *team member*.

```
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

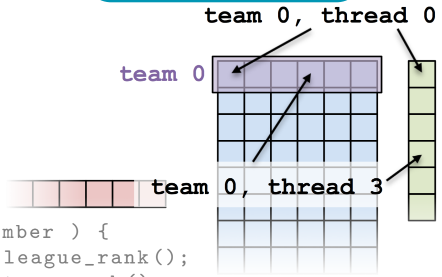
TeamThreadRange (0)



First attempt at exercise:

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

TeamThreadRange (0)



First attempt at exercise:

```
operator() (member_type & teamMember ) {
  const size_t row = teamMember.league_rank();
  const size_t 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?

TeamThreadRange (1)

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

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

TeamThreadRange (1)

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

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

If this were a parallel execution,

we'd use Kokkos::parallel_reduce.

TeamThreadRange (1)

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

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

If this were a parallel execution,

we'd use `Kokkos::parallel_reduce`.

Key idea: this *is* a parallel execution.

TeamThreadRange (1)

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

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

If this were a parallel execution,

we'd use `Kokkos::parallel_reduce`.

Key idea: this *is* a parallel execution.

$\Rightarrow$ **Nested parallel patterns**

TeamThreadRange (2)

TeamThreadRange:

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

TeamThreadRange (2)

TeamThreadRange:

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

- The **mapping** of work indices to threads is **architecture-dependent**.
- The **amount of work** given to the TeamThreadRange **need not be a multiple** of the team_size.
- Intrateam **reduction handled** by Kokkos.

Anatomy of nested parallelism:

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

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

What should the team size be?

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

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

What should the team size be?

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

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

GPUs

- Special hardware available for coordination within a team.
- Within a team 32 (NVIDIA) or 64 (AMD) threads execute "lock step."
- Maximum team size: **1024**; Recommended team size: **128/256**

What should the team size be?

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

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

GPUs

- Special hardware available for coordination within a team.
- Within a team 32 (NVIDIA) or 64 (AMD) threads execute "lock step."
- Maximum team size: **1024**; Recommended team size: **128/256**

Intel Xeon Phi:

- Recommended team size: # hyperthreads per core
- Hyperthreads share entire cache hierarchy
  a well-coordinated team avoids cache-thrashing

Exercise #5: Inner Product, Hierarchical Parallelism

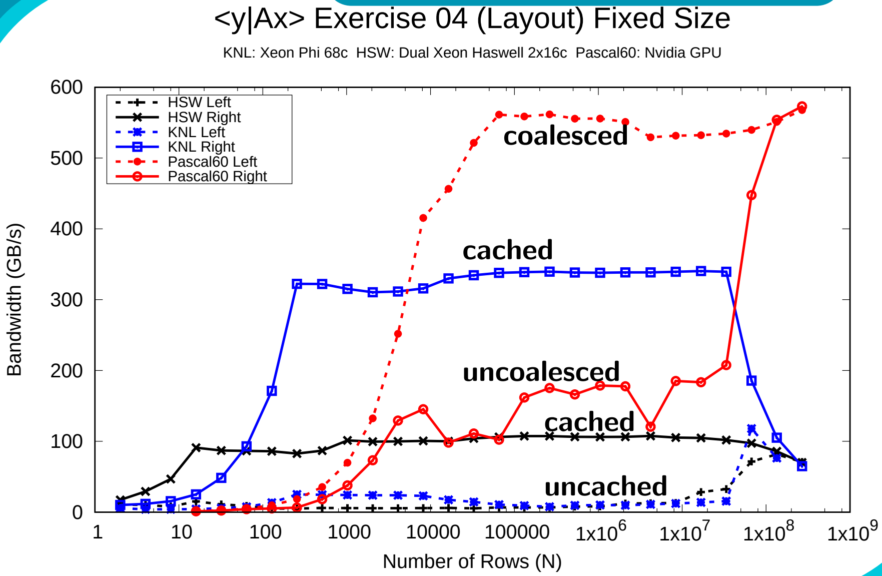
Details:

- Location: `Intro-Full/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?

