The Kokkos Lectures

The Fundamentals: A Condensed Short Tutorial

Daniel Arndt, Oak Ridge National Laboratory

ATPESC 2023

August 3, 2023

A Condensed Short Tutorial

This lecture covers fundamental concepts of Kokkos with Hands-On Exercises as homework.

Slides: `https://github.com/kokkos/kokkos-tutorials/blob/main/Intro-Short/KokkosTutorial_Short.pdf`

For the full lectures, with more capabilities covered, and more in-depth explanations visit:
`https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series`

The Kokkos Lectures

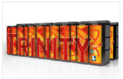
Watch the Kokkos Lectures for all of those and more in-depth explanations or do them on your own.

- Module 1: Introduction, Building and Parallel Dispatch
- Module 2: Views and Spaces
- Module 3: Data Structures + MultiDimensional Loops
- Module 4: Hierarchical Parallelism
- Module 5: Tasking, Streams and SIMD
- Module 6: Internode: MPI and PGAS
- Module 7: Tools: Profiling, Tuning and Debugging
- Module 8: Kernels: Sparse and Dense Linear Algebra

`https://kokkos.link/the-lectures`

The HPC Hardware Landscape

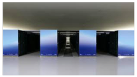
Current Generation: Programming Models OpenMP 3, CUDA and OpenACC depending on machine


LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3

LLNL SIERRA
IBM Power9 / NVIDIA Volta
CUDA / *OpenMP*[a]

ORNL Summit
IBM Power9 / NVIDIA Volta
CUDA / *OpenACC* / *OpenMP* [a]

SNL Astra
ARM CPUs
OpenMP 3


Riken Fugaku
ARM CPUs with SVE
OpenMP 3 / *OpenACC* [b]

Upcoming Generation: Programming Models OpenMP 5, CUDA, HIP and DPC++ depending on machine

NERSC Perlmutter
AMD CPU / NVIDIA GPU
CUDA / *OpenMP 5* [c]

ORNL Frontier
AMD CPU / AMD GPU
HIP / *OpenMP 5* [d]

ANL Aurora
Xeon CPUs / Intel GPUs
DPC++ / *OpenMP 5* [e]

LLNL El Capitan
AMD CPU / AMD GPU
HIP / *OpenMP 5* [d]

(a) Initially not working. Now more robust for Fortran than C++, but getting better.
(b) Research effort.
(c) OpenMP 5 by NVIDIA.
(d) OpenMP 5 by HPE.

(e) OpenMP 5 by Intel.

Industry Estimate

A full time software engineer writes 10 lines of production code per hour: 20k LOC/year.

- Typical HPC production app: 300k-600k lines
  - Sandia alone maintains a few dozen
- Large Scientific Libraries:
  - E3SM: 1,000k lines
  - Trilinos: 4,000k lines

Conservative estimate: need to rewrite 10% of an app to switch Programming Model

Industry Estimate

A full time software engineer writes 10 lines of production code per hour: 20k LOC/year.

- Typical HPC production app: 300k-600k lines
  - Sandia alone maintains a few dozen
- Large Scientific Libraries:
  - E3SM: 1,000k lines
  - Trilinos: 4,000k lines

Conservative estimate: need to rewrite 10% of an app to switch Programming Model

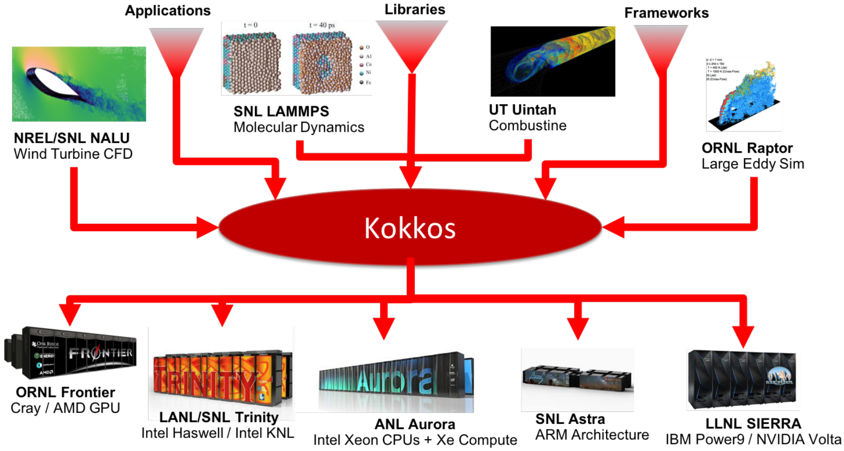
Software Cost Switching Vendors

Just switching Programming Models costs multiple person-years per app!

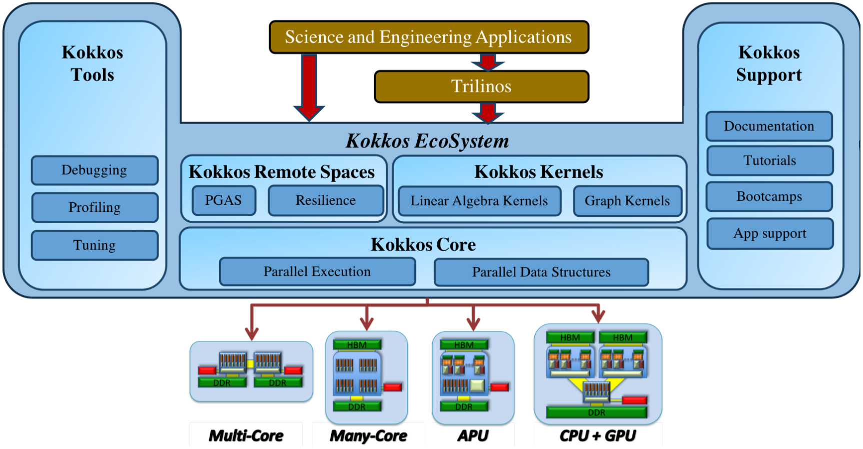
What is Kokkos?

- A C++ Programming Model for Performance Portability
  - Implemented as a template library on top CUDA, HIP, OpenMP, ...
  - Aims to be descriptive not prescriptive
  - Aligns with developments in the C++ standard
- Expanding solution for common needs of modern science and engineering codes
  - Math libraries based on Kokkos
  - Tools for debugging, profiling and tuning
  - Utilities for integration with Fortran and Python
- It is an Open Source project with a growing community
  - Maintained and developed at `https://github.com/kokkos`
  - Hundreds of users at many large institutions

