

The Kokkos Lectures

The Compact Course

June 15, 2021

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2020-7475 PE

A 1-Day Tutorial

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

Slides: https://github.com/kokkos/kokkos-tutorials/Intro-Medium/KokkosTutorial_Medium.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>

Instructions to get into the AWS cloud instances can be found at:

<https://github.com/kokkos/kokkos-tutorials/issues>

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



LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3



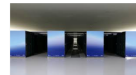
LLNL SIERRA
IBM Power9 / NVIDIA Volta
CUDA / OpenMP^(a)



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.

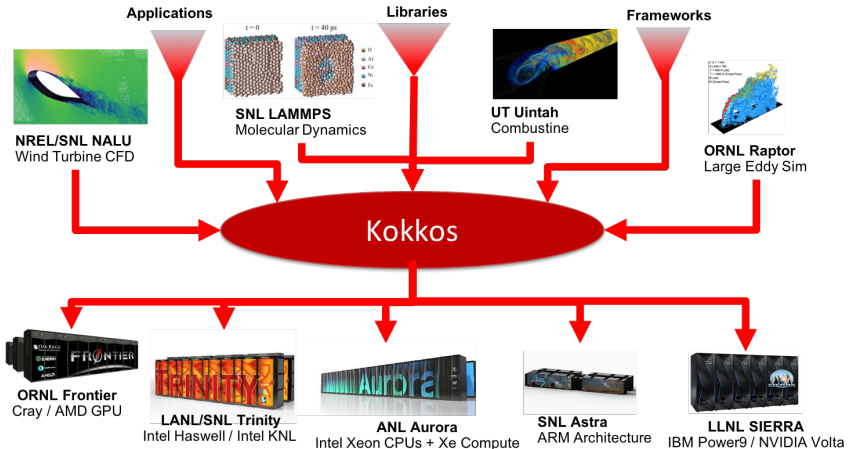
- ▶ Typical HPC production app: 300k-600k lines
 - ▶ Sandia alone maintains a few dozen
- ▶ Large Scientific Libraries:
 - ▶ E3SM: 1,000k lines
 - ▶ Trilinos: 4,000k lines

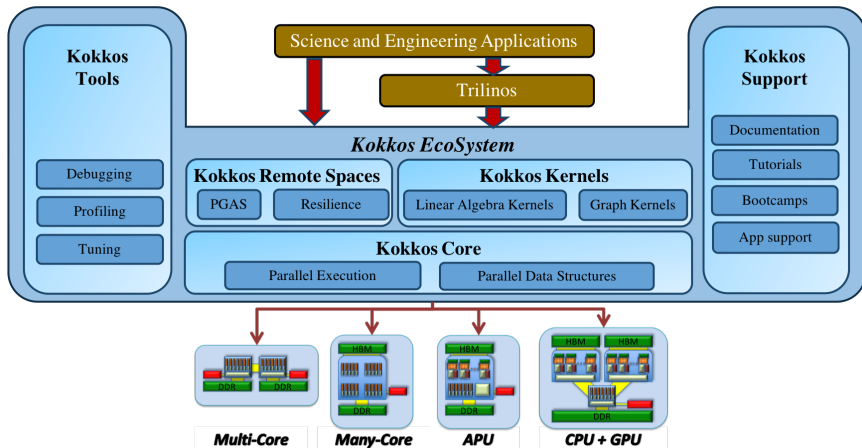
Conservative estimate: need to rewrite 10% of an app to switch Programming Model

Software Cost Switching Vendors

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

- ▶ 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
- ▶ Is is an Open Source project with a growing community
 - ▶ Maintained and developed at <https://github.com/kokkos>
 - ▶ Hundreds of users at many large institutions

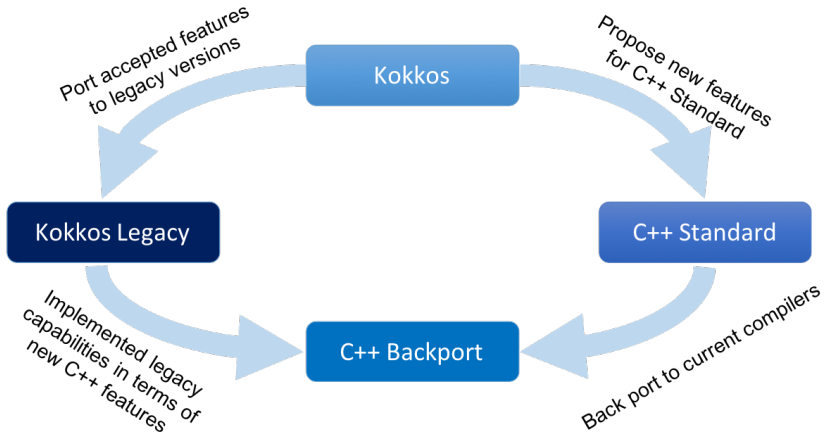






Kokkos Core:	C.R.Trott , J. Ciesko, V. Dang, N. Ellingwood, D. Ibanez, , H. Finkel, N. Liber, D. Lebrun-Grandie, D. Arndt, B. Turcksin, J. Madsen, R. Gayatri former: H.C. Edwards, D. Labreche, G. Mackey, S. Bova, D. Sunder- land, D.S. Hollman, J. Miles, J. Wilke
Kokkos Kernels:	S. Rajamanickam , L. Berger, V. Dang, N. Ellingwood, E. Harvey, B. Kelley, K. Kim, A. Powell, C.R. Trott former: J. Wilke, S. Acer
Kokkos Tools	D. Poliakoff , C. Lewis, S. Hammond, D. Ibanez, J. Madsen, S. Moore, C.R. Trott
Kokkos Support	C.R. Trott , G. Shipmann, G. Womeldorff, and all of the above former: H.C. Edwards, G. Lopez, F. Foertter

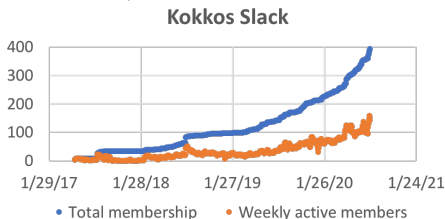
Kokkos helps improve ISO C++



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

Kokkos has a growing OpenSource Community

- ▶ 18 ECP projects list Kokkos as Critical Dependency
 - ▶ 41 list C++ as critical
 - ▶ 24 list Lapack as critical
 - ▶ 21 list Fortran as critical
- ▶ Slack Channel: 600 members from 75+ institutions
 - ▶ 20% Sandia Nat. Lab.
 - ▶ 35% other US Labs
 - ▶ 20% universities
 - ▶ 25% other
- ▶ GitHub: 800 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://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.

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.

Data parallel patterns and work

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

Kokkos maps **work** to execution resources

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.

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.

How are computational bodies given to Kokkos?

How are computational bodies given to Kokkos?

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

How are computational bodies given to Kokkos?

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

Quick review, a **functor** is a function with data. Example:

```
struct ParallelFunctor {  
    ...  
    void operator()( a work assignment ) const {  
        /* ... computational body ... */  
        ...  
    };  
};
```

How is work assigned to functor operators?

How is work assigned to functor operators?

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

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

How is work assigned to functor operators?

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

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

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

```
struct Functor {  
    void operator()(const 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.

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);
```


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);  
    }  
);
```

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.

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

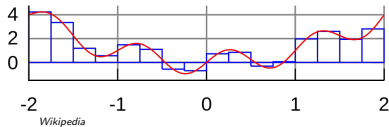
```
parallel_for(N, [=] (const int64_t i) {  
    /* loop body */  
});
```

Important concept

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

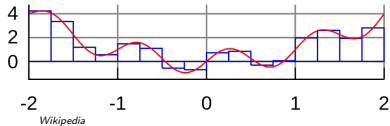
Riemann-sum-style numerical integration:

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



Riemann-sum-style numerical integration:

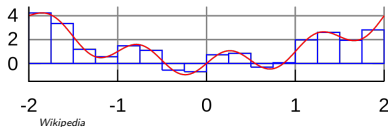
$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



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

Riemann-sum-style numerical integration:

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

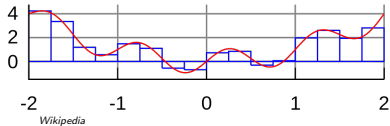


```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

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

Riemann-sum-style numerical integration:

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



Pattern?

```
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;
```

Policy?

Body?

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

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

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

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

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

```
double finalReducedValue = 0;
#pragma omp parallel for reduction(+:finalReducedValue)
for (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!

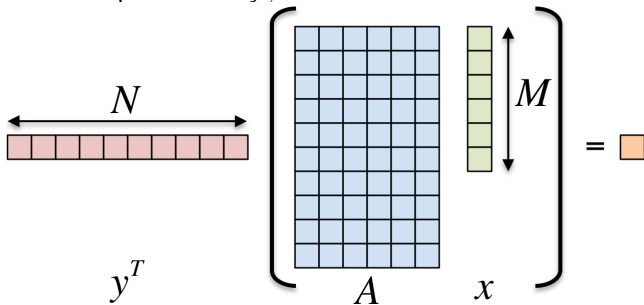
Giving unique names to each kernel is immensely helpful for debugging and profiling. You will regret it if you don't!

- ▶ 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);
```

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



Details:

- ▶ y is $N \times 1$, A is $N \times M$, x is $M \times 1$
- ▶ We'll use this exercise throughout the tutorial

Exercise #1: include, initialize, finalize Kokkos

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

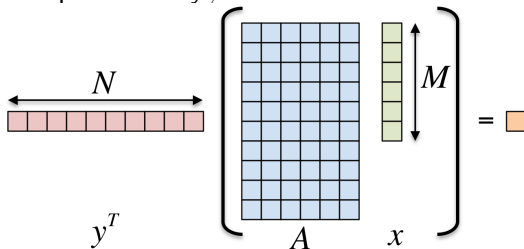
```
#include <Kokkos_Core.hpp>
int main(int argc, char* argv[]) {
    /* ... do any necessary setup (e.g., initialize MPI) ... */
    Kokkos::initialize(argc, argv);
    {
        /* ... do computations ... */
    }
    Kokkos::finalize();
    return 0;
}
```

(Optional) Command-line arguments or environment variables:

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

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

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



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.