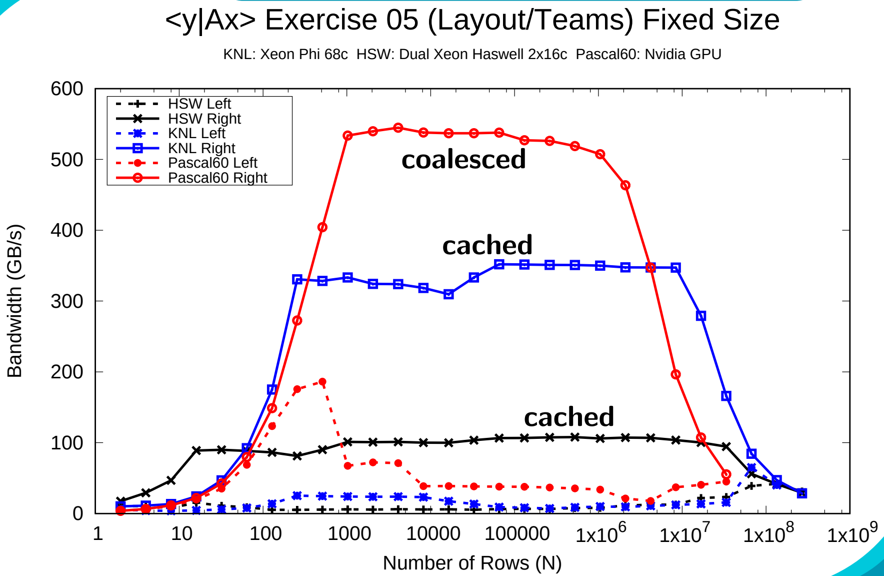
Things to try:

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

Reminder, Exercise #4 with Flat Parallelism



Exercise #5: Inner Product, Hierarchical Parallelism



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.
- `TeamVectorRange` uses both **thread** and **vector** parallelism.

Anatomy of nested parallelism:

```
parallel_outer("Label",
  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 */
      }[, ...]);
    parallel_middle(
    TeamVectorRange(teamMember, someSize),
      [=] (const int indexTeamVector[, ...]) {
        /* nested body */
      }[, ...]);
    /* end of outer body */
  }[, ...]);
```

Sum sanity checks (0)

Question: What will the value of `totalSum` be?

```
int totalSum = 0;
parallel_reduce("Sum", RangePolicy<>(0, numberOfThreads),
  KOKKOS_LAMBDA (size_t& index, int& partialSum) {
    int thisThreadsSum = 0;
    for (int i = 0; i < 10; ++i) {
      ++thisThreadsSum;
    }
    partialSum += thisThreadsSum;
}, totalSum);
```

Sum sanity checks (0)

Question: What will the value of `totalSum` be?

```
int totalSum = 0;
parallel_reduce("Sum", RangePolicy<>(0, numberOfThreads),
  KOKKOS_LAMBDA (size_t& index, int& partialSum) {
    int thisThreadsSum = 0;
    for (int i = 0; i < 10; ++i) {
      ++thisThreadsSum;
    }
    partialSum += thisThreadsSum;
}, totalSum);
```

`totalSum = numberOfThreads * 10`

Question: What will the value of `totalSum` be?

```
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>(numberOfTeams, team_size),
  KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
    int thisThreadsSum = 0;
    for (int i = 0; i < 10; ++i) {
      ++thisThreadsSum;
    }
    partialSum += thisThreadsSum;
}, totalSum);
```

Sum sanity checks (1)

Question: What will the value of `totalSum` be?

```
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>(numberOfTeams, team_size),
  KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
    int thisThreadsSum = 0;
    for (int i = 0; i < 10; ++i) {
      ++thisThreadsSum;
    }
    partialSum += thisThreadsSum;
}, totalSum);
```

`totalSum = numberOfTeams * team_size * 10`

Sum sanity checks (2)

Question: What will the value of `totalSum` be?

```
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>(numberOfTeams, team_size),
  KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
    int thisTeamsSum = 0;
    parallel_reduce(TeamThreadRange(teamMember, team_size),
      [=] (const int index, int& thisTeamsPartialSum) {
      int thisThreadsSum = 0;
      for (int i = 0; i < 10; ++i) {
        ++thisThreadsSum;
      }
      thisTeamsPartialSum += thisThreadsSum;
    }, thisTeamsSum);
    partialSum += thisTeamsSum;
}, totalSum);
```

Sum sanity checks (2)

Question: What will the value of `totalSum` be?

```
int totalSum = 0;
parallel_reduce("Sum", TeamPolicy<>(numberOfTeams, team_size),
  KOKKOS_LAMBDA (member_type& teamMember, int& partialSum) {
    int thisTeamsSum = 0;
    parallel_reduce(TeamThreadRange(teamMember, team_size),
      [=] (const int index, int& thisTeamsPartialSum) {
      int thisThreadsSum = 0;
      for (int i = 0; i < 10; ++i) {
        ++thisThreadsSum;
      }
      thisTeamsPartialSum += thisThreadsSum;
    }, thisTeamsSum);
    partialSum += thisTeamsSum;
}, totalSum);
```