Kokkos at the Center



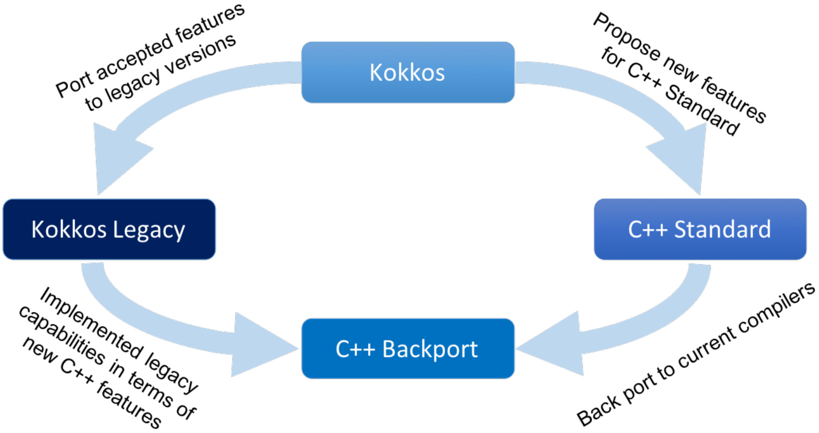
The Kokkos EcoSystem



The Kokkos Team

Kokkos Core:	**C. Trott**, **D. Lebrun-Grandié**, D. Arndt, J. Ciesko, C. Clevenger, N. Ellingwood, R. Gayatri, D. Ibanez, D. Lee, S. Lee, N. Liber, P. Miller, N. Morales, A. Powell, F. Rizzi, M. Simberg, C. Skrzyński, B. Turcksin
Kokkos Kernels:	**S. Rajamanickam**, **L. Berger-Vergiat**, V. Dang, N. Ellingwood, J. Foucar, E. Harvey, B. Kelley, K. Liegeois, C. Pearson, E. Prudencio
Kokkos Support	**C. Trott**, G. Shipmann, G. Womeldorff, and all of the above

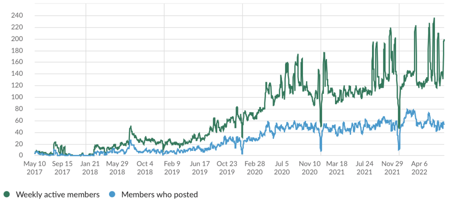
Kokkos helps improve ISO C++


Kokkos
Propose new features for C++ Standard
C++ Standard
Back port to current compilers
C++ Backport
Implemented legacy capabilities in terms of new C++ features
Kokkos Legacy
Port accepted features to legacy versions

Ten current or former Kokkos team members are members of the ISO C++ standard committee.

Kokkos has a growing OpenSource Community

- 20 ECP projects list Kokkos as Critical Dependency
  - 41 list C++ as critical
  - 25 list Lapack as critical
  - 21 list Fortran as critical
- Slack Channel: 900 members from 90+ institutions
  - 15% Sandia Nat. Lab.
  - 24% other US Labs
  - 22% universities
  - 39% other
- GitHub: 1.1k stars



Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series:
  - Slides, recording and Q&A for the Full Lectures
- https://kokkos.github.io/kokkos-core-wiki/:
  - Wiki including API reference
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

Data parallel patterns

Learning objectives:

- How computational bodies are passed to the Kokkos runtime.
- How work is mapped to execution resources.
- 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 execution resources

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 execution resources

- 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 execution resources

- 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, and Kokkos decides how to map that work to execution resources.

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 {...}
}
```

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 (6)

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 $\Rightarrow$ **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 */
});
```

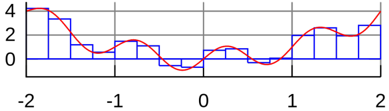
Important concept

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

Scalar integration (0)

Riemann-sum-style numerical integration:

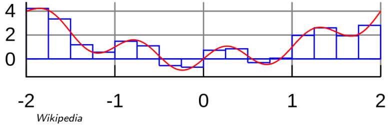
$$y = \int_{\textit{lower}}^{\textit{upper}} \textit{function}(x)\, dx$$



Wikipedia

Riemann-sum-style numerical integration:

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

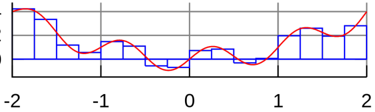


```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  const double x =
    lower + (i/numberOfIntervals) * (upper - lower);
  const double thisIntervalsContribution = function(x);
  totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

Scalar integration (0)

Riemann-sum-style numerical integration:

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



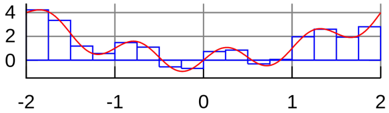
Wikipedia

```
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?*

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.

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 += ...
}
```

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.

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!

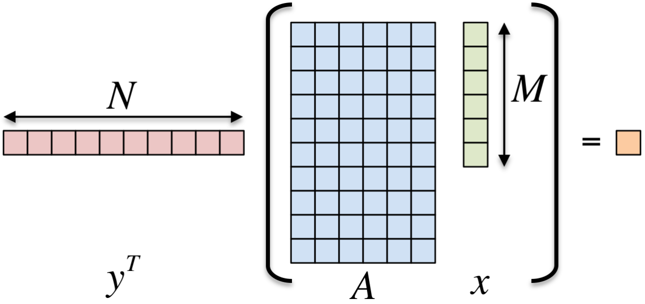
- Non-nested parallel patterns can take an optional string argument.
- The label doesn't need to be unique, but it is helpful.
- Anything convertible to "std::string"
- Used by profiling and debugging tools (see Profiling Tutorial)

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

Using your own ${HOME}

- Git
- GCC 8.2 (or newer) *OR* Intel 19.0.5 (or newer) *OR* Clang 8.0 (or newer)
- CUDA nvcc 11.0 (or newer) *AND* NVIDIA compute capability 6.0 (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/LectureSeries`

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

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

Exercise #1: include, initialize, finalize Kokkos

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

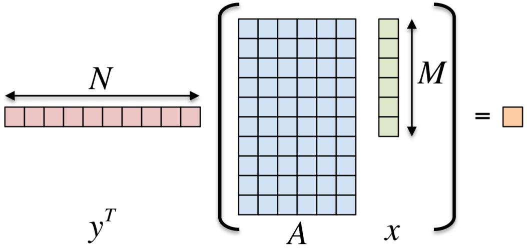
```
#include <Kokkos_Core.hpp>
int main(int argc, char* argv[]) {
  /* ... do any necessary setup (e.g., initialize MPI) ... */
  Kokkos::initialize(argc, argv);
  {
  /* ... do computations ... */
  }
  Kokkos::finalize();
  return 0;
}
```

(Optional) Command-line arguments or environment variables:

`--kokkos-num-threads=INT` or `KOKKOS_NUM_THREADS`	total number of threads
`--kokkos-device-id=INT` or `KOKKOS_DEVICE_ID`	device (GPU) ID to use

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

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



Details:

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

Exercise #1: logistics

Compiling for CPU

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

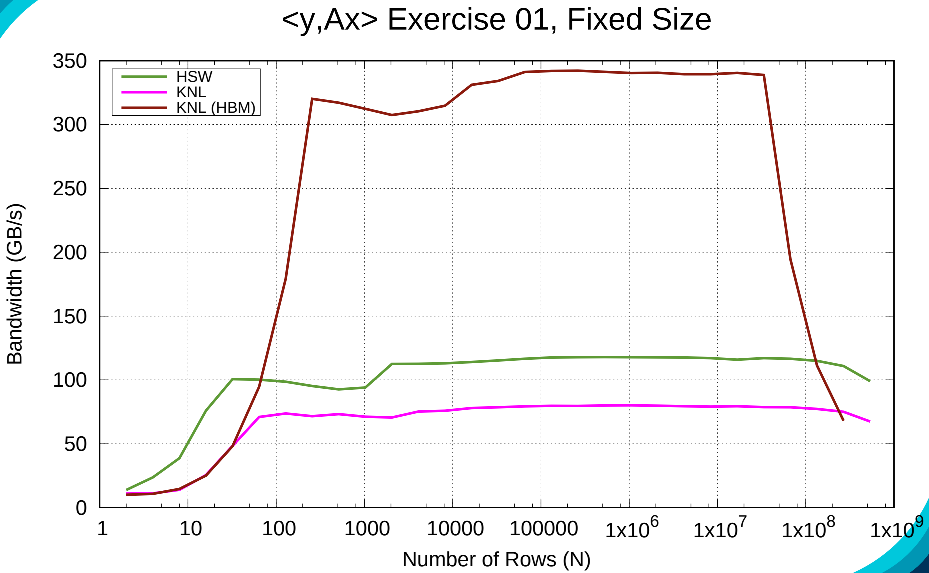
Running on CPU with OpenMP back-end

```
# 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



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.