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=...
```

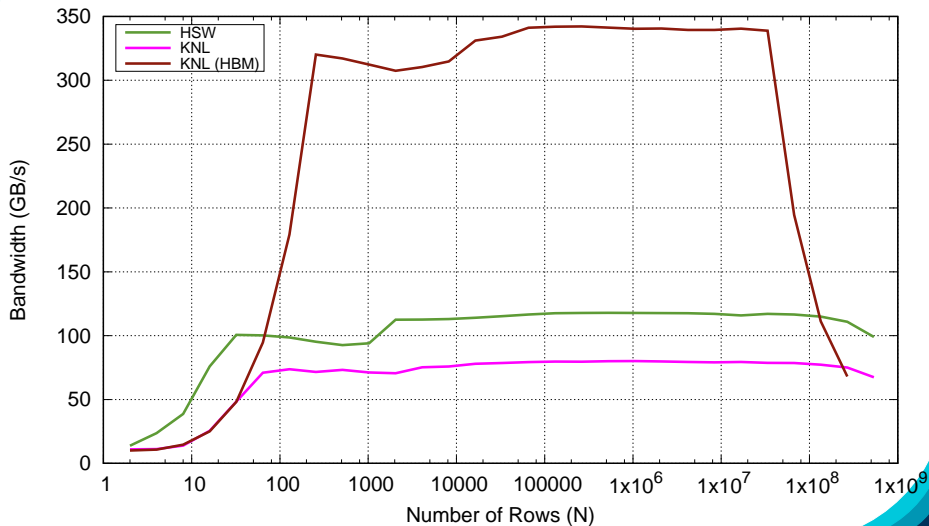
Running on CPU with OpenMP back-end

```
# 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 = $2^s == 2^{n+m}$

<y,Ax> Exercise 01, Fixed Size



- ▶ **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];
    }
};
```

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).

⇒ **Views**

View abstraction

- ▶ A *lightweight* C++ class with a pointer to array data and a little meta-data,
- ▶ that is *templated* on the data type (and other things).

High-level example of Views for daxpy using lambda:

```
View<double*, ...> x(...), y(...);  
...populate x, y...  
  
parallel_for("DAXPY",N, [=] (const int64_t i) {  
    // Views x and y are captured by value (copy)  
    y(i) = a * x(i) + y(i);  
});
```

View abstraction

- ▶ A *lightweight* C++ class with a pointer to array data and a little meta-data,
- ▶ that is *templated* on the data type (and other things).

High-level example of Views for daxpy using lambda:

```
View<double*, ...> x(...), y(...);  
...populate x, y...  
  
parallel_for("DAXPY",N, [=] (const int64_t i) {  
    // Views x and y are captured by value (copy)  
    y(i) = a * x(i) + y(i);  
});
```

Important point

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

View overview:

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

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.

View life cycle:

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

View life cycle:

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

Example:

```
View<double*[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 `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

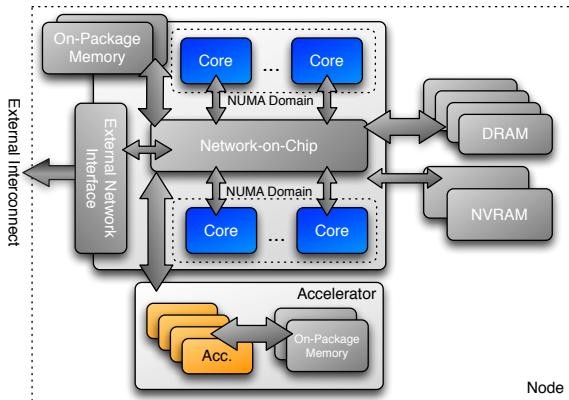
Execution and Memory Spaces

Learning objectives:

- ▶ Heterogeneous nodes and the **space** abstractions.
- ▶ How to control where parallel bodies are run, **execution space**.
- ▶ How to control where view data resides, **memory space**.
- ▶ How to avoid illegal memory accesses and manage data movement.
- ▶ The need for `Kokkos::initialize` and `finalize`.
- ▶ Where to use Kokkos annotation macros for portability.

Execution Space

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



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

Host

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

Parallel

```
Kokkos::parallel_for("MyKernel", numberOfSomethings,  
                    [=] (const int64_t somethingIndex) {  
                        const double y = ...;  
                        // do something interesting  
                    })  
);
```

Host

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

Parallel

```
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**

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

- ▶ Where will **Host** code be run? CPU? GPU?
⇒ Always in the **host process**
- ▶ Where will **Parallel** code be run? CPU? GPU?
⇒ The **default execution space**

Host

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

Parallel

```
Kokkos::parallel_for("MyKernel", numberOfSomethings,
                    [=] (const int64_t somethingIndex) {
                        const double y = ...;
                        // do something interesting
                    }
                    );
```

- ▶ Where will **Host** code be run? CPU? GPU?
⇒ Always in the **host process**
- ▶ Where will **Parallel** code be run? CPU? GPU?
⇒ The **default execution space**
- ▶ How do I **control** where the **Parallel** body is executed?
Changing the default execution space (*at compilation*),
or specifying an execution space in the **policy**.

Changing the parallel execution space:

Custom

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

Default

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

Changing the parallel execution space:

Custom

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

Default

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

Requirements for enabling execution spaces:

- ▶ Kokkos must be **compiled** with the execution spaces enabled.
- ▶ Execution spaces must be **initialized** (and **finalized**).
- ▶ **Functions** must be marked with a **macro** for non-CPU spaces.
- ▶ **Lambdas** must be marked with a **macro** for non-CPU spaces.

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```
struct ParallelFunctor {  
  KOKKOS_INLINE_FUNCTION  
  double helperFunction(const int64_t s) const {...}  
  KOKKOS_INLINE_FUNCTION  
  void operator()(const int64_t index) const {  
    helperFunction(index);  
  }  
}  
// Where kokkos defines:  
#define KOKKOS_INLINE_FUNCTION inline /* #if CPU-only */  
#define KOKKOS_INLINE_FUNCTION inline __device__ __host__ /* #if CPU+Cuda */
```

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```
struct ParallelFunctor {
  KOKKOS_INLINE_FUNCTION
  double helperFunction(const int64_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const int64_t index) const {
    helperFunction(index);
  }
}

// Where kokkos defines:
#define KOKKOS_INLINE_FUNCTION inline /* #if CPU-only */
#define KOKKOS_INLINE_FUNCTION inline __device__ __host__ /* #if CPU+Cuda */
```

Lambda annotation with KOKKOS_LAMBDA macro

```
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 motivating example: summing an array

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

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

Memory space motivating example: summing an array

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

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

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

Memory space motivating example: summing an array

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

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

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

Memory space motivating example: summing an array

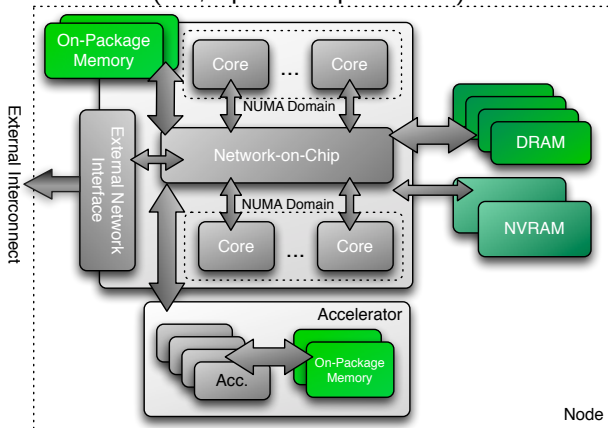
```
View<double*> data("data", size);
for (int64_t i = 0; i < size; ++i) {
    data(i) = ...read from file...
}

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

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

⇒ **Memory Spaces**

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



Important concept: Memory spaces

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

Important concept: Memory spaces

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

▶ `View<double***, MemorySpace> data(...);`

Important concept: Memory spaces

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

- ▶ `View<double***, MemorySpace> data(...);`
- ▶ Available **memory spaces**:
 `HostSpace, CudaSpace, CudaUVMSpace, ... more`

Important concept: Memory spaces

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

- ▶ `View<double***, MemorySpace> data(...);`
- ▶ Available **memory spaces**:
 `HostSpace`, `CudaSpace`, `CudaUVMSpace`, ... more
- ▶ Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space

Important concept: Memory spaces

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

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

Important concept: Memory spaces

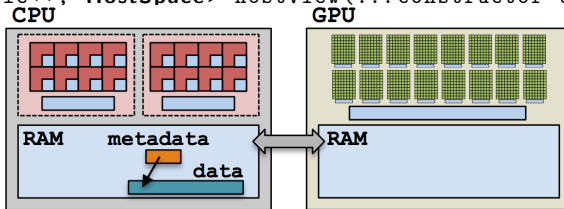
Every view stores its data in a **memory space** set at compile time.

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

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

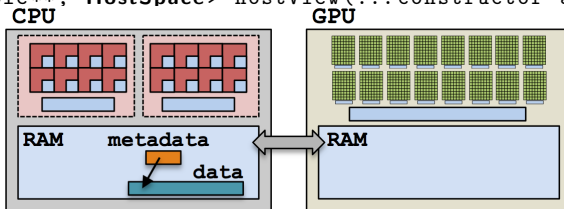
Example: HostSpace

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



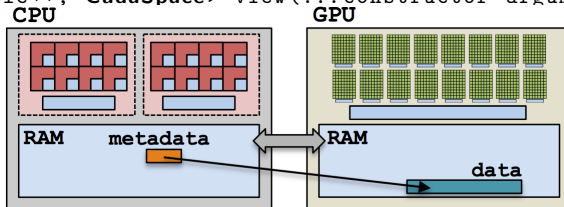
Example: HostSpace

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



Example: CudaSpace

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



Anatomy of a kernel launch:

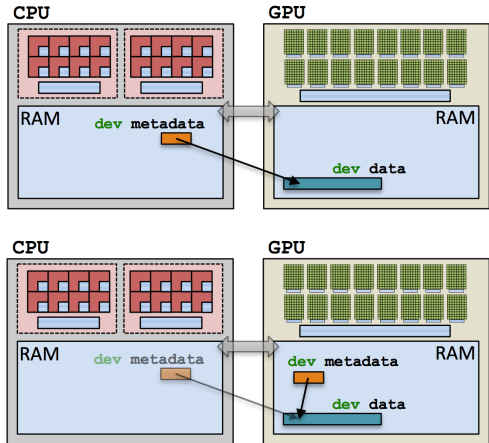
1. User declares views, allocating.
2. User instantiates a functor with views.
3. User launches `parallel_something`:
 - ▶ Functor is copied to the device.
 - ▶ Kernel is run.
 - ▶ Copy of functor on the device is released.

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

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

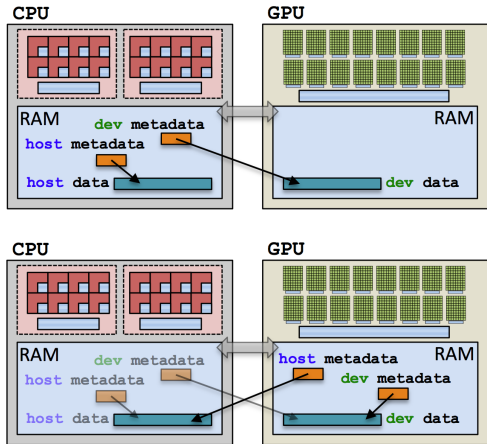
Example: one view

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



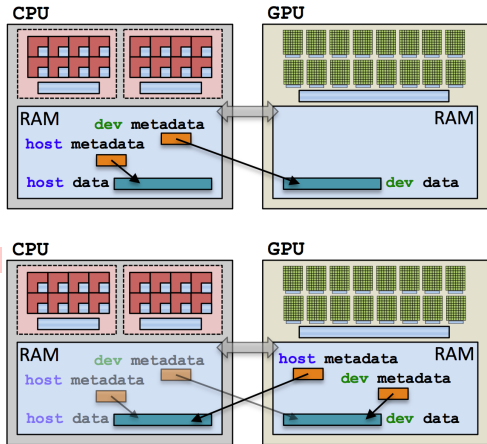
Example: two views

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



Example: two views

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



Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

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

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


Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

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

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

fault

Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