`totalSum = numberOfTeams * team_size * team_size * 10`

Restricting Execution: single pattern

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 broadcastedValue = 0;
single(PerTeam(teamMember), [&] (int& broadcastedValue_local) {
        broadcastedValue_local = special value assigned by one;
}, broadcastedValue);
// Now everyone has the special value
```

Exercise #6: Three-Level Parallelism

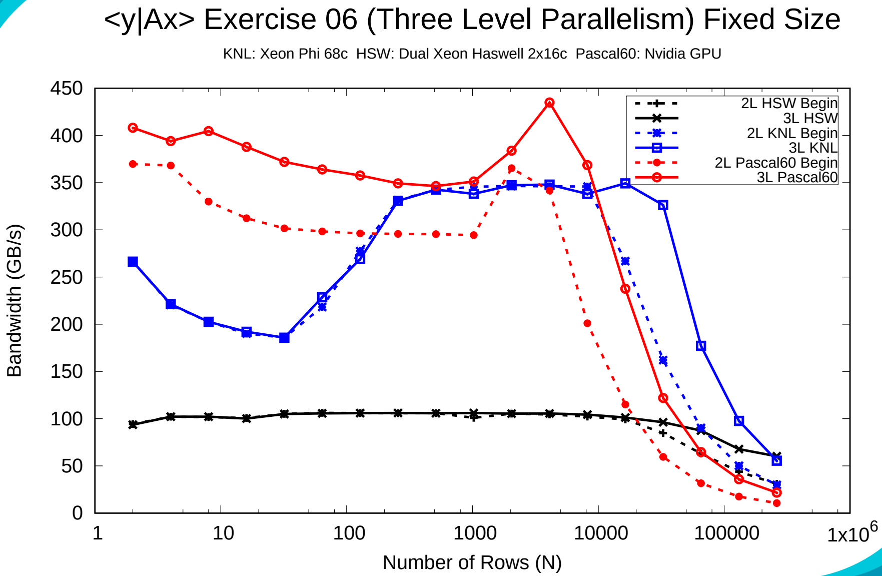
The previous example was extended with an outer loop over "Elements" to expose a third natural layer of parallelism.

Details:

- Location: `Intro-Full/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 behavior with Exercise 5 for very non-square matrices
- Compare behavior of CPU vs GPU



<y|Ax> Exercise 06 (Three Level Parallelism) Fixed Size
KNL: Xeon Phi 68c HSW: Dual Xeon Haswell 2x16c Pascal60: Nvidia GPU
Bandwidth (GB/s)
0
50
100
150
200
250
300
350
400
450
Number of Rows (N)
1
10
100
1000
10000
100000
1x10^6
2L HSW Begin
3L HSW
2L KNL Begin
3L KNL
2L Pascal60 Begin
3L Pascal60

Section Summary

- **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

Types of Scratch Space Uses

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).

Types of Scratch Space Uses

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.

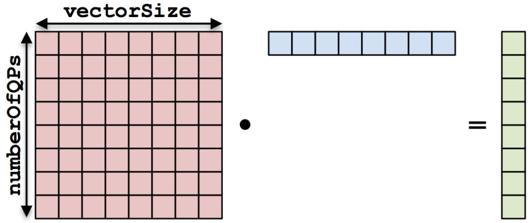
Manually Managed Cache

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

Now: Discuss Manually Managed Cache Usecase.

Example: contractDataFieldScalar (1)

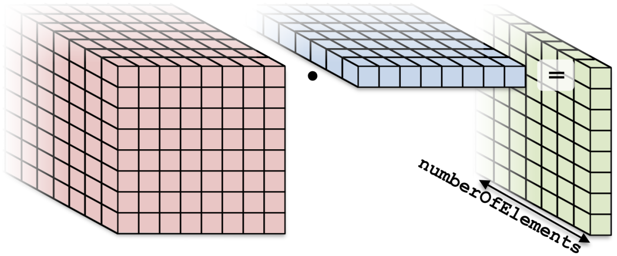
One slice of contractDataFieldScalar:



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

Example: contractDataFieldScalar (2)

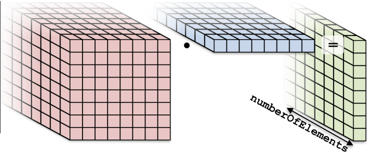
contractDataFieldScalar:



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

Example: contractDataFieldScalar (3)

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

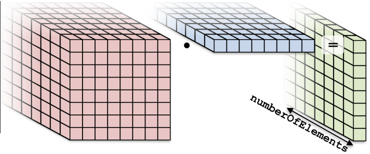


Parallelization approaches:

- Each thread handles an `element`.
  Threads: `numberOfElements`

Example: contractDataFieldScalar (3)

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

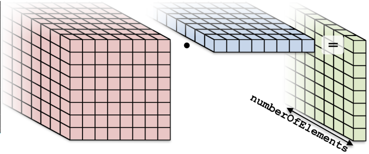


Parallelization approaches:

- Each thread handles an `element`.
  Threads: `numberOfElements`
- Each thread handles a `qp`.
  Threads: `numberOfElements * numberOfQPs`

Example: contractDataFieldScalar (3)

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

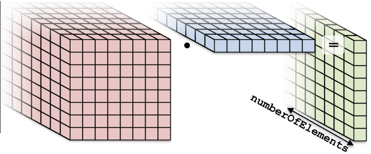


Parallelization approaches:

- Each thread handles an `element`.
  Threads: `numberOfElements`
- Each thread handles a `qp`.
  Threads: `numberOfElements * numberOfQPs`
- Each thread handles an `i`.
  Threads: `numElements * numQPs * vectorSize`
  Requires a `parallel_reduce`.

Example: contractDataFieldScalar (3)

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

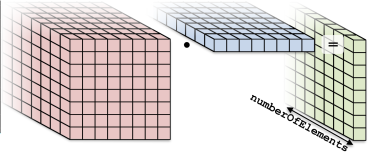


Parallelization approaches:

- Each thread handles an `element`.
  Threads: `numberOfElements`
- Each thread handles a `qp`.
  Threads: `numberOfElements * numberOfQPs`
- Each thread handles an `i`.
  Threads: `numElements * numQPs * vectorSize`
  Requires a `parallel_reduce`.

Example: contractDataFieldScalar (4)

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

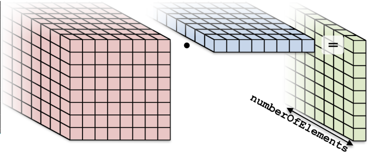


Flat kernel: Each thread handles a quadrature point