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) const {
    _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) const {
    _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) const {
    _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 (shallow copy)
    y(i) = a * x(i) + y(i);
  });
```

Views (0)

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 (shallow copy)
    y(i) = a * x(i) + y(i);
  });
```

Important point

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

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.

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 `std::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 `std::shared_ptr`

Example:

```
View<double*[5]> a("a", N), b("b", K);
a = b;
View<double**> c(b);
a(0,2) = 1;
b(0,2) = 2;
c(0,2) = 3;
print_value( 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 `std::shared_ptr`

Example:

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

What gets printed?

```
3.0
```

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) == 5);
static_assert(a.static_extent(1) == 5);
assert(a.data() != nullptr);
assert(a.label() == "A");
```

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

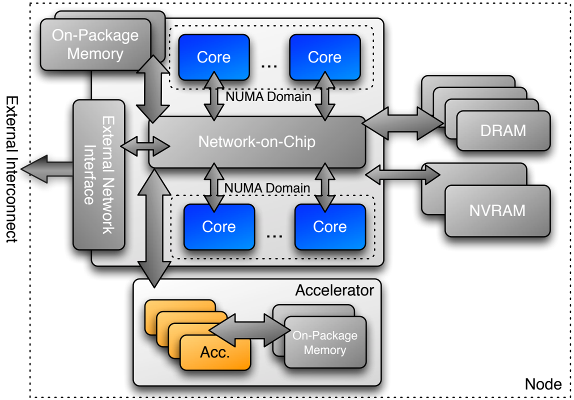
- Location: `Exercises/02/Begin/`
- Assignment: Change data storage from arrays to Views.
- Compile and run on CPU, and then on GPU with UVM

```
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

Execution Space

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



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

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

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?

  $\Rightarrow$ 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?
  $\Rightarrow$ Always in the **host process**
- Where will Parallel code be run? CPU? GPU?
  $\Rightarrow$ 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:

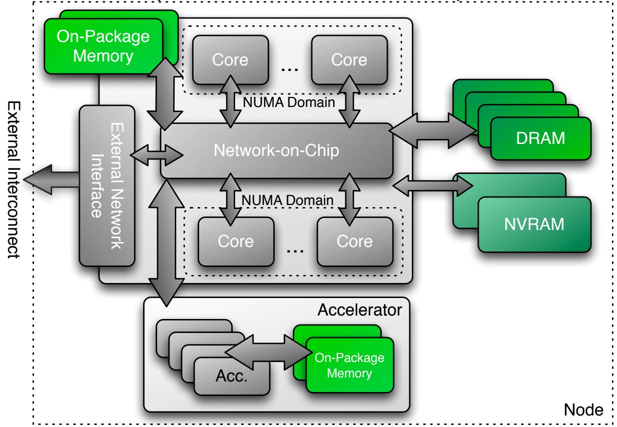
Function annotation with `KOKKOS_INLINE_FUNCTION` macro

```
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

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

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

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

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

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**.

Important concept: Memory spaces

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

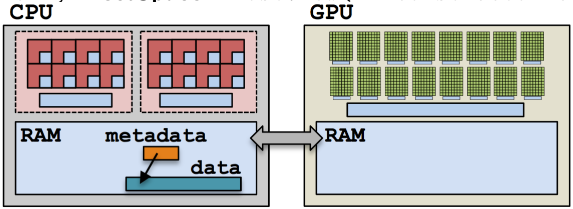
- `View<double***,`*Memory***Space**`> data(...);`
- Available **memory spaces**:

  `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**.

```
// Equivalent:
View<double*> a("A",N);
View<double*,DefaultExecutionSpace::memory_space> b("B",N);
```

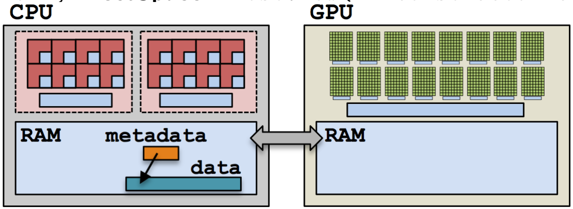
Example: HostSpace

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



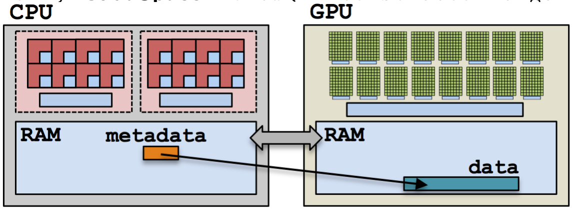
Example: HostSpace

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



Example: CudaSpace

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



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

Important concept: Mirrors

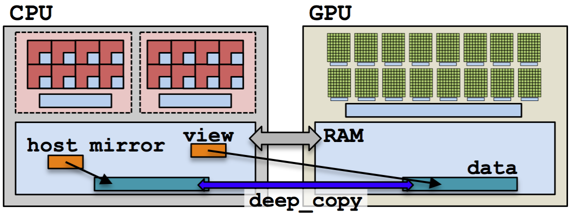
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

Important concept: Mirrors

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

Mirroring schematic

```
using view_type = Kokkos::View<double**, Space>;
view_type view(...);
view_type::HostMirror hostView =
  Kokkos::create_mirror_view(view);
```



Mirroring pattern

1. **Create** a `view`'s array in some memory space.

```
using view_type = Kokkos::View<double*, Space>;
view_type view(...);
```

Mirroring pattern

1. **Create** a `view`'s array in some memory space.

   ```
   using view_type = Kokkos::View<double*, Space>;
   view_type view(...);
   ```

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

   ```
   view_type::HostMirror hostView =
     Kokkos::create_mirror_view(view);
   ```

Mirroring pattern

1. **Create** a `view`'s array in some memory space.
   ```
   using view_type = Kokkos::View<double*, Space>;
   view_type view(...);
   ```
2. **Create** `hostView`, a *mirror* of the `view`'s array residing in the host memory space.
   ```
   view_type::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.
   ```
   using view_type = Kokkos::View<double*, Space>;
   view_type view(...);
   ```
2. **Create** `hostView`, a *mirror* of the `view`'s array residing in the host memory space.
   ```
   view_type::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);
   ```

Mirroring pattern

1. **Create** a view's array in some memory space.
   ```
   using view_type = Kokkos::View<double*, Space>;
   view_type view(...);
   ```
2. **Create** hostView, a *mirror* of the view's array residing in the host memory space.
   ```
   view_type::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.
```
using view_type = Kokkos::View<double*, Space>;
view_type view(...);
```
2. **Create** `hostView`, a *mirror* of the `view`'s array residing in the host memory space.
```
view_type::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?