```
View<double*, HostSpace> array("array", size);
for (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...
}

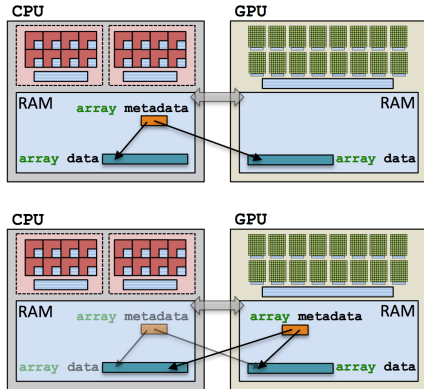
double sum = 0;
Kokkos::parallel_reduce("Label",
    RangePolicy< Cuda>(0, size),
    KOKKOS_LAMBDA (const int64_t index, double & valueToUpdate) {
        valueToUpdate += array(index);          illegal access
    },
    sum);
```

What's the solution?

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

CudaUVMSpace

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



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

Important concept: Mirrors

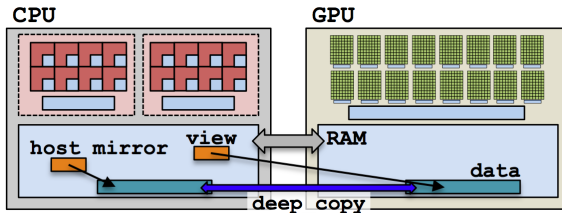
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

Important concept: Mirrors

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

Mirroring schematic

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



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

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


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);
```

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.).

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);
```

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 });
```

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);
```

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

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

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

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

Details:

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

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

Things to try:

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

- ▶ Data is stored in Views that are “pointers” to **multi-dimensional arrays** residing in **memory spaces**.
- ▶ Views **abstract away** platform-dependent allocation, (automatic) deallocation, and access.
- ▶ **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

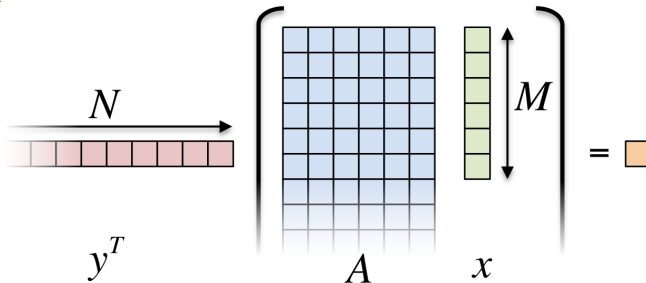
Learning objectives:

- ▶ 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.

```

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);

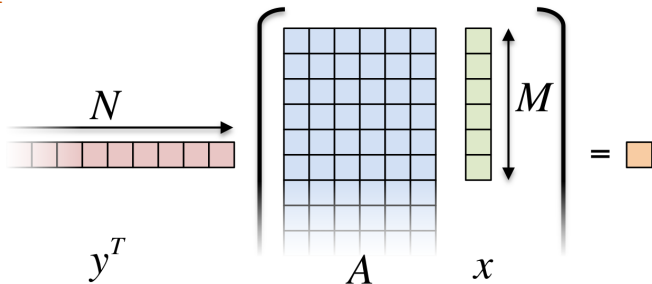
```



```

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+).

```

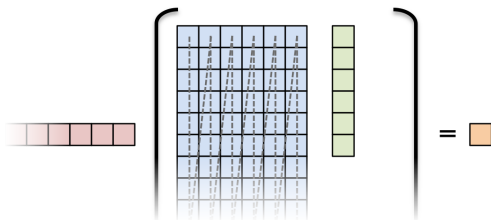


Driving question: How should A be laid out in memory?

Layout is the mapping of multi-index to memory:

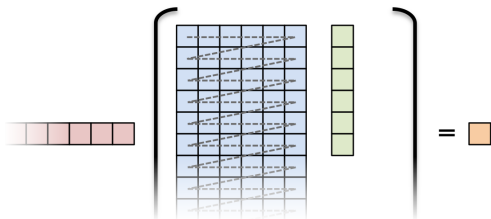
LayoutLeft

in 2D, “column-major”



LayoutRight

in 2D, “row-major”



Important concept: Layout

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

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

Important concept: Layout

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

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

- ▶ Most-common layouts are `LayoutLeft` and `LayoutRight`.
 `LayoutLeft`: left-most index is stride 1.
 `LayoutRight`: right-most index is stride 1.
- ▶ If no layout specified, default for that memory space is used.
 `LayoutLeft` for `CudaSpace`, `LayoutRight` for `HostSpace`.
- ▶ Layouts are extensible: ≈ 50 lines
- ▶ Advanced layouts: `LayoutStride`, `LayoutTiled`, ...

Details:

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

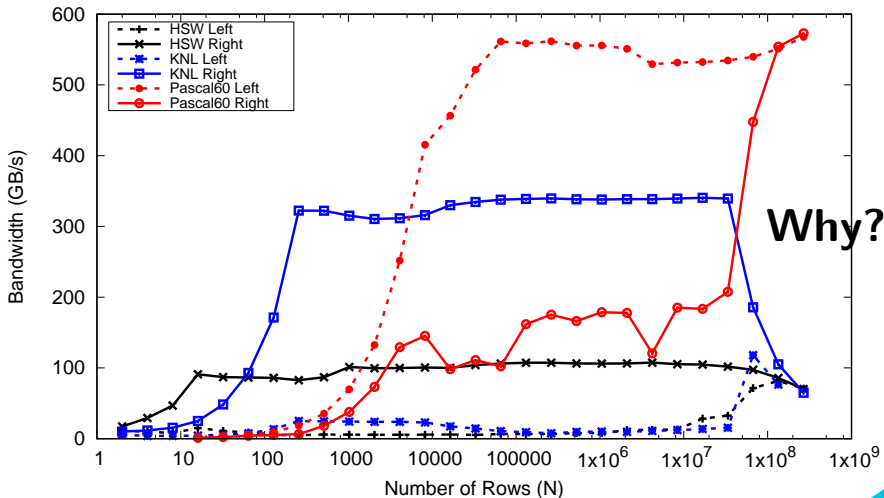
Things to try:

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

Exercise #4: Inner Product, Flat Parallelism

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

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



Thread independence:

```
operator()(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;  
}
```

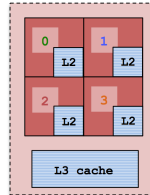
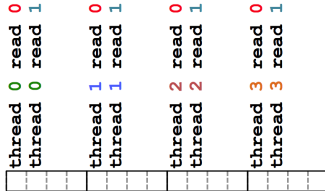
Question: once a thread reads *d*, does it need to wait?

- ▶ **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.

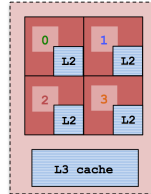
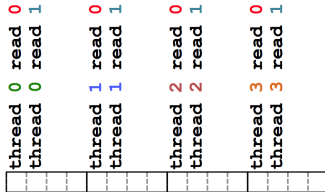
In particular, all threads in a group (*warp* or *wavefront*) must finish their loads before *any* thread can move on.

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

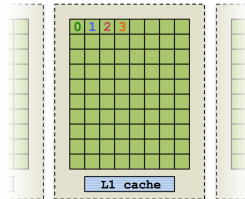
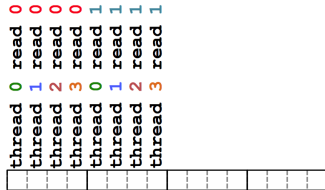
CPUs: few (independent) cores with separate caches:



CPU: few (independent) cores with separate caches:



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



Important point

For performance, accesses to views in HostSpace must be **cached**, while access to views in CudaSpace must be **coalesced**.

Caching: if thread t 's current access is at position i , thread t 's next access should be at position $i+1$.

Coalescing: if thread t 's current access is at position i , thread $t+1$'s current access should be at position $i+1$.

Important point

For performance, accesses to views in HostSpace must be **cached**, while access to views in CudaSpace must be **coalesced**.

Caching: if thread t 's current access is at position i , thread t 's next access should be at position $i+1$.

Coalescing: if thread t 's current access is at position i , thread $t+1$'s current access should be at position $i+1$.

Warning

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

Consider the array summation example:

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

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

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

Consider the array summation example:

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

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

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

Given P threads, **which indices** do we want thread 0 to handle?

Contiguous:

0, 1, 2, ..., N/P

Strided:

0, N/P, 2*N/P, ...

Consider the array summation example:

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

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

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

Given P threads, **which indices** do we want thread 0 to handle?

Contiguous:

0, 1, 2, ..., N/P

CPU

Strided:

0, N/P, 2*N/P, ...

GPU

Why?

Iterating for the execution space:

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

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

Iterating for the execution space:

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

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

Important point

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

Rule of Thumb

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

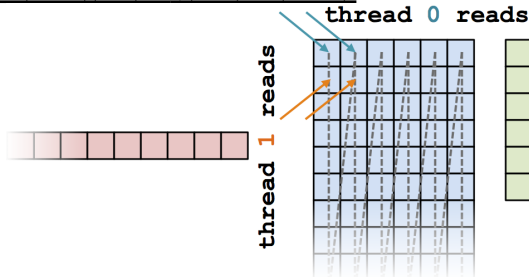
Example:

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

Important point

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

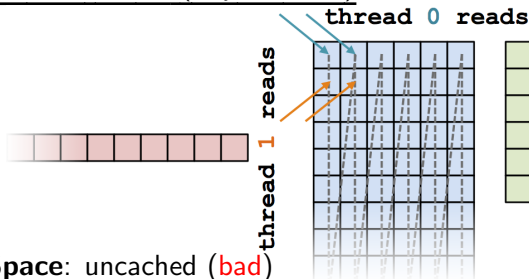
Analysis: column-major (LayoutLeft)



Important point

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

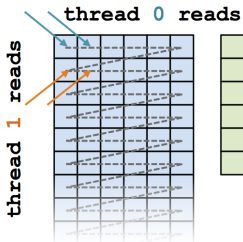
Analysis: column-major (LayoutLeft)



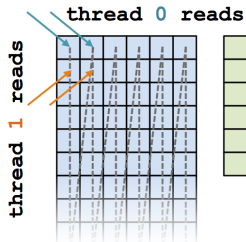
- ▶ **HostSpace**: uncached (**bad**)
- ▶ **CudaSpace**: coalesced (**good**)

Analysis: Kokkos architecture-dependent

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



(a) OpenMP

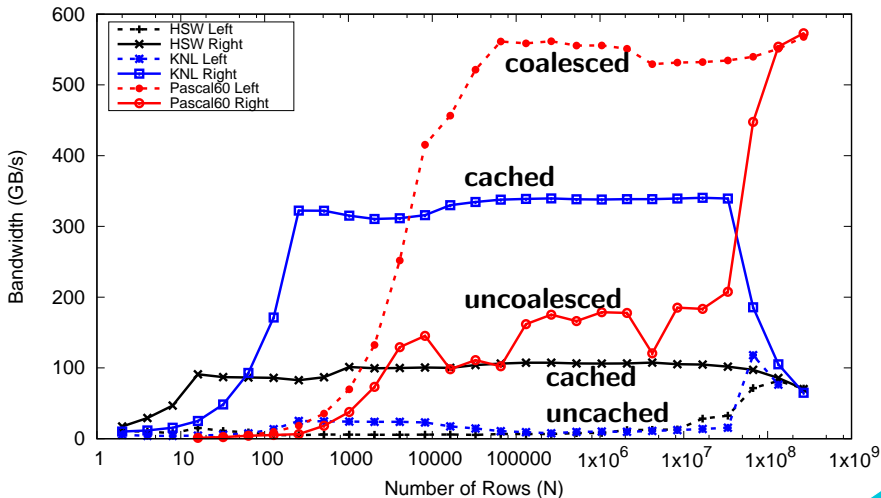


(b) Cuda

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

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

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



- ▶ Every View has a Layout set at compile-time through a **template parameter**.
- ▶ LayoutRight and LayoutLeft are **most common**.
- ▶ Views in HostSpace default to LayoutRight and Views in CudaSpace default to LayoutLeft.
- ▶ Layouts are **extensible** and **flexible**.
- ▶ For performance, memory access patterns must result in **caching** on a CPU and **coalescing** on a GPU.
- ▶ Kokkos maps parallel work indices *and* multidimensional array layout for **performance portable memory access patterns**.
- ▶ There is **nothing in** OpenMP, OpenACC, or OpenCL to manage layouts.
⇒ You'll need multiple versions of code or pay the performance penalty.

Advanced Reductions

Learning objectives:

- ▶ How to use Reducers to perform different reductions.
- ▶ How to do multiple reductions in one kernel.
- ▶ Using `Kokkos::View`'s as result for asynchronicity.

So far only "sum" reduction. What about other things?

Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
  KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
  }, Kokkos::Max<double>(max_value));
```

So far only "sum" reduction. What about other things?

Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
    KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
        double my_value = function(...);
        if(my_value > valueToUpdate) valueToUpdate = my_value;
    }, Kokkos::Max<double>(max_value));
```

- Note how the operation in the body matches the reducer op!

So far only "sum" reduction. What about other things?

Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
    KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

- ▶ Note how the operation in the body matches the reducer op!
- ▶ The scalar type is used as a template argument.

So far only "sum" reduction. What about other things?

Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
    KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
    double my_value = function(...);
    if(my_value > valueToUpdate) valueToUpdate = my_value;
}, Kokkos::Max<double>(max_value));
```

- ▶ Note how the operation in the body matches the reducer op!
- ▶ The scalar type is used as a template argument.
- ▶ Many reducers available: Sum, Prod, Min, Max, MinLoc, ... see: <https://github.com/kokkos/kokkos/wiki/Data-Parallelism>

So far only "sum" reduction. What about other things? Using a Reducer:

```
double max_value = 0;
parallel_reduce("Label", numberOfIntervals,
    KOKKOS_LAMBDA(const int64_t i, double & valueToUpdate) {
        double my_value = function(...);
        if(my_value > valueToUpdate) valueToUpdate = my_value;
    }, Kokkos::Max<double>(max_value));
```

- ▶ Note how the operation in the body matches the reducer op!
- ▶ The scalar type is used as a template argument.
- ▶ Many reducers available: Sum, Prod, Min, Max, MinLoc, ... see: <https://github.com/kokkos/kokkos/wiki/Data-Parallelism>
- ▶ Some reducers (like MinLoc) use special scalar types!

```
MinLoc<T,I,S>::value_type result;
parallel_reduce("Label", N, Functor, MinLoc<T,I,S>(result));
```

Sometimes multiple reductions are needed

- ▶ New experimental feature in Kokkos (version 3.2)
- ▶ Provide multiple reducers/result arguments
- ▶ Functor/Lambda operator takes matching thread-local variables
- ▶ Mixing scalar types is fine.

```
float max_value = 0;
double sum = 0;
parallel_reduce("Label", numberOfIntervals,
    KOKKOS_LAMBDA(const int64_t i, float& tl_max, double& tl_sum){
    float a_i = a[i];
    if(a_i > tl_max) tl_max = a_i;
    tl_sum += a_i;
}, Kokkos::Max<float>(max_value), sum);
```

Reducing into a Scalar is blocking!

- ▶ Providing a reference to scalar means no lifetime expectation.
 - ▶ Call to `parallel_reduce` returns after writing the result.
- ▶ `Kokkos::View` can be used as a result, allowing for potentially non-blocking execution.
- ▶ Can provide `View` to host memory, or to memory accessible by the `ExecutionSpace` for the reduction.
- ▶ Works with Reducers too!

```
View<double,HostSpace> h_sum("sum_h");  
View<double,CudaSpace> d_sum("sum_d");  
using policy_t = RangePolicy<Cuda>;  
  
parallel_reduce("Label", policy_t(0,N), SomeFunctor,  
    h_sum);  
  
parallel_reduce("Label", policy_t(0,N), SomeFunctor,  
    Kokkos::Sum<double,CudaSpace>(d_sum));
```

Subviews: Taking slices of Views

Learning objectives:

- ▶ Introduce Kokkos::subview—basic capabilities and syntax
- ▶ Suggested usage and practices
- ▶ View assignment rules

Sometimes you have to call functions on a subset of data:

Sometimes you have to call functions on a subset of data:

Example: call a frobenius norm on a matrix slice of a rank-3 tensor:

```
double special_norm(View<double***> tensor, int i) {  
  
    auto matrix = ???;  
    // Call a function that takes a matrix:  
    return some_library::frobenius_norm(matrix);  
}
```

Sometimes you have to call functions on a subset of data:

Example: call a frobenius norm on a matrix slice of a rank-3 tensor:

```
double special_norm(View<double***> tensor, int i) {  
  
    auto matrix = ???;  
    // Call a function that takes a matrix:  
    return some_library::frobenius_norm(matrix);  
}
```

In Fortran or Matlab or Python you can get such a slice:

```
tensor(i,:,:)
```


Sometimes you have to call functions on a subset of data:

Example: call a frobenius norm on a matrix slice of a rank-3 tensor:

```
double special_norm(View<double***> tensor, int i) {  
  
    auto matrix = ???;  
    // Call a function that takes a matrix:  
    return some_library::frobenius_norm(matrix);  
}
```

In Fortran or Matlab or Python you can get such a slice:

```
tensor(i,:,:)
```

Kokkos can do that too!

Subview

`Kokkos::subview` can be used to get a view to a subset of an existing View.

Subview description:

- ▶ A subview is a “slice” of a View

Subview description:

- ▶ A subview is a “slice” of a View
 - ▶ The function template `Kokkos::subview()` takes a `View` and a slice for each dimension and returns a `View` of the appropriate shape.

Subview description:

- ▶ A subview is a “slice” of a View
 - ▶ The function template `Kokkos::subview()` takes a View and a slice for each dimension and returns a View of the appropriate shape.
 - ▶ The subview and original View point to the same data—no extra memory allocation nor copying

Subview description:

- ▶ A subview is a “slice” of a View
 - ▶ The function template `Kokkos::subview()` takes a View and a slice for each dimension and returns a View of the appropriate shape.
 - ▶ The subview and original View point to the same data—no extra memory allocation nor copying
- ▶ Can be constructed on host or within a kernel, since no allocation of memory occurs

Subview description:

- ▶ A subview is a “slice” of a View
 - ▶ The function template `Kokkos::subview()` takes a View and a slice for each dimension and returns a View of the appropriate shape.
 - ▶ The subview and original View point to the same data—no extra memory allocation nor copying
- ▶ Can be constructed on host or within a kernel, since no allocation of memory occurs
- ▶ Similar capability as provided by Matlab, Fortran, Python, etc., using “colon” notation

Introductory Usage Demo:

Given a View:

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

Introductory Usage Demo:

Given a View:

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

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

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


Introductory Usage Demo:

Given a View:

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

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

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

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

```
auto sv_i0 =  
    Kokkos::subview(v, i0, Kokkos::ALL, Kokkos::ALL);  
  
// Just like in Python, you can do the same thing with  
// the equivalent of v(i0, 0:N1, 0:N2)  
auto sv_i0_other =  
    Kokkos::subview(v, i0, Kokkos::make_pair(0, N1),  
                    Kokkos::make_pair(0, N2));
```

Subview can take three types of slice arguments:

▶ Index

- ▶ For every index i the rank of the resulting View is decreased by one.
- ▶ Must be between $0 \leq i < \text{extent}(\text{dim})$

▶ Kokkos::pair

- ▶ References a half-open range of indices.
- ▶ The begin and end must be within the extents of the original view.