```
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;
}
```

Example: contractDataFieldScalar (6)

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

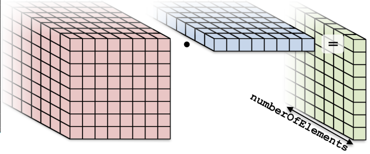


Teams kernel: Each team handles an element

```
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;
    });
}
```

Example: contractDataFieldScalar (6)

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



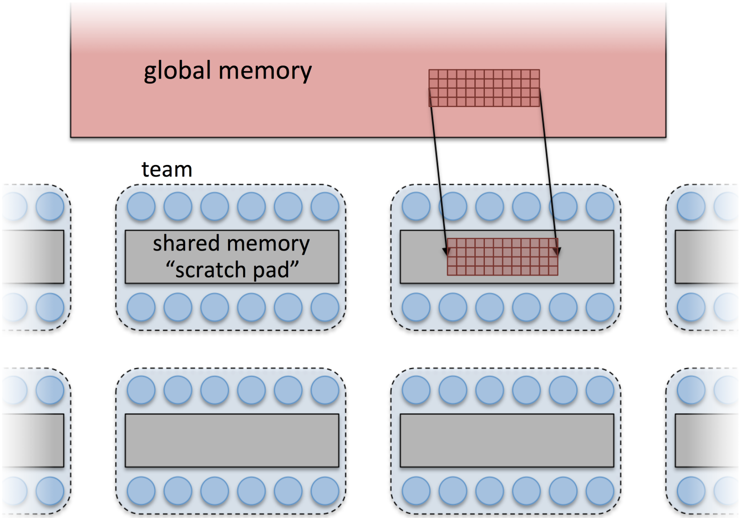
Teams kernel: Each team handles an element

```
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)

Scratch memory (0)

Each team has access to a “scratch pad”.



Scratch memory (1)

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 (1)

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 (2)

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 x W.
- **With scratch memory:** Kokkos pre-allocates space for each Team or Thread of size T x W.
- `PerThread` and `PerTeam` scratch can be used concurrently.
- Level 0 and Level 1 scratch memory can be used concurrently.

Scratch memory (2)

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 x W.
- **With scratch memory:** Kokkos pre-allocates space for each Team or Thread of size T x 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, then use Kokkos' scratch memory.

Scratch memory (3)

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.

Scratch memory (3)

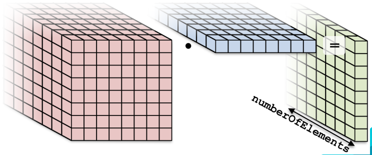
To use scratch memory, you need to:

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

```
TeamPolicy<ExecutionSpace> policy(numberOfTeams, teamSize);

// Define a scratch memory view type
typedef View<double*,ExecutionSpace::scratch_memory_space
                    ,MemoryUnmanaged> ScratchPadView;
// Compute how much scratch memory (in bytes) is needed
size_t bytes = ScratchPadView::shmem_size(vectorSize);

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

    // Create a view from the pre-existing scratch memory
    ScratchPadView scratch(teamMember.team_scratch(level),
                           vectorSize);
});
```

Kernel outline for teams with scratch memory:

```
operator()(member_type teamMember) {
  ScratchPadView scratch(teamMember.team_scratch(0),
                         vectorSize);

  // TODO: 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;
    });
}
```

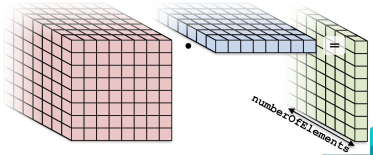


Example: contractDataFieldScalar (8)

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);
  }
}
```



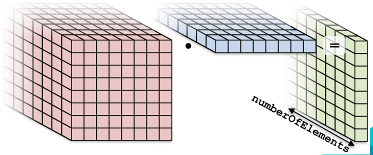
How to populate the scratch memory?

- ~~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);
```



Example: contractDataFieldScalar (8)

How to populate the scratch memory?