```
using ViewType = Kokkos::View<double*, Space>;
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: `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.

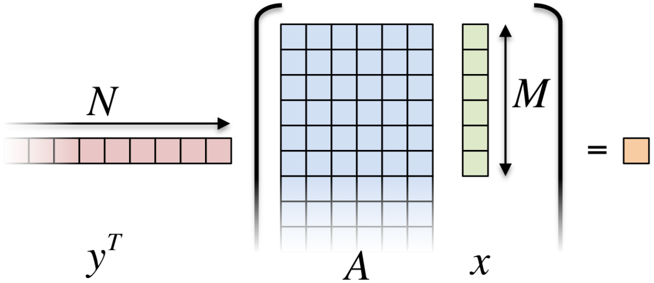
Managing memory access patterns for performance portability

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.

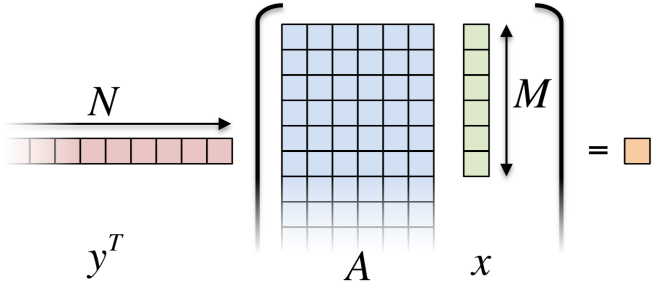
Example: inner product (0)

```
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);
```



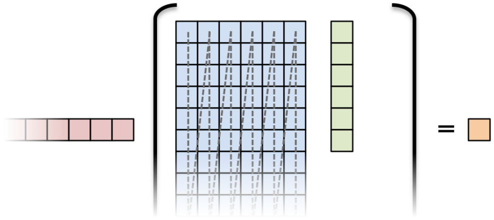
Driving question: How should A be laid out in memory?

Example: inner product (1)

Layout is the mapping of multi-index to memory:

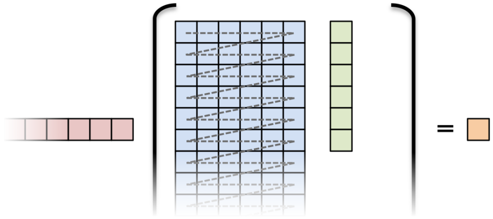
LayoutLeft

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

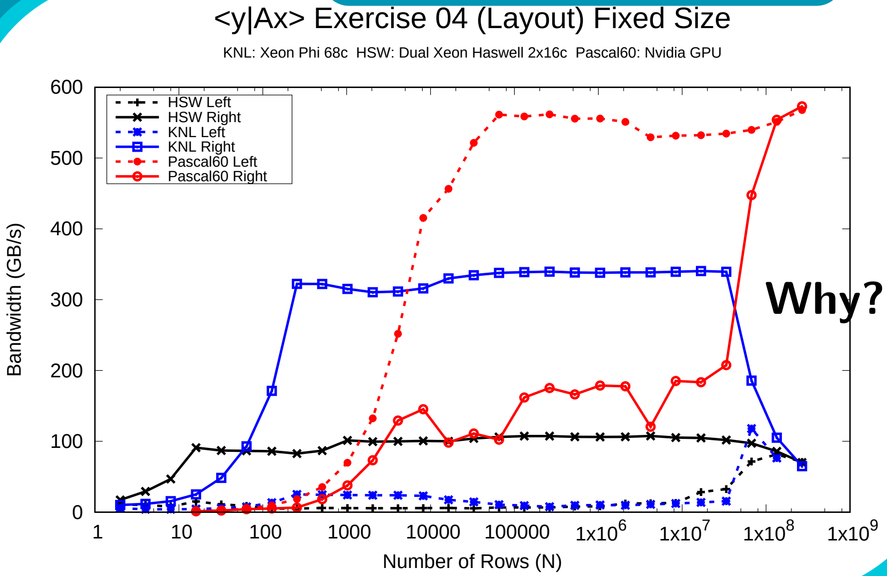
Details:

- Location: `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



Thread independence:

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

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

Thread independence:

```
operator()(int index, double & valueToUpdate) const {
  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()(int index, double & valueToUpdate) const {
  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 execute synchronized.
    - i.e., threads in groups can/must execute instructions together.

Thread independence:

```
operator()(int index, double & valueToUpdate) const {
  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 execute synchronized.
  - 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()(int index, double & valueToUpdate) const {
  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 execute synchronized.
    - 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 (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 (2)

Rule of Thumb

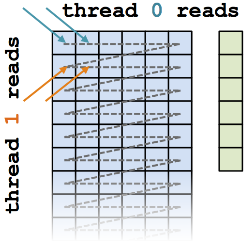
Kokkos index mapping and default layouts provide efficient access if **iteration indices** correspond to the **first index** of array.

Example:

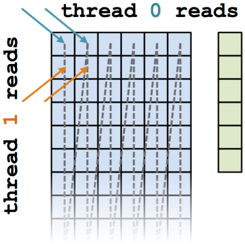
```
View<double***, ...> view(...);
...
Kokkos::parallel_for("Label", ... ,
  KOKKOS_LAMBDA (int workIndex) {
    ...
    view(..., ... , workIndex ) = ...;
    view(... , workIndex, ... ) = ...;
    view(workIndex, ... , ... ) = ...;
  });
...
```

Analysis: Kokkos architecture-dependent

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



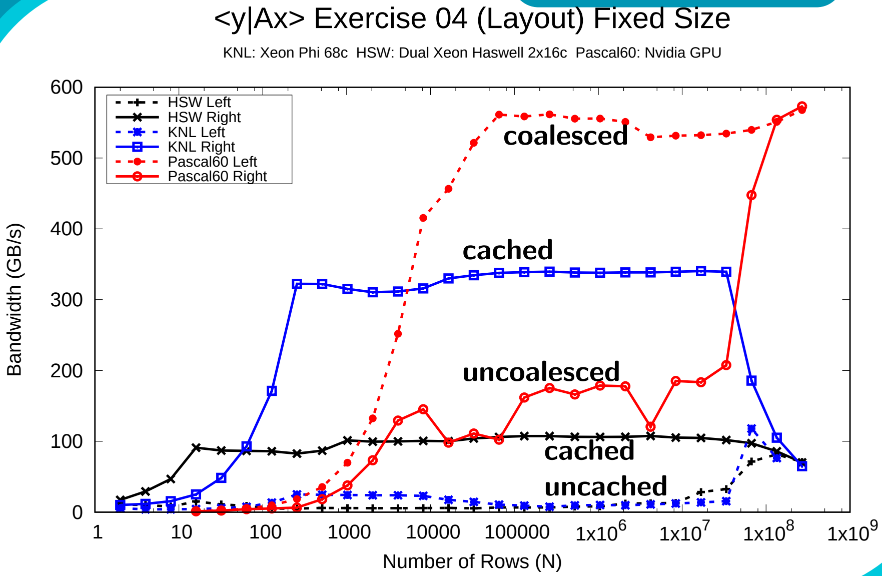
(a) OpenMP



(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 `SYCL` to manage layouts.
  $\Rightarrow$ You'll need multiple versions of code or pay the performance penalty.