▶ Kokkos::ALL

- ▶ References the entire extent in that dimension.
- ▶ Equivalent to providing `make_pair(0, v.extent(dim))`

Usage notes:

- ▶ Use auto for the type of a subview (unless you can't)

Usage notes:

- ▶ Use auto for the type of a subview (unless you can't)

Usage notes:

- ▶ Use auto for the type of a subview (unless you can't)
 - ▶ The return type of `Kokkos::subview()` is implementation defined for performance reasons

Usage notes:

- ▶ Use `auto` for the type of a subview (unless you can't)
 - ▶ The return type of `Kokkos::subview()` is implementation defined for performance reasons
 - ▶ You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere

Usage notes:

- ▶ Use `auto` for the type of a subview (unless you can't)
 - ▶ The return type of `Kokkos::subview()` is implementation defined for performance reasons
 - ▶ You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- ▶ Use `Kokkos::pair` for partial ranges if subview created within a kernel

Usage notes:

- ▶ Use `auto` for the type of a subview (unless you can't)
 - ▶ The return type of `Kokkos::subview()` is implementation defined for performance reasons
 - ▶ You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- ▶ Use `Kokkos::pair` for partial ranges if subview created within a kernel
- ▶ Constructing subviews in inner loop code can have performance implications (for now...)

Usage notes:

- ▶ Use `auto` for the type of a subview (unless you can't)
 - ▶ The return type of `Kokkos::subview()` is implementation defined for performance reasons
 - ▶ You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- ▶ Use `Kokkos::pair` for partial ranges if subview created within a kernel
- ▶ Constructing subviews in inner loop code can have performance implications (for now...)
 - ▶ This will likely be far less of an issue in the future.

Usage notes:

- ▶ Use `auto` for the type of a subview (unless you can't)
 - ▶ The return type of `Kokkos::subview()` is implementation defined for performance reasons
 - ▶ You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- ▶ Use `Kokkos::pair` for partial ranges if subview created within a kernel
- ▶ Constructing subviews in inner loop code can have performance implications (for now...)
 - ▶ This will likely be far less of an issue in the future.
 - ▶ Prioritize readability and maintainability first, then make changes only if you see a performance impact.

Usage notes:

- ▶ Use `auto` for the type of a subview (unless you can't)
 - ▶ The return type of `Kokkos::subview()` is implementation defined for performance reasons
 - ▶ You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- ▶ Use `Kokkos::pair` for partial ranges if subview created within a kernel
- ▶ Constructing subviews in inner loop code can have performance implications (for now...)
 - ▶ This will likely be far less of an issue in the future.
 - ▶ Prioritize readability and maintainability first, then make changes only if you see a performance impact.

Details:

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

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

`View::operator=()` **just does the “Right Thing”™**

▶ `View<int**> a; a = View<int*[5]>("b", 4)`

`View::operator=()` just does the “Right Thing”™

▶ `View<int**> a; a = View<int*[5]>("b", 4) ==> Okay`

`View::operator=()` **just does the “Right Thing”™**

- ▶ `View<int**> a; a = View<int*[5]>("b", 4) ==> Okay`
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4) ==> Okay`
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
`==> Okay, checked at runtime`

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4) ==> Okay`
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
`==> Okay, checked at runtime`
- ▶ `View<int*[5]> a; a = View<int*[3]>("b", 4)`

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
=> Okay, checked at runtime
- ▶ `View<int*[5]> a; a = View<int*[3]>("b", 4)`
=> Compilation error

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
=> Okay, checked at runtime
- ▶ `View<int*[5]> a; a = View<int*[3]>("b", 4)`
=> Compilation error
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 3)`

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
=> Okay, checked at runtime
- ▶ `View<int*[5]> a; a = View<int*[3]>("b", 4)`
=> Compilation error
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 3)`
=> Runtime error

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4) ==> Okay`
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
`==> Okay, checked at runtime`
- ▶ `View<int*[5]> a; a = View<int*[3]>("b", 4)`
`==> Compilation error`
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 3)`
`==> Runtime error`
- ▶ `View<int*, CudaSpace> a;`
`a = View<int*, HostSpace>("b", 4)`

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
=> Okay, checked at runtime
- ▶ `View<int*[5]> a; a = View<int*[3]>("b", 4)`
=> Compilation error
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 3)`
=> Runtime error
- ▶ `View<int*, CudaSpace> a;`
`a = View<int*, HostSpace>("b", 4)`
=> Compilation error
- ▶ `View<int**, LayoutLeft> a;`
`a = View<int**, LayoutRight>("b", 4, 5)`

`View::operator=()` just does the “Right Thing”™

- ▶ `View<int**> a; a = View<int*[5]>("b", 4) => Okay`
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 5)`
=> Okay, checked at runtime
- ▶ `View<int*[5]> a; a = View<int*[3]>("b", 4)`
=> Compilation error
- ▶ `View<int*[5]> a; a = View<int**>("b", 4, 3)`
=> Runtime error
- ▶ `View<int*, CudaSpace> a;`
`a = View<int*, HostSpace>("b", 4)`
=> Compilation error
- ▶ `View<int**, LayoutLeft> a;`
`a = View<int**, LayoutRight>("b", 4, 5)`
=> Compilation error

`View::operator=()` **just does the “Right Thing”™**

▶ `View<const int*> a; a = View<int*>("b", 4)`

`View::operator=()` **just does the “Right Thing”™**

- ▶ `View<const int*> a; a = View<int*>("b", 4)`
=> Okay

`View::operator=()` **just does the “Right Thing”™**

- ▶ `View<const int*> a; a = View<int*>("b", 4)`
=> Okay
- ▶ `View<int*> a; a = View<const int*>("b", 4)`

`View::operator=()` **just does the “Right Thing”™**

- ▶ `View<const int*> a; a = View<int*>("b", 4)`
=> Okay
- ▶ `View<int*> a; a = View<const int*>("b", 4)`
=> Compilation error

`View::operator=()` **just does the “Right Thing”™**

- ▶ `View<const int*> a; a = View<int*>("b", 4)`
=> Okay
- ▶ `View<int*> a; a = View<const int*>("b", 4)`
=> Compilation error
- ▶ `View<int*[5], LayoutStride> a;`
`a = View<int*[5], LayoutLeft>("b", 4)`

`View::operator=()` **just does the “Right Thing”™**

- ▶ `View<const int*> a; a = View<int*>("b", 4)`
=> Okay
- ▶ `View<int*> a; a = View<const int*>("b", 4)`
=> Compilation error
- ▶ `View<int*[5], LayoutStride> a;`
`a = View<int*[5], LayoutLeft>("b", 4)` => Okay,
converting compile-time strides into runtime strides

`View::operator=()` **just does the “Right Thing”™**

- ▶ `View<const int*> a; a = View<int*>("b", 4)`
=> Okay
- ▶ `View<int*> a; a = View<const int*>("b", 4)`
=> Compilation error
- ▶ `View<int*[5], LayoutStride> a;`
`a = View<int*[5], LayoutLeft>("b", 4)` => Okay,
converting compile-time strides into runtime strides
- ▶ `View<int*[5], LayoutLeft> a;`
`a = View<int*[5], LayoutStride>("b", 4)`

`View::operator=()` just does the “Right Thing”™

- ▶ `View<const int*> a; a = View<int*>("b", 4)`
=> Okay
- ▶ `View<int*> a; a = View<const int*>("b", 4)`
=> Compilation error
- ▶ `View<int*[5], LayoutStride> a;`
`a = View<int*[5], LayoutLeft>("b", 4)` => Okay,
converting compile-time strides into runtime strides
- ▶ `View<int*[5], LayoutLeft> a;`
`a = View<int*[5], LayoutStride>("b", 4)` => Okay,
but only if strides match layout left (checked at runtime)

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

```
► View<int***> a;  
a = Kokkos::subview(v, ALL, 42, ALL);
```


Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

```
► View<int***> a;  
a = Kokkos::subview(v, ALL, 42, ALL);  
=> Compilation error
```

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ▶ `View<int***> a;`
`a = Kokkos::subview(v, ALL, 42, ALL);`
=> Compilation error
- ▶ `View<int*> a;`
`a = Kokkos::subview(v, ALL, 5, 42);`

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ▶ `View<int***> a;`
`a = Kokkos::subview(v, ALL, 42, ALL);`
`=> Compilation error`
- ▶ `View<int*> a;`
`a = Kokkos::subview(v, ALL, 5, 42);`
`=> Okay for LayoutLeft but => Compilation error for LayoutRight`

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ▶ `View<int***> a;`
`a = Kokkos::subview(v, ALL, 42, ALL);`
=> Compilation error
- ▶ `View<int*> a;`
`a = Kokkos::subview(v, ALL, 5, 42);`
=> Okay for LayoutLeft but => Compilation error for LayoutRight
- ▶ `View<int**> a;`
`a = Kokkos::subview(v, ALL, 15, ALL);`

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ▶ `View<int***> a;`
`a = Kokkos::subview(v, ALL, 42, ALL);`
=> **Compilation error**
- ▶ `View<int*> a;`
`a = Kokkos::subview(v, ALL, 5, 42);`
=> **Okay for LayoutLeft** but => **Compilation error for LayoutRight**
- ▶ `View<int**> a;`
`a = Kokkos::subview(v, ALL, 15, ALL);`
=> **Runtime error (!)**