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

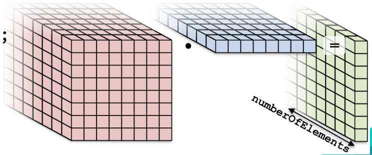
```
if (teamMember.team_rank() == 0) {
  for (int i = 0; i < vectorSize; ++i) {
    scratch(i) = B(element, i);
  }
}
```

- ~~Each thread loads one entry?~~ teamSize $\neq$ vectorSize

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

- TeamThreadRange or ThreadVectorRange

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



How to populate the scratch memory?

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

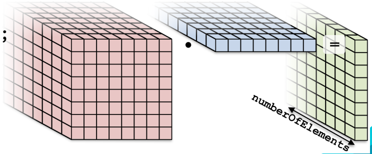
```
if (teamMember.team_rank() == 0) {
  for (int i = 0; i < vectorSize; ++i) {
    scratch(i) = B(element, i);
  }
}
```

- ~~Each thread loads one entry?~~ teamSize $\neq$ vectorSize

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

- TeamThreadRange or ThreadVectorRange

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



Example: contractDataFieldScalar (9)

(incomplete) Kernel for teams with scratch memory:

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

  parallel_for(ThreadVectorRange(teamMember, vectorSize),
    [=] (int i) {
      scratch(i) = B(element, i);
    });
  // TODO: fix a problem at this location

  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:

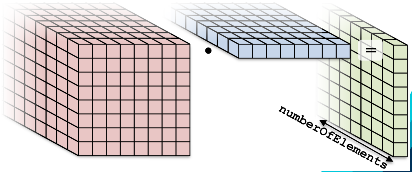
```
operator()(member_type teamMember) {
  ScratchPadView scratch(...);

  parallel_for(ThreadVectorRange(teamMember, vectorSize),
    [=] (int i) {
      scratch(i) = B(element, i);
    });
  // TODO: fix a problem at this location

  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.

Example: contractDataFieldScalar (10)

Kernel for teams with scratch memory:

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

  parallel_for(ThreadVectorRange(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;
    });
}
```



Exercise #7: Scratch Memory

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

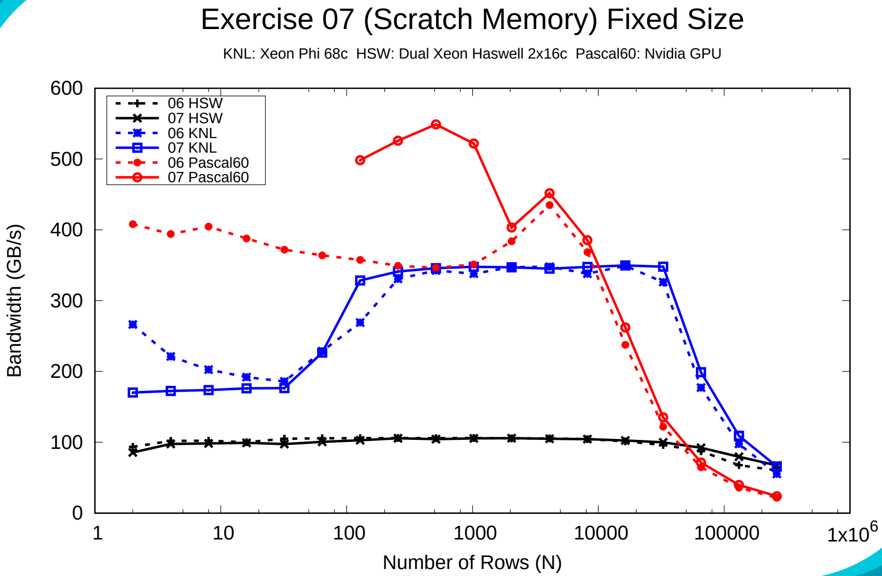
Details:

- Location: `Intro-Full/Exercises/07/`
- Create a scratch view
- Fill the scratch view in parallel using a TeamThreadRange or ThreadVectorRange

Things to try:

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

Exercise #7: Scratch Memory



Scratch Memory: API Details

Allocating scratch in different levels:

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

Scratch Memory: API Details

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));
```

Scratch Memory: API Details

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.

Section Summary

- **Scratch Memory** can be use with the `TeamPolicy` to provide thread or team **private** memory.
- Usecase: 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.

Task parallelism

Fine-grained dependent execution.

Learning objectives:

- Basic interface for fine-grained tasking in Kokkos
- How to express dynamic dependency structures in Kokkos tasking
- When to use Kokkos tasking

Task Parallelism Looks Like Data Parallelism

Recall that **data parallel** code is composed of a pattern, a policy, and a functor

```
Kokkos::parallel_for(
  Kokkos::RangePolicy<>(exec_space, 0, N),
  SomeFunctor()
);
```

Task parallel code similarly has a pattern, a policy, and a functor

```
Kokkos::task_spawn(
  Kokkos::TaskSingle(scheduler, TaskPriority::High),
  SomeFunctor()
);
```

What does a task functor look like?