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

MDRangePolicy (0)

Motivating example: Consider the nested for loops:

```
for ( int i = 0; i < N0; ++i )
for ( int j = 0; j < N1; ++j )
for ( int k = 0; k < N2; ++k )
  some_init_fcn(i, j, k);
```

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

```
Kokkos::parallel_for("Label", N0,
                     KOKKOS_LAMBDA (const i) {
                       for ( int j = 0; j < N1; ++j )
                       for ( int k = 0; k < N2; ++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?

OpenMP has a solution: the collapse clause

```
#pragma omp parallel for collapse(3)
for (int64_t i = 0; i < N0; ++i) {
  for (int64_t j = 0; j < N1; ++j) {
    for (int64_t k = 0; k < N2; ++k) {
      /* loop body */
    }
  }
}
```

OpenMP has a solution: the collapse clause

```
#pragma omp parallel for collapse(3)
for (int64_t i = 0; i < N0; ++i) {
  for (int64_t j = 0; j < N1; ++j) {
    for (int64_t k = 0; k < N2; ++k) {
      /* loop body */
    }
  }
}
```

Note this changed the policy by adding a ‘collapse‘ clause.

MDRangePolicy (1)

OpenMP has a solution: the collapse clause

```
#pragma omp parallel for collapse(3)
for (int64_t i = 0; i < N0; ++i) {
  for (int64_t j = 0; j < N1; ++j) {
    for (int64_t k = 0; k < N2; ++k) {
      /* loop body */
    }
  }
}
```

Note this changed the policy by adding a ‘collapse‘ clause.

With Kokkos you also change the policy:

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
     /* loop body */
});
```

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.

MDRangePolicy (2)

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.
- As with Kokkos Views: only rectangular iteration spaces.

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.
- As with Kokkos Views: only rectangular iteration spaces.
- Provide initializer lists for begin and end values.

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.
- As with Kokkos Views: only rectangular iteration spaces.
- Provide initializer lists for begin and end values.
- The functor/lambda takes matching number of indicies.

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

MDRangePolicy (3)

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.

MDRangePolicy (3)

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.

MDRangePolicy (3)

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.
- Can do other reductions with reducers.

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
      /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.
- Can do other reductions with reducers.
- Can use ‘View‘s as reduction argument.

MDRangePolicy (3)

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.
- Can do other reductions with reducers.
- Can use ‘View‘s as reduction argument.
- Multiple reducers not yet implemented though.

In structured grid applications a **tiling** strategy is often used to help with caching.

Tiling

MDRangePolicy uses a tiling strategy for the iteration space.

- Specified as a third initializer list.
- For GPUs a tile is handled by a single thread block.
  - If you provide too large a tile size this will fail!
- In Kokkos 3.3 we will add auto tuning for tile sizes.

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2},{T0,T1,T2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

MDRangePolicy (5)

Initializing a Matrix:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

MDRangePolicy (5)

Initializing a Matrix:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

How do I make sure that I get the right access pattern?

Iteration Pattern

MDRangePolicy provides compile time control over iteration patterns.

```
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 (7)

Initializing a Matrix fast:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Left,Iterate::Left>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Right,Iterate::Right>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

Initializing a Matrix fast:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Left,Iterate::Left>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Right,Iterate::Right>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

Default Patterns Match

Default iteration patterns match the default memory layouts!

Exercise - mdrange: Initialize multi-dim views with MDRangePolicy

Details:

- Location: `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
```

Common Policy Arguments

Template Parameters common to ALL policies.

- `ExecutionSpace`: control where code executes
  - **Options:** Serial, OpenMP, Threads, Cuda, HIP, ...
- `Schedule<Options>`: set scheduling policy.
  - **Options:** Static, Dynamic
- `IndexType<Options>`: control internal indexing type
  - **Options:** int, long, etc
- `WorkTag`: enables multiple operators in one functor

```
struct Foo {
  struct Tag1{}; struct Tag2{};
  KOKKOS_FUNCTION void operator(Tag1, int i) const {...}
  KOKKOS_FUNCTION void operator(Tag2, int i) const {...}
  void run_both(int N) {
    parallel_for(RangePolicy<Tag1>(0,N),*this);
    parallel_for(RangePolicy<Tag2>(0,N),*this);
  }
});
```

MDRangePolicy

- allows for tightly nested loops similar to OpenMP's collapse clause.
- requires functors/lambdas with as many parameters as its rank is.
- works with `parallel_for` and `parallel_reduce`.
- uses a tiling strategy for the iteration space.
- provides compile time control over iteration patterns.

Hierarchical parallelism

Finding and exploiting more parallelism in your computations.

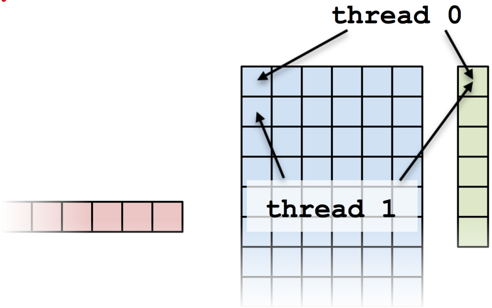
Learning objectives:

- 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);
```

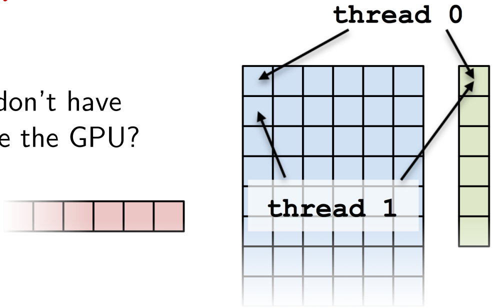


Example: inner product (0)

(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?



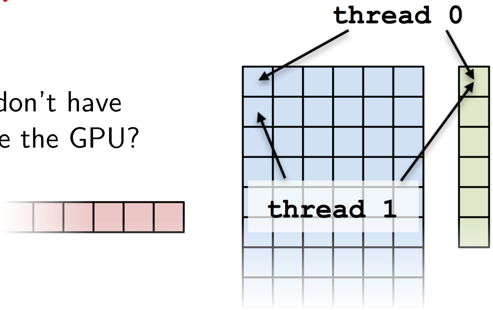
Example: inner product (0)

(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)

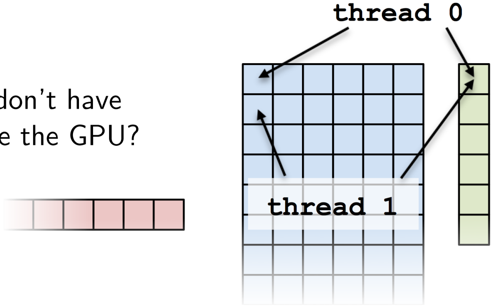
(Flat parallel) Kernel:

```
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?