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ▶ `View<int***> a;`
`a = Kokkos::subview(v, ALL, 42, ALL);`
=> Compilation error
- ▶ `View<int*> a;`
`a = Kokkos::subview(v, ALL, 5, 42);`
=> Okay for LayoutLeft but => Compilation error for LayoutRight
- ▶ `View<int**> a;`
`a = Kokkos::subview(v, ALL, 15, ALL);`
=> Runtime error (!)
- ▶ `View<int**, LayoutStride> a;`
`a = Kokkos::subview(v, ALL, 15, ALL);`

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ▶ `View<int***> a;`
`a = Kokkos::subview(v, ALL, 42, ALL);`
=> **Compilation error**
- ▶ `View<int*> a;`
`a = Kokkos::subview(v, ALL, 5, 42);`
=> **Okay for LayoutLeft** but => **Compilation error for LayoutRight**
- ▶ `View<int**> a;`
`a = Kokkos::subview(v, ALL, 15, ALL);`
=> **Runtime error (!)**
- ▶ `View<int**, LayoutStride> a;`
`a = Kokkos::subview(v, ALL, 15, ALL);`
=> **Okay**

- ▶ Use subviews to get a portion of a View. Helps with:
 - ▶ code reuse
 - ▶ code readability
 - ▶ library function compatibility

- ▶ Use subviews to get a portion of a View. Helps with:
 - ▶ code reuse
 - ▶ code readability
 - ▶ library function compatibility
- ▶ Kokkos supports slicing Views similar to Python/Matlab/Fortran slicing syntax

```
auto sv = Kokkos::subview(v, 42, ALL, std::make_pair(3, 17));
```

- ▶ Use subviews to get a portion of a View. Helps with:
 - ▶ code reuse
 - ▶ code readability
 - ▶ library function compatibility

- ▶ Kokkos supports slicing Views similar to Python/Matlab/Fortran slicing syntax

```
auto sv = Kokkos::subview(v, 42, ALL, std::make_pair(3, 17));
```

- ▶ The return type of subview is complicated. Use **auto!!**
- ▶ `View::operator=()` just does the “Right Thing”™
 - ▶ So generally don't worry about it at first! This is advanced stuff, and more for future reference.

Tightly Nested Loops with MDRangePolicy

Learning objectives:

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

Motivating example: Consider the nested for loops:

```
for ( int i = 0; i < 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 N_i is too small to amortize the cost of constructing a parallel region, but $N_i * N_j * N_k$ 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.

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

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);
```

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'.

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.

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.

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);
```

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;  
    });
```

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`

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!

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
```

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/lambdaes 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:

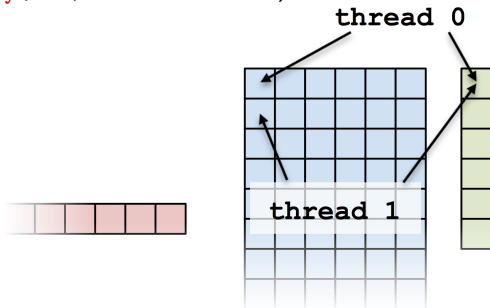
- ▶ Similarities and differences between outer and inner levels of parallelism
- ▶ Thread teams (league of teams of threads)
- ▶ Performance improvement with well-coordinated teams

(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);

```



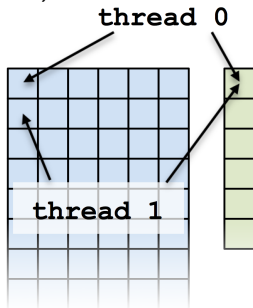
(Flat parallel) Kernel:

```

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?



(Flat parallel) Kernel:

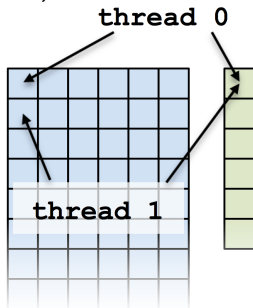
```

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?



(Flat parallel) Kernel:

```

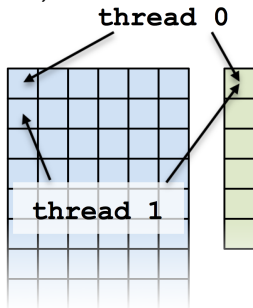
Kokkos::parallel_reduce("yAx",N,
  KOKKOS_LAMBDA (const int row, double & valueToUpdate) {
    double thisRowsSum = 0;
    for (int col = 0; col < M; ++col) {
      thisRowsSum += A(row,col) * x(col);
    }
    valueToUpdate += y(row) * thisRowsSum;
  }, result);

```

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

Solutions?

- ▶ Atomics
- ▶ Thread teams



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);
}
```

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**.

Important concept: Thread team

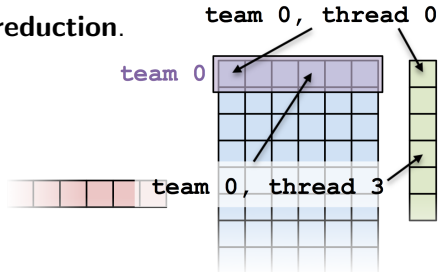
A collection of threads which are guaranteed to be executing **concurrently** and **can synchronize**.

Important concept: Thread team

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

High-level **strategy**:

1. Do **one parallel launch** of N teams.
2. Each team handles a row.
3. The threads within **teams perform a reduction**.
4. The thread teams **perform a reduction**.



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);
```

Important point

Using teams is changing the execution *policy*.

“**Flat** parallelism” uses RangePolicy:

We specify a *total amount of work*.

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

Important point

Using teams is changing the execution *policy*.

“**Flat** parallelism” uses RangePolicy:

We specify a *total amount of work*.

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

“**Hierarchical** parallelism” uses TeamPolicy:

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

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

Important point

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

```
typedef typename TeamPolicy<ExecSpace>::member_type member_type;

void operator()(const member_type & teamMember) {
    // 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();
}
```


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;  
    }  
}
```

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.

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.

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.

⇒ **Nested parallel patterns**

TeamThreadRange:

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

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.

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

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


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

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

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**

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

Details:

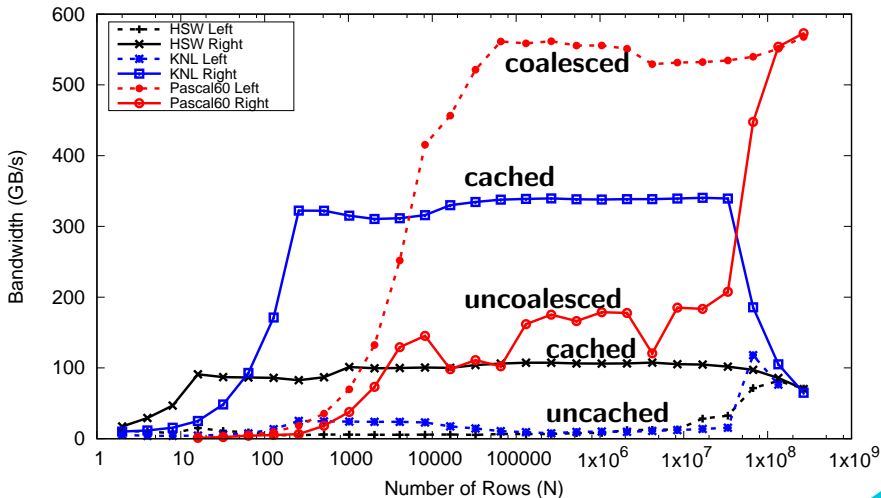
- ▶ Location: Exercises/team_policy/
- ▶ Replace `RangePolicy<Space>` with `TeamPolicy<Space>`
- ▶ Use `AUTO` for `team_size`
- ▶ Make the inner loop a `parallel_reduce` with `TeamThreadRange` policy
- ▶ Experiment with the combinations of `Layout`, `Space`, `N` to view performance
- ▶ Hint: what should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

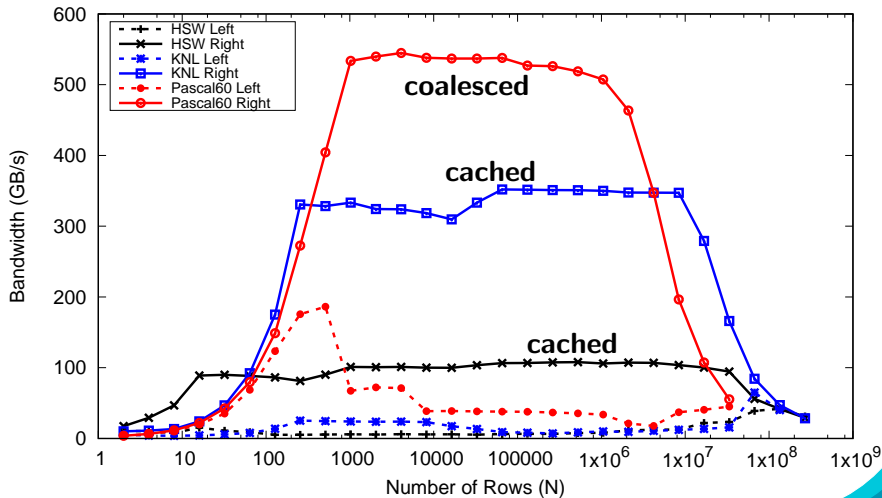
<y|Ax> Exercise 04 (Layout) Fixed Size

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



<y|Ax> Exercise 05 (Layout/Teams) Fixed Size

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



Exposing Vector Level Parallelism

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

```

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);
```


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

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

`totalSum = numberOfThreads * 10`

Question: What will the value of totalSum be?

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

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

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);
```