```
struct MyTask {
  using value_type = double;
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result);
};
```

- Tell Kokkos what the value type of your task's output is.
- Take a team member argument, analogous to the team member passed in by `Kokkos::TeamPolicy` in hierarchical parallelism
- The output is expressed by assigning to a parameter, similar to with `Kokkos::parallel_reduce`

What policies does Kokkos tasking provide?

- `Kokkos::TaskSingle()`
  - Run the task with a single worker thread
- `Kokkos::TaskTeam()`
  - Run the task with all of the threads in a team
  - Think of it like being inside of a `parallel_for` with a `TeamPolicy`
- Both policies take a scheduler, an optional predecessor, and an optional priority (more on schedulers and predecessors later)

What patterns does Kokkos tasking provide?

- `Kokkos::task_spawn()`
  - `Kokkos::host_spawn()` (same thing, but from host code)
- `Kokkos::respawn()`
  - Argument order is backwards; policy comes second!
  - First argument is ‘this‘ always (not ‘*this‘)
- `task_spawn()` and `host_spawn()` return a `Kokkos::Future` representing the completion of the task (see next slide), which can be used as a predecessor to another operation.

How do futures and dependencies work?

```
struct MyTask {
  using value_type = double;
  Kokkos::Future<double, Kokkos::DefaultExecutionSpace> dep;
  int depth;
  KOKKOS_INLINE_FUNCTION MyTask(int d) : depth(d) { }
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result) {
    if(depth == 1) result = 3.14;
    else if(dep.is_null()) {
      dep =
        Kokkos::task_spawn(
          Kokkos::TaskSingle(member.scheduler()),
          MyTask(depth-1)
        );
      Kokkos::respawn(this, dep);
    }
    else {
      result = depth * dep.get();
    }
  }
};
```

The Scheduler Abstraction

```
template <class Scheduler>
struct MyTask {
  using value_type = double;
  Kokkos::BasicFuture<double, Scheduler> dep;
  int depth;
  KOKKOS_INLINE_FUNCTION MyTask(int d) : depth(d) { }
  template <class TeamMember>
  KOKKOS_INLINE_FUNCTION
  void operator()(TeamMember& member, double& result);
};
```

Available Schedulers:

- `TaskScheduler<ExecSpace>`
- `TaskSchedulerMultiple<ExecSpace>`
- `ChaseLevTaskScheduler<ExecSpace>`

Spawning from the host

```
using execution_space = Kokkos::DefaultExecutionSpace;
using scheduler_type = Kokkos::TaskScheduler<execution_space>;
using memory_space = scheduler_type::memory_space;
using memory_pool_type = scheduler_type::memory_pool;
size_t memory_pool_size = 1 << 22;

auto scheduler =
  scheduler_type(memory_pool_type(memory_pool_size));

Kokkos::BasicFuture<double, scheduler_type> result =
  Kokkos::host_spawn(
    Kokkos::TaskSingle(scheduler),
    MyTask<scheduler_type>(10)
  );
Kokkos::wait(scheduler);
printf("Result␣is␣%f", result.get());
```

Things to Keep in Mind

- Tasks always run to completion
- There is no way to wait or block inside of a task
  - `future.get()` does not block!
- Tasks that do not `respawn` themselves are complete
  - The value in the `result` parameter is made available through `future.get()` to any dependent tasks.
- The second argument to `respawn` can only be either a predecessor (future) or a scheduler, not a proper execution policy
  - We are fixing this to provide a more consistent overload in the next release.
- Tasks can only have one predecessor (at a time)
  - Use `scheduler.when_all()` to aggregate predecessors (see next slide)

```
using void_future =
  Kokkos::BasicFuture<void, scheduler_type>;
auto f1 =
  Kokkos::task_spawn(Kokkos::TaskSingle(scheduler), X{});
auto f2 =
  Kokkos::task_spawn(Kokkos::TaskSingle(scheduler), Y{});
void_future f_array[] = { f1, f2 };
void_future f_12 = scheduler.when_all(f_array, 2);
auto f3 =
  Kokkos::task_spawn(
    Kokkos::TaskSingle(scheduler, f_12), FuncXY{}
  );
```

- To create an aggregate `Future`, use `scheduler.when_all()`
- `scheduler.when_all()` always returns a `void` future.
- (Also, any future is implicitly convertible to a `void` future of the same `Scheduler` type)

Exercise #8: Fibonacci

Formula

$$F_N = F_{N-1} + F_{N-2}$$
$$F_0 = 0$$
$$F_1 = 1$$

Serial algorithm