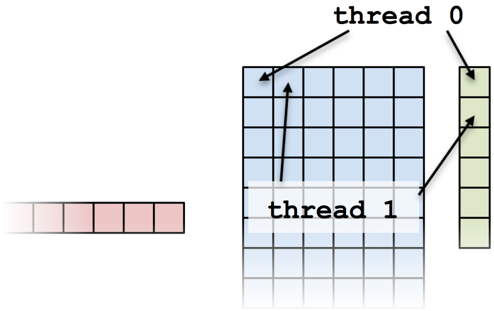
- Atomics
- Thread teams



Example: inner product (1)

Atomics kernel:

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

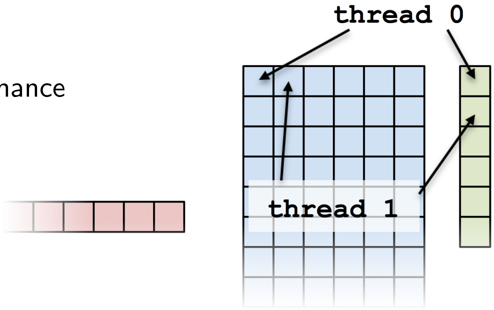


Example: inner product (1)

Atomics kernel:

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

Problem: Poor performance



Example: inner product (2)

Using an atomic with every element is doing scalar integration with atomics. (See module 3)

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)

Using an atomic with every element is doing scalar integration with atomics. (See module 3)

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**.

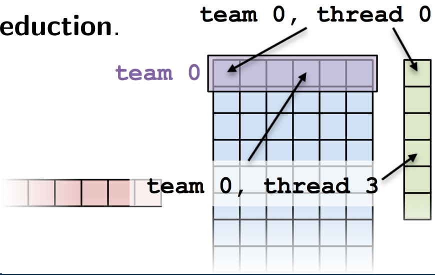
Example: inner product (3)

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.
2. Each team handles a 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);
```

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);
```

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

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

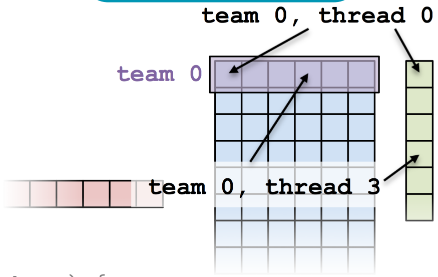
```
// total work = numberOfTeams * teamSize
parallel_for("Label",
  TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize), functor);
```

Important point

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

```
using member_type = typename TeamPolicy<ExecSpace>::member_type;

void operator()(const member_type & teamMember) {
  // How many teams are there?
  const unsigned int league_size = teamMember.league_size();

  // Which team am I on?
  const unsigned int league_rank = teamMember.league_rank();

  // How many threads are in the team?
  const unsigned int team_size = teamMember.team_size();

  // Which thread am I on this team?
  const unsigned int team_rank = teamMember.team_rank();

  // Make threads in a team wait on each other:
  teamMember.team_barrier();
}
```

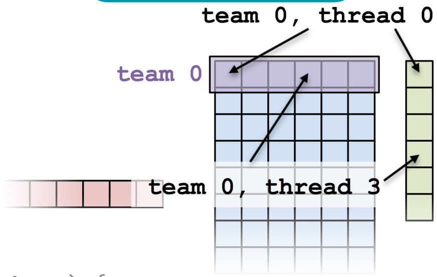
TeamThreadRange (0)



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?

Second attempt at exercise:

Divide row length among team members.

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

  int begin = teamMember.team_rank();
  for(int col = begin; col < M; col += teamMember.team_size()) {
    atomic_add(&result, y(row) * A(row,col) * x(entry));
  }
}
```

TeamThreadRange (1)

Second attempt at exercise:

Divide row length among team members.

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

  int begin = teamMember.team_rank();
  for(int col = begin; col < M; col += teamMember.team_size()) {
    atomic_add(&result, y(row) * A(row,col) * x(entry));
  }
}
```

- Still bad because `atomic_add` performs badly under high contention, how can team's member threads performantly cooperate for a nested reduction?
- On CPUs you get a bad data access pattern: this hardcodes coalesced access, but not caching.

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 (2)

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 (2)

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 (2)

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 (3)

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 (3)

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` and/or `reduce`.
- 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: TeamPolicy

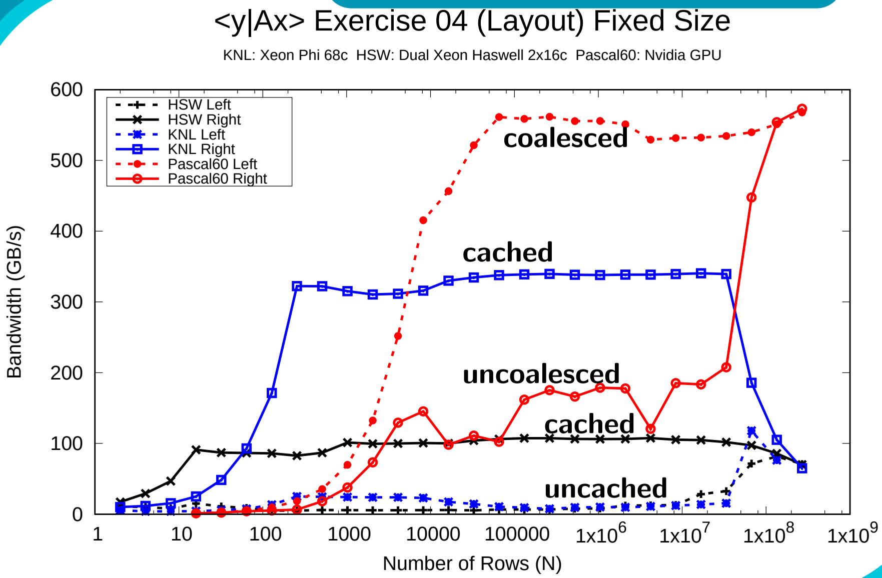
Details:

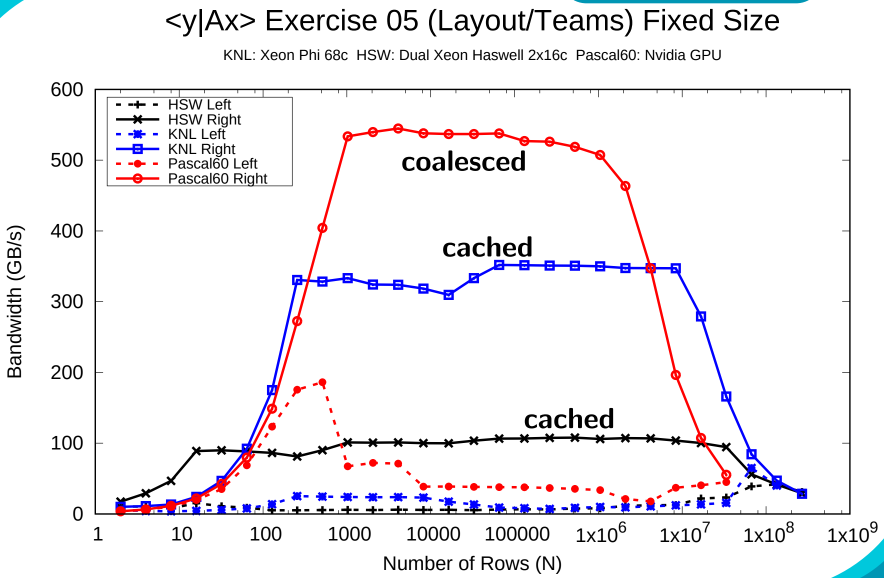
- Location: `Exercises/team_policy/`
- Replace `RangePolicy<Space>` with `TeamPolicy<Space>`
- Use `AUTO` for `team_size`
- Make the inner loop a `parallel_reduce` with `TeamThreadRange` policy
- Experiment with the combinations of `Layout`, `Space`, `N` to view performance
- Hint: what should the layout 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