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

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
```

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

Details:

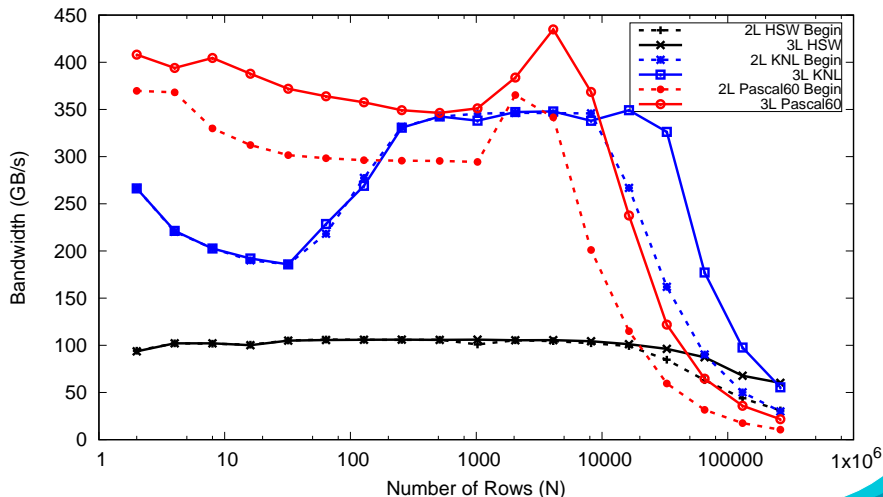
- ▶ Location: `Exercises/team_vector_loop/`
- ▶ Use the `single` policy instead of checking team rank
- ▶ Parallelize all three loop levels.

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

<y|Ax> Exercise 06 (Three Level Parallelism) Fixed Size

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



- ▶ **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.

Kokkos Tools

Leveraging Kokkos' built-in instrumentation.

Learning objectives:

- ▶ The need for Kokkos aware tools.
- ▶ How instrumentation helps.
- ▶ Simple profiling tools.
- ▶ Simple debugging tools.

Output from NVIDIA NVProf for Trilinos Tpetra

```

==278743== Profiling application: ./TpetraCore_Performance-CGSolve.exe --size=200
==278743== Profiling result:
   Type  Time(%)   Time    Calls    Avg      Min      Max   Name
GPU activities: 26.09% 380.32ms      1 380.32ms 380.32ms 380.32ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>::pack_functor<K
okkos::View<double*>, Kokkos::View<unsigned long const *>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
22.28% 324.77ms      1 324.77ms 324.77ms 324.77ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal<int, __int64, Kokkos::Device<Kokkos
::Cuda, Kokkos::CudaUVMSpace>, unsigned long, unsigned long>, Kokkos::RangePolicy<>, unsigned long, void>, Kokkos::RangePolicy<>, Kokkos::Invalid
Type, Kokkos::Cuda>>(int)
21.83% 318.26ms      77 4.1332ms 3.8786ms 22.643ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosSparse::Impl::SPMV_Functor<KokkosSparse::CrMatrix<double const, int const, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpac
e>, Kokkos::MemoryTraits<unsigned int-1>, unsigned long const >, Kokkos::View<double const *>, Kokkos::View<double*>, int=0, bool=0>, Kokkos::Te
amPolicy<>, Kokkos::Cuda>>(double const *)
15.51% 226.15ms      1 226.15ms 226.15ms 226.15ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>::pack_functor<K
okkos::View<int*>, Kokkos::View<unsigned long const *>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
3.60% 52.486ms      227 231.22us 230.17us 232.93us void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::Axpyb_Functor<double, Kokkos::View<double const *>, double, Kokkos::View<double*>, int=2, int=2, int>, Kokkos::Ran
gePolicy<>, Kokkos::Cuda>>(double)
1.86% 27.174ms      13 2.0903ms 1.0560us 27.157ms [CUDA memcopy HtoD]
1.81% 26.358ms      153 172.22us 138.27us 206.08us void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<KokkosBlas::Impl::DotFunctor<Kokkos::View<double>, Kokkos::View<double const *>, Kokkos::View<double const *>, int>, Kokkos::Ran
gePolicy<>, Kokkos::InvalidType, Kokkos::Cuda>>(double)
1.61% 23.431ms      1 23.431ms 23.431ms 23.431ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
1.39% 20.299ms      1 20.299ms 20.299ms 20.299ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)

```

Output from NVIDIA NVProf for Trilinos Tpetra

```

==278743== Profiling application: ./TpetraCore_Performance-CGSolve.exe --size=200
==278743== Profiling result:
   Type  Time(%)   Time    Calls    Avg      Min      Max   Name
GPU activities: 26.09% 380.32ms    1 380.32ms 380.32ms 380.32ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>::pack_functor<K
okkos::View<double*>, Kokkos::View<unsigned long const *>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
22.28% 324.77ms    1 324.77ms 324.77ms 324.77ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal<int, __int64, Kokkos::Device<Kokkos
::Cuda, Kokkos::CudaUVMSpace>, unsigned long, unsigned long>, Kokkos::RangePolicy<>, unsigned long, void>, Kokkos::RangePolicy<>, Kokkos::Invalid
dType, Kokkos::Cuda>>(int)
21.83% 318.26ms    77 4.1332ms 3.8786ms 22.643ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosSparse::Impl::SPMV_Functor<KokkosSparse::CrMatrix<double const, int const, Kokkos::Device<Kokkos::Cuda, Kokkos::CudaUVMSpac
e>, Kokkos::MemoryTraits<unsigned int-1>, unsigned long const >, Kokkos::View<double const *>, Kokkos::View<double*>, int=0, bool=0>, Kokkos::Te
amPolicy<>, Kokkos::Cuda>>(double const *)
15.51% 226.15ms    1 226.15ms 226.15ms 226.15ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<Tpetra::CrMatrix<double, int, __int64, Kokkos::Compat::KokkosDeviceWrapperNode<Kokkos::Cuda, Kokkos::CudaUVMSpace>::pack_functor<K
okkos::View<int*>, Kokkos::View<unsigned long const *>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double)
3.60% 52.486ms    227 231.22us 230.17us 232.93us void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::Axpybpy_Functor<double, Kokkos::View<double const *>, double, Kokkos::View<double*>, int=2, int=2, int>, Kokkos::Ran
gePolicy<>, Kokkos::Cuda>>(double)
1.86% 27.174ms    13 2.0903ms 1.0560us 27.157ms [CUDA memcpy HtoD]
1.81% 26.358ms    153 172.22us 138.27us 206.08us void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelReduce<KokkosBlas::Impl::DotFunctor<Kokkos::View<double>, Kokkos::View<double const *>, Kokkos::View<double const *>, int>, Kokkos::Rang
ePolicy<>, Kokkos::InvalidType, Kokkos::Cuda>>(double)
1.61% 23.431ms    1 23.431ms 23.431ms 23.431ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)
1.39% 20.299ms    1 20.299ms 20.299ms 20.299ms void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::
ParallelFor<KokkosBlas::Impl::V_Update_Functor<Kokkos::View<double
int=0, int>, Kokkos::RangePolicy<>, Kokkos::Cuda>>(double const *)

```

What are those Kernels doing?

Generic code obscures what is happening from the tools

Historically a lot of profiling tools are coming from a Fortran and C world:

- ▶ Focused on functions and variables
- ▶ C++ has a lot of other concepts:
 - ▶ Classes with member functions
 - ▶ Inheritance
 - ▶ Template Metaprogramming
- ▶ Abstraction Models (Generic Programming) obscure things
 - ▶ From a profiler perspective interesting stuff happens in the abstraction layer (e.g. `#pragma omp parallel`)
 - ▶ Symbol names get really complex due to deep template layers

Instrumentation enables context information to reach tools.

Most profiling tools have an instrumentation interface

- ▶ E.g. nvtx for NVIDIA, ITT for Intel.
- ▶ Allows to name regions
- ▶ Sometimes can mark up memory operations.

Instrumentation enables context information to reach tools.

Most profiling tools have an instrumentation interface

- ▶ E.g. nvtx for NVIDIA, ITT for Intel.
- ▶ Allows to name regions
- ▶ Sometimes can mark up memory operations.

KokkosP

Kokkos has its own instrumentation interface KokkosP, which can be used to write tools.

- ▶ Knows about parallel dispatch
- ▶ Knows about allocations, deallocations and deep_copy
- ▶ Provides region markers
- ▶ Leverages naming information (kernels, Views)

There are two components to Kokkos Tools: the KokkosP instrumentation interface and the actual Tools.

KokkosP Interface

- ▶ The internal instrumentation layer of Kokkos.
- ▶ Always available even in release builds.
- ▶ Zero overhead if no tool is loaded.

Kokkos Tools

- ▶ Tools leveraging the KokkosP instrumentation layer.
- ▶ Are loaded at runtime by Kokkos.
 - ▶ Set `KOKKOS_PROFILE_LIBRARY` environment variable to load a shared library.
 - ▶ Compile tools into the executable and use the API callback setting mechanism.

Download tools from

<https://github.com/kokkos/kokkos-tools>

- ▶ Tools are largely independent of the Kokkos configuration
 - ▶ May need to use the same C++ standard library.
 - ▶ Use the same tool for CUDA and OpenMP code for example.
- ▶ Simple makefiles that are independent of Kokkos config.
- ▶ In most cases just type make in the specific tool directory.

Loading Tools:

- ▶ Set KOKKOS_PROFILE_LIBRARY environment variable to the full path to the shared library of the tool.
- ▶ Kokkos dynamically loads symbols from the library during initialize and fills function pointers.
- ▶ If no tool is loaded the overhead is a function pointer comparison to nullptr.