```
int fib(int n) {
  if(n < 2) return n;
  else {
    return fib(n-1) + fib(n-2);
  }
}
```

Details:

- Location: `Intro-Full/Exercises/08`
- Implement the `FibonacciTask` task functor recursively
- Spawn the root task from the host and wait for the scheduler to make it ready

Hints:

- Do the F_{N-1} and F_{N-2} subproblems in separate tasks
- Use a `scheduler.when_all()` to wait on the subproblems

SIMD

Portable vector intrinsic types.

Learning objectives:

- How to use SIMD types to improve vectorization.
- SIMD Types as an alternative to ThreadVector loops.
- SIMD Types to achieve outer loop vectorization.

So far there were two options for achieving vectorization:

- **Hope For The Best**: Kokkos semantics make loops inherently vectorizable, sometimes the compiler figures it even out.
- **Hierarchical Parallelism**: `TeamVectorRange` and `ThreadVectorRange` help the compiler with hints such as `#pragma ivdep` or `#pragma omp simd`.

These strategies do run into limits though:

- Compilers often do not vectorize loops on their own.
- An optimal vectorization strategy would require *outer-loop vectorization*.
- Vectorization with `TeamVectorRange` sometimes requires artifically introducing an additional loop level.

Outer-Loop Vectorization

A simple scenario where for outer-loop vectorization:

```
for(int i=0; i<N; i++) {
  // expect K to be small odd 1,3,5,7 for physics reasons
  for(int k=0; k<K; k++) b(i) += a(i,k);
}
```

Vectorization the K-Loop is not profitable:

- It is a short reduction.
- Remainders will eat up much time.

Using ThreadVectorRange is cumbersome and requires split of N-Loop:

```
parallel_for("VectorLoop",TeamPolicy<>(0,N/V,V),
  KOKKOS_LAMBDA ( const team_t& team ) {
  int i = team.league_rank() * V;
  for(int k=0; k<K; k++)
    parallel_for(ThreadVectorRange(team,V), [&](int ii) {
      b(i+ii) += a(i+ii,k);
    });
});
```

SIMD Types

To help with this situation and (in particular in the past) fix the lack of auto-vectorizing compilers `SIMD-Types` have been invented. They:

- Are short vectors of scalars.
- Have operators such as `+=` so one can use them like scalars.
- Are compile time sized.
- Usually map directly to hardware vector instructions.

Important concept: SIMD Type

A SIMD variable is a **short vector** which acts like a scalar.

Using such a `simd` type one can simply achieve *outer-loop* vectorization by using arrays of `simd` and dividing the loop range by its *size*.

The ISO C++ standard has a *Technical Specification* for `simd` (in *parallelism v2*):

```
template< class T, class Abi >
class simd {
public:
  using value_type = T;
  using reference = /* impl defined */;
  using abi_type = Abi;
  static constexpr size_t size();
  void copy_from(T const*, aligned_tag);
  void copy_to(T*, aligned_tag) const;
  T& operator[] (size_t);
  //Element wise operators
};

// Element Wise non-member operators
```

C++23? SIMD ABI

One interesting innovation here is the `Abi` parameter allowing for different, hardware specific, implementations.

The most important in the proposal are:

- **scalar**: single element type.
- **fixed_size**$< N >$: stores `N` elements.
- **max_fixed_size**$< T >$: stores maximum number of elements for `T`.
- **native**: best fit for hardware.

C++23? SIMD ABI

One interesting innovation here is the `Abi` parameter allowing for different, hardware specific, implementations.

The most important in the proposal are:

- **scalar**: single element type.
- **fixed_size**$< N >$: stores `N` elements.
- **max_fixed_size**$< T >$: stores maximum number of elements for `T`.
- **native**: best fit for hardware.

But `std::experimental::simd` is not in the standard yet, and doesn't support GPUs ...

C++23? SIMD ABI

One interesting innovation here is the `Abi` parameter allowing for different, hardware specific, implementations.

The most important in the proposal are:

- **scalar**: single element type.
- **fixed_size**$< N >$: stores `N` elements.
- **max_fixed_size**$< T >$: stores maximum number of elements for `T`.
- **native**: best fit for hardware.

But `std::experimental::simd` is not in the standard yet, and doesn't support GPUs ...

It also has other problems making it insufficient for our codes ...

Just at Sandia we had at least **5** different SIMD types in use.

A unification effort was started with the goal of:

- Match the proposed `std::simd` API as far as possible.
- Support GPUs.
- Can be used stand-alone or in conjunction with Kokkos.
- Replaces all current implementations at Sandia for SIMD.

We now have an implementation developed by Dan Ibanez, which is close to meeting all of those criteria:

- For now available at `https://github.com/kokkos/simd-math`.
- Considered Experimental, but supports X86, ARM, Power, NVIDIA GPUs.
- Will be integrated into Kokkos in the next two months.

Exercise #9: Simple SIMD usage.

Details:

- Location: `Intro-Full/Exercises/09/Begin/`
- Include the `simd.hpp` header.
- Change the data type of the views to use `simd::simd<double,simd::simd_abi:native>`.
- Create an unmanaged `View<double*>` of `results` using the `data()` function for the final reduction.