<y|Ax> Exercise 05 (Layout/Teams) Fixed Size
KNL: Xeon Phi 68c HSW: Dual Xeon Haswell 2x16c Pascal60: Nvidia GPU
HSW Left
HSW Right
KNL Left
KNL Right
Pascal60 Left
Pascal60 Right
coalesced
cached
cached
Bandwidth (GB/s)
Number of Rows (N)
0
100
200
300
400
500
600
1
10
100
1000
10000
100000
1x10^6
1x10^7
1x10^8
1x10^9

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`

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);
```

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: TeamVectorLoop

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

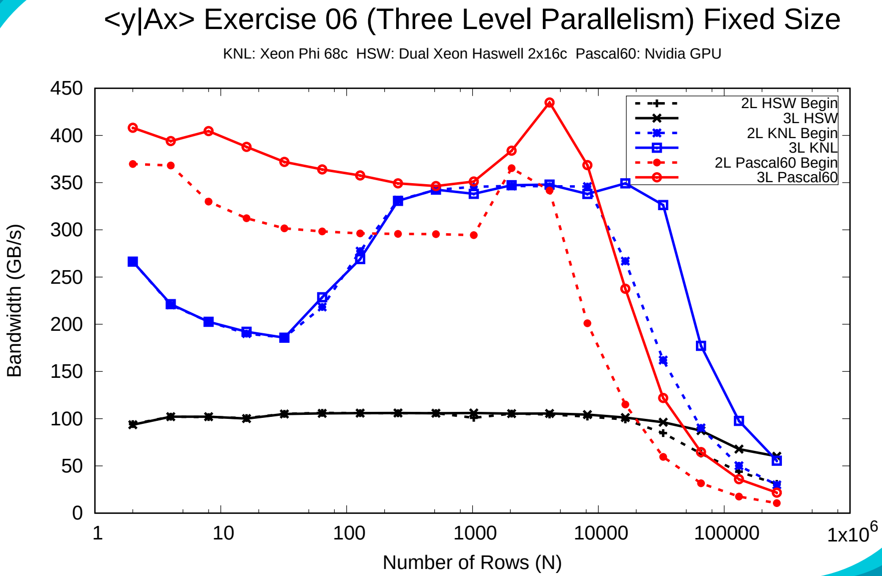
Details:

- Location: `Exercises/team_vector_loop/`
- 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 TeamPolicy Exercise for very non-square matrices
- Compare behavior of CPU vs GPU

Exercise: TeamVectorLoop



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`, `ThreadVectorRange`, and `TeamVectorRange` policy.
- Execution can be restricted to a subset of the team with the `single` pattern using either a `PerTeam` or `PerThread` policy.

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.

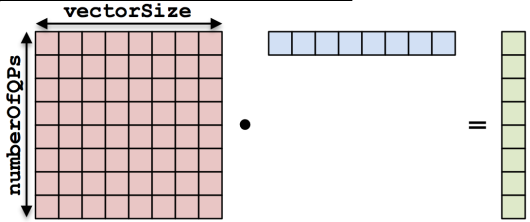
Manually Managed Cache

- 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)

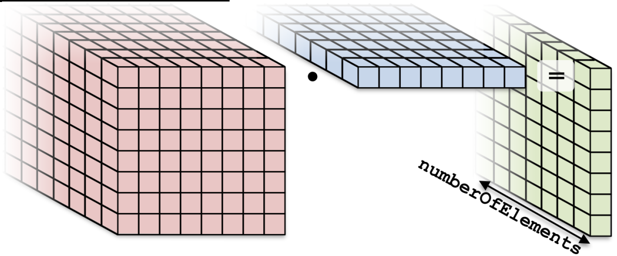
One slice of contractDataFieldScalar:



```
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)

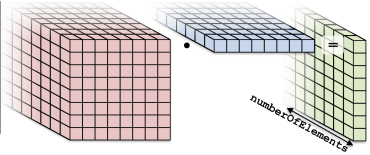
contractDataFieldScalar:



```
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;
  }
}
```



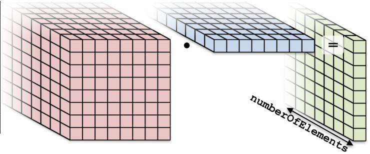
Parallelization approaches:

- 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;
  }
}
```

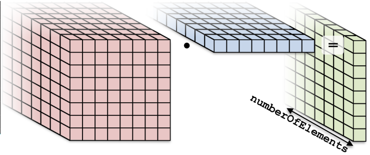


Parallelization approaches:

- 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;
  }
}
```

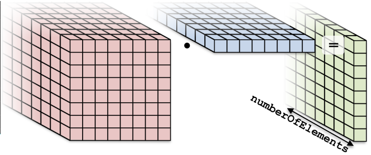


Parallelization approaches:

- 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;
  }
}
```

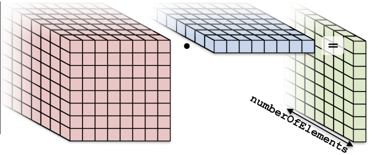


Parallelization approaches:

- 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;
  }
}
```

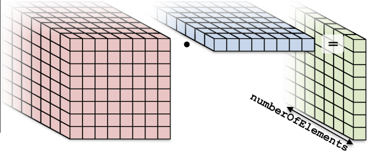


Flat kernel: Each thread handles a quadrature point

```
parallel_for("L",MDRangePolicy<Rank<2>>({0,0},{numE,numQP}),
  KOKKOS_LAMBDA(int element, 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;
  }
}
```

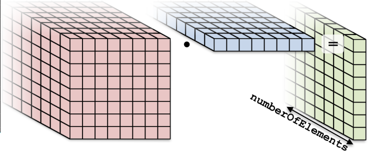


Teams kernel: Each team handles an element

```
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;
    });
}
```

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;
  }
}
```



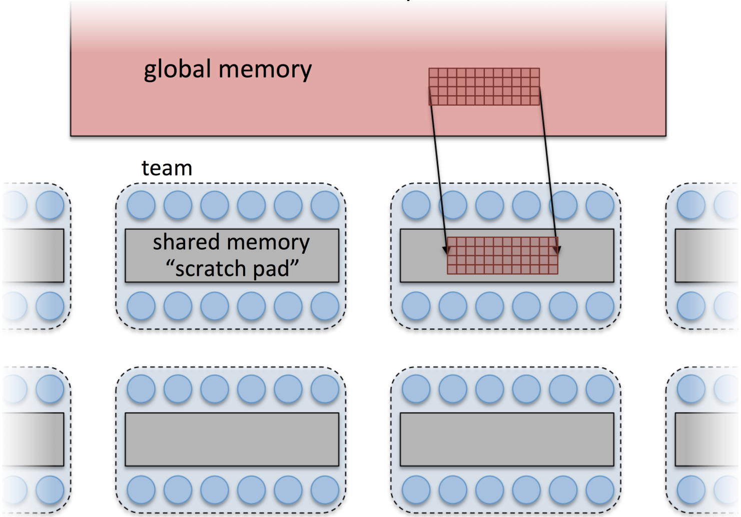
Teams kernel: Each team handles an element

```
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:

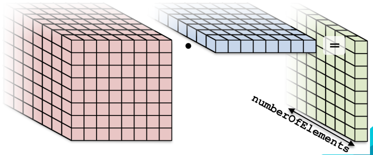
1. **Tell Kokkos how much** scratch memory you'll need.
2. **Make** scratch memory **views** inside your kernels.