```
# Compile for CPU
make -j KOKKOS_DEVICES=OpenMP
# Compile for GPU
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./simd.cuda
```

Things to try:

- Vary problem size (-N ...; -K ...)
- Compare behavior of scalar vs vectorized on CPU and GPU

The GPU SIMD Problem

The above exercise used a **scalar** simd type on the **GPU**.
Why wouldn't we use a fixed_size instead?

- Using a `fixed_size` ABI will create a scalar of size `N` in each CUDA thread!
- Loading a `fixed_size` variable from memory would result in uncoalesced access.
- If you have correct layouts you get `outer-loop` vectorization implicitly on GPUs.

The GPU SIMD Problem

The above exercise used a **scalar** simd type on the **GPU**.
Why wouldn't we use a fixed_size instead?

- Using a `fixed_size` ABI will create a scalar of size `N` in each CUDA thread!
- Loading a `fixed_size` variable from memory would result in uncoalesced access.
- If you have correct layouts you get `outer-loop` vectorization implicitly on GPUs.

But what if you really want to use **warp**-level parallelziation for SIMD types?

The GPU SIMD Problem

The above exercise used a **scalar** simd type on the **GPU**.
Why wouldn't we use a fixed_size instead?

- Using a `fixed_size` ABI will create a scalar of size `N` in each CUDA thread!
- Loading a `fixed_size` variable from memory would result in uncoalesced access.
- If you have correct layouts you get `outer-loop` vectorization implicitly on GPUs.

But what if you really want to use **warp**-level parallelziation for SIMD types?
We need *two* SIMD types: a *storage* type and a *temporary* type!

Important concept: simd::storage_type

Every `simd<T,ABI>` has an associated `storage_type` typedef.

To help with the GPU issue we split types between **storage** types used for `Views`, and **temporary** variables.

- Most `simd::simd` types will just have the same `storage_type`.
- `simd<T,cuda_warp<N>>` will use warp level parallelism.
- `simd<T,cuda_warp<N>>::storage_type` is different though!.
- Used in conjunction with `TeamPolicy`.

```
using simd_t = simd::simd<T,simd::simd_abi::cuda_warp<V> >;
using simd_storage_t = simd_t::storage_type;
View<simd_storage_t**> data("D",N,M); // will hold N*M*V Ts
parallel_for("Loop", TeamPolicy<>(N,M,V),
  KOKKOS_LAMBDA(const team_t& team) {
    int i = team.league_rank();
    parallel_for(TeamThreadRange(team,M), [&](int j) {
      data(i,j) = 2.0*simd_t(data(i,j));
}); });
```

Exercise #10: SIMD storage usage.

Details:

- Location: `Intro-Full/Exercises/10/Begin/`
- Include the `simd.hpp` header.
- Change the data type of the views to use `simd::simd<double,simd::simd_abi:cuda_warp< 32 >>::storage_type`.
- Create an unmanaged `View<double*>` of `results` using the `data()` function for the final reduction.
- Use inside of the lambda the `simd::simd<double,simd::simd_abi:cuda_warp< 32 >>` as scalar type.

```
# Compile for GPU
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./simd.cuda
```

Kokkos SIMD supports math operations:

- Common stuff like `abs`,`sqrt`,`exp`, ...

It also supports masking:

```
using simd_t = simd<double,simd_abi::native>;
using simd_mask_t = simd_t::mask_type;

simd_t threshold(100.0), a(a(i));
simd_mask_t is_smaller = threshold<a;
simd_t only_smaller = choose(is_smaller,a,threshold);
```

SIMD Summary

- SIMD types help vectorize code.
- In particular for **outer-loop** vectorization.
- There are **storage** and **temporary** types.
- Masking is supported too.
- Currently considered experimental at `https://github.com/Kokkos/simd-math`: please try it out and provide feedback.
- Will move into Kokkos proper likely in the next release.

Kokkos advanced capabilities NOT covered today

- Directed acyclic graph (DAG) of tasks pattern
  - Dynamic graph of heterogeneous tasks (maximum flexibility)
  - Static graph of homogeneous task (low overhead)
- Portable, thread scalable memory pool
- Plugging in customized multidimensional array data layout
  e.g., arbitrarily strided, hierarchical tiling

- For **portability**: OpenMP, OpenACC, ... or Kokkos.
- Only Kokkos obtains performant memory access patterns via **architecture-aware** arrays and work mapping.
  i.e., not just portable, *performance portable*.
- With Kokkos, **simple things stay simple** (parallel-for, etc.).
  i.e., it's *no more difficult* than OpenMP.
- **Advanced performance-optimizing patterns are simpler** with Kokkos than with native versions.
  i.e., you're *not missing out* on advanced features.
  - *full day tutorial only*