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

// Define a scratch memory view type
using ScratchPadView =
    View<double*,ExecutionSpace::scratch_memory_space>;
// 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);
});
```

Example: contractDataFieldScalar (7)

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) * B(element, i);
        total += A(element, qp, i) * scratch(i);
      }
      result(element, qp) = total;
    });
}
```

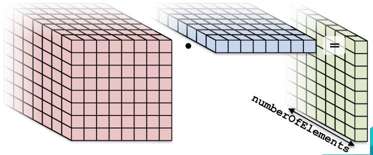


Example: contractDataFieldScalar (8)

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);
  }
}
```



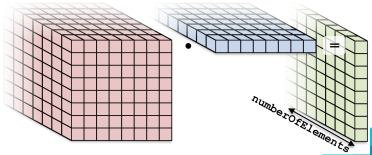
How to populate the scratch memory?

- ~~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);
```



How to populate the scratch memory?

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

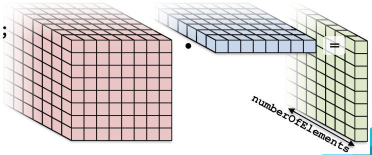
```
if (teamMember.team_rank() == 0) {
  for (int i = 0; i < vectorSize; ++i) {
    scratch(i) = B(element, i);
  }
}
```

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

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

- TeamVectorRange

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



How to populate the scratch memory?

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

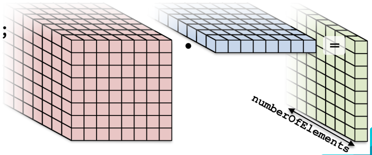
```
if (teamMember.team_rank() == 0) {
  for (int i = 0; i < vectorSize; ++i) {
    scratch(i) = B(element, i);
  }
}
```

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

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

- TeamVectorRange

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



(incomplete) Kernel for teams with scratch memory:

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

  parallel_for(TeamVectorRange(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:

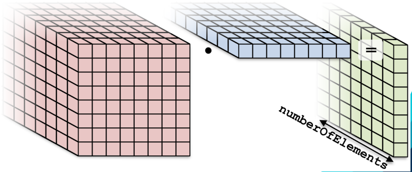
```
operator()(member_type teamMember) {
  ScratchPadView scratch(...);

  parallel_for(TeamVectorRange(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: Scratch Memory

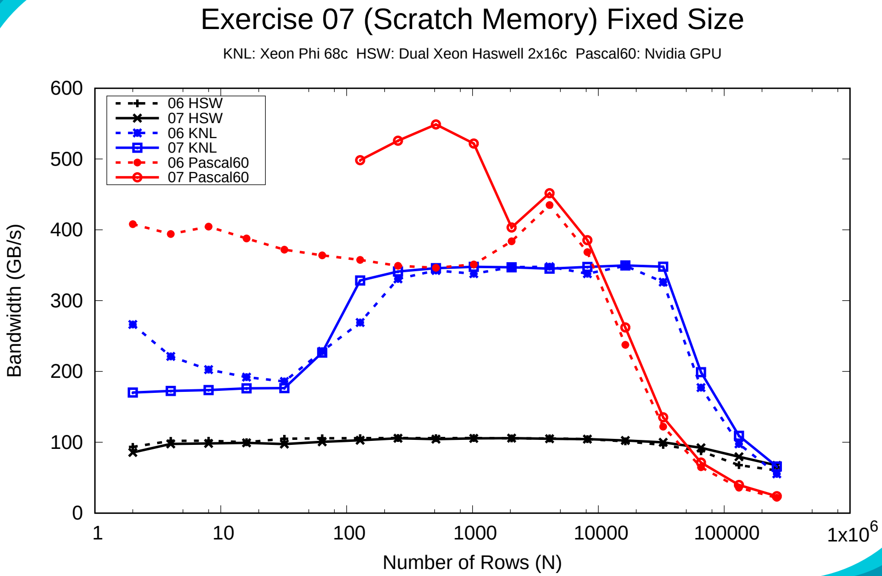
Use Scratch Memory to explicitly cache the x-vector for each element.

Details:

- Location: `Exercises/team_scratch_memory/`
- Create a scratch view
- Fill the scratch view in parallel using a TeamVectorRange

Things to try:

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


Exercise 07 (Scratch Memory) Fixed Size
KNL: Xeon Phi 68c HSW: Dual Xeon Haswell 2x16c Pascal60: Nvidia GPU
Bandwidth (GB/s)
Number of Rows (N)
06 HSW
07 HSW
06 KNL
07 KNL
06 Pascal60
07 Pascal60
0
100
200
300
400
500
600
1
10
100
1000
10000
100000
1x10^6

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.

What we didn't cover

This was a short introduction
Didn't cover many things:

This was a short introduction

Didn’t cover many things:

- Full BuildSystem integration.

What we didn't cover

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.

What we didn't cover

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Advanced data structures.

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Advanced data structures.
- Subviews.

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Advanced data structures.
- Subviews.
- Atomic operations and Scatter Contribute patterns.

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Advanced data structures.
- Subviews.
- Atomic operations and Scatter Contribute patterns.
- SIMD vectorization.

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Advanced data structures.
- Subviews.
- Atomic operations and Scatter Contribute patterns.
- SIMD vectorization.
- MPI and PGAS integration.

This was a short introduction

Didn’t cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Advanced data structures.
- Subviews.
- Atomic operations and Scatter Contribute patterns.
- SIMD vectorization.
- MPI and PGAS integration.
- Tools for Profiling, Debugging and Tuning.

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Advanced data structures.
- Subviews.
- Atomic operations and Scatter Contribute patterns.
- SIMD vectorization.
- MPI and PGAS integration.
- Tools for Profiling, Debugging and Tuning.
- Math Kernels (KokkosKernels).

The Kokkos Lectures

Watch the Kokkos Lectures for all of those and more in-depth explanations or do them on your own.

- Module 1: Introduction, Building and Parallel Dispatch
- Module 2: Views and Spaces
- Module 3: Data Structures + MultiDimensional Loops
- Module 4: Hierarchical Parallelism
- Module 5: Tasking, Streams and SIMD
- Module 6: Internode: MPI and PGAS
- Module 7: Tools: Profiling, Tuning and Debugging
- Module 8: Kernels: Sparse and Dense Linear Algebra

`https://kokkos.link/the-lectures`

Find More

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://kokkos.link/the-lectures:
  - Slides, recording and Q&A for the Full Lectures
- https://github.com/kokkos/kokkos/wiki:
  - Wiki including API reference
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

Acknowledgments

This manuscript has been authored by UT-Battelle, LLC, under Contract No. DE-AC0500OR22725 with the U.S. Department of Energy.