```
View<double*> a("A",N);
View<double*, HostSpace> h_a = create_mirror_view(a);

Profiling::pushRegion("Setup");
parallel_for("Init_A",RangePolicy<h_exec_t>(0,N),
    KOKKOS_LAMBDA(int i) { h_a(i) = i; });
deep_copy(a,h_a);
Profiling::popRegion();

Profiling::pushRegion("Iterate");
for(int r=0; r<10; r++) {
    View<double*> tmp("Tmp",N);
    parallel_scan("K_1",RangePolicy<exec_t>(0,N),
        KOKKOS_LAMBDA(int i, double& lsum, bool f) {
            if(f) tmp(i) = lsum;
            lsum += a(i);
        });
    double sum;
    parallel_reduce("K_2",N, KOKKOS_LAMBDA(int i, double& lsum) {
        lsum += tmp(i);
    },sum);
}
Profiling::popRegion();
```

Output of: nvprof ./test.cuda

```

==141309== Profiling application: ./test.cuda
==141309== Profiling result:

```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	40.95%	1.4516ms	20	72.580us	65.215us	81.663us	_ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_12ParallelScanI4mainEULIRdE_NS_11RangePolicyIJNS_4CudaEEEEES6_EEEEvT_
	40.75%	1.4444ms	18	80.246us	1.1520us	1.4186ms	[CUDA memcpy HtoD]
	8.84%	313.34us	11	28.485us	28.415us	28.703us	void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::ParallelFor<Kokkos::Impl::ViewValueFunction<Kokkos::Cuda, double, bool-1>, Kokkos::RangePolicy<>, Kokkos::Cuda>>>(Kokkos::Cuda)
	7.91%	280.25us	10	28.025us	27.423us	29.024us	_ZN6Kokkos4Impl33cuda_parallel_launch_local_memoryINS0_14ParallelReduceINS0_18CudaFunctorAdapterI4mainEULIRdE_NS_11RangePolicyIJNS_4CudaEEEEdvEES8_NS_11InvalidTypeES7_EEEEvT_
	1.20%	42.592us	28	1.5210us	1.3440us	2.1760us	[CUDA memcpy DtoH]
	0.13%	4.5760us	1	4.5760us	4.5760us	4.5760us	Kokkos::_GLOBAL_N_52_tmpxft_0001ee3d_00000000_6_Kokkos_Cuda_Locks_cpp1_ii_915ea793::init_lock_array_kernel_atomic(void)
	0.08%	2.8480us	1	2.8480us	2.8480us	2.8480us	Kokkos::Impl::_GLOBAL_N_55_tmpxft_0001ee3b_00000000_6_Kokkos_Cuda_Instance_cpp1_ii_a8bc5097::query_cuda_kernel_arch(int*)
	0.08%	2.6880us	1	2.6880us	2.6880us	2.6880us	Kokkos::_GLOBAL_N_52_tmpxft_0001ee3d_00000000_6_Kokkos_Cuda_Locks_cpp1_ii_915ea793::init_lock_array_kernel_threadid(int)
	0.06%	2.1440us	2	1.0720us	1.0560us	1.0880us	[CUDA memset]

Output of: nvprof ./test.cuda

```

==141309== Profiling application: ./test.cuda
==141309== Profiling result:
   Type    Time(%)      Time       Calls      Avg       Min       Max    Name
GPU activities:  40.95%  1.4516ms       20  72.580us  65.215us  81.663us  _ZN6Kokkos4Impl133cuda_parallel_launch_local_memoryINS0_12ParallelScanIZ4mainEULIRdE_NS_11RangePolicyIJNS_4CudaEEEEdvsEES8_NS_11InvalidTypeES7_EEEEvT_
   40.75%  1.4444ms       18  80.246us  1.1520us  1.4186ms  [CUDA memcpy HtoD]
   8.84%  313.34us       11  28.485us  28.415us  28.703us  void Kokkos::Impl::cuda_parallel_launch_local_memory<Kokkos::Impl::ParallelFor<Kokkos::Impl::ViewValueFunction<Kokkos::Cuda, double, bool-1>, Kokkos::RangePolicy<>, Kokkos::Cuda>>>(Kokkos::Cuda)
   7.91%  280.25us       10  28.025us  27.423us  29.024us  _ZN6Kokkos4Impl133cuda_parallel_launch_local_memoryINS0_14ParallelReduceINS0_18CudaFunctorAdapterIZ4mainEULIRdE_NS_11RangePolicyIJNS_4CudaEEEEdvsEES8_NS_11InvalidTypeES7_EEEEvT_
   1.20%  42.592us       28  1.5210us  1.3440us  2.1760us  [CUDA memcpy DtoH]
   0.13%  4.5760us       1  4.5760us  4.5760us  4.5760us  Kokkos::GLOBAL_N_52_tpxft_0001ee3d_00000000_6_Kokkos_Cuda_Locks_cpp1_i1_915ea793::init_lock_array_kernel_atomic(void)
   0.08%  2.8480us       1  2.8480us  2.8480us  2.8480us  Kokkos::Impl::GLOBAL_N_55_tpxft_0001ee3b_00000000_6_Kokkos_Cuda_Instance_cpp1_i1_a8bc5097::query_cuda_kernel_arch(int*)
   0.08%  2.6880us       1  2.6880us  2.6880us  2.6880us  Kokkos::GLOBAL_N_52_tpxft_0001ee3d_00000000_6_Kokkos_Cuda_Locks_cpp1_i1_915ea793::init_lock_array_kernel_threadid(int)
   0.06%  2.1440us       2  1.0720us  1.0560us  1.0880us  [CUDA memset]

```

Lets make one larger:

```

_ZN6Kokkos4Impl133cuda_parallel_launch_local_memoryINS0_14ParallelReduceINS0_18CudaFunctorAdapterIZ4mainEULIRdE_NS_11RangePolicyIJNS_4CudaEEEEdvsEES8_NS_11InvalidTypeES7_EEEEvT_

```

And demangled:

```

void Kokkos::Impl::cuda_parallel_launch_local_memory
<Kokkos::Impl::ParallelReduce<Kokkos::Impl::CudaFunctorAdapter
<main::{lambda(int, double&)#1}, Kokkos::RangePolicy<Kokkos::Cuda>
double, void>, Kokkos::Cuda, Kokkos::InvalidType, Kokkos::RangePol
(Kokkos::Impl::ParallelReduce<Kokkos::Impl::CudaFunctorAdapter<
main::{lambda(int, double&)#1}, Kokkos::RangePolicy<Kokkos::Cuda>,
double, void>, Kokkos::Cuda, Kokkos::InvalidType, Kokkos::RangePol

```

Aaa this is horrifying can't we do better??

Aaa this is horrifying can't we do better??

Lets use SimpleKernelTimer from Kokkos Tools:

- ▶ Simple tool producing a summary similar to nvprof
- ▶ Good way to get a rough overview of whats going on
- ▶ Writes a file HOSTNAME-PROCESSID.dat per process
- ▶ Use the reader accompanying the tool to read the data

Usage:

```
git clone git@github.com:kokkos/kokkos-tools
cd kokkos-tools/profiling/simple_kernel_timer
make
export KOKKOS_PROFILE_LIBRARY=${PWD}/kp_kernel_timer.so
export PATH=${PATH}:${PWD}
cd ${WORKDIR}
./text.cuda
kp_reader *.dat
```

Output from SimpleKernelTimer:

```

Regions:
-
  (Region)      0.02977      4      0.00744 147.131 60.772      Iterate
  (Region)      0.00769      4      0.00192 38.010 15.700      Setup
-----
Kernels:
-
  (ParFor)      0.00878      4      0.00220 43.402 17.927      Kokkos::View::initialization [A_mirror]
  (ParScan)     0.00651     40      0.00016 32.178 13.291      K_1
  (ParFor)      0.00191     40      0.00005 9.454 3.905      Kokkos::View::initialization [Tmp]
  (ParRed)      0.00169     40      0.00004 8.372 3.458      K_2
  (ParFor)      0.00100      4      0.00025 4.965 2.051      Init_A
  (ParFor)      0.00033      4      0.00008 1.629 0.673      Kokkos::View::initialization [A]
-----
Summary:
Total Execution Time (incl. Kokkos + non-Kokkos):      0.04899 seconds
Total Time in Kokkos kernels:                          0.02024 seconds
  -> Time outside Kokkos kernels:                      0.02876 seconds
  -> Percentage in Kokkos kernels:                     41.31 %
Total Calls to Kokkos Kernels:                          132

```

Output from SimpleKernelTimer:

```

Regions:
-
  (Region)      0.02977      4      0.00744 147.131 60.772      Iterate
  (Region)      0.00769      4      0.00192 38.010 15.700      Setup
-----
Kernels:
-
                                Kokkos::View::initialization [A_mirror]
  (ParFor)      0.00878      4      0.00220 43.402 17.927
  (ParScan)     0.00651     40      0.00016 32.178 13.291      K_1
  (ParFor)      0.00191     40      0.00005 9.454 3.905      Kokkos::View::initialization [Tmp]
  (ParRed)      0.00169     40      0.00004 8.372 3.458      K_2
  (ParFor)      0.00100      4      0.00025 4.965 2.051      Init_A
  (ParFor)      0.00033      4      0.00008 1.629 0.673      Kokkos::View::initialization [A]
-----
Summary:
Total Execution Time (incl. Kokkos + non-Kokkos):      0.04899 seconds
Total Time in Kokkos kernels:                          0.02024 seconds
  -> Time outside Kokkos kernels:                      0.02876 seconds
  -> Percentage in Kokkos kernels:                     41.31 %
Total Calls to Kokkos Kernels:                          132

```

Will introduce *Regions* later.

Kernel Naming

Naming Kernels avoid seeing confusing Profiler output!

Lets look at Tpetra again with the Simple Kernel Timer Loaded:

At the top we get Region output:

```
Regions:
- CG: global
  (REGION)  0.547101 1 0.547101 26.922698 5.470153
- CG: spmv
  (REGION)  0.323189 77 0.004197 15.904024 3.231379
- CG: axpby
  (REGION)  0.091971 154 0.000597 4.525865 0.919565
- KokkosBlas::axpby[ETI]
  (REGION)  0.055017 228 0.000241 2.707360 0.550081
- KokkosBlas::update[ETI]
  (REGION)  0.030842 2 0.015421 1.517718 0.308370
- CG: dot
  (REGION)  0.028661 153 0.000187 1.410413 0.286568
- KokkosBlas::dot[ETI]
  (REGION)  0.028120 153 0.000184 1.383756 0.281152
```

Then we get kernel output:

Kernels:

```
- Tpetra::CrsMatrix::sortAndMergeIndicesAndValues
  (ParRed)    0.708770  1  0.708770  34.878388  7.086590
- KokkosSparse::spmv<NoTranspose,Dynamic>
  (ParFor)    0.319268  77  0.004146  15.711118  3.192184
- Tpetra::Details::Impl::ConvertColumnIndicesFromGlobalToLocal
  (ParRed)    0.292309  1  0.292309  14.384452  2.922633
- Tpetra::CrsMatrix pack values
  (ParFor)    0.267800  1  0.267800  13.178373  2.677581
- Tpetra::CrsMatrix pack column indices
  (ParFor)    0.157867  1  0.157867  7.768592  1.578422
- KokkosBlas::Axpby::S15
  (ParFor)    0.054251  227  0.000239  2.669699  0.542429
- Kokkos::View::initialization [Tpetra::CrsMatrix::val]
  (ParFor)    0.033584  2  0.016792  1.652666  0.335789
- Kokkos::View::initialization [lgMap]
  (ParFor)    0.033417  2  0.016708  1.644441  0.334118
- KokkosBlas::dot<1D>
  (ParRed)    0.027782  153  0.000182  1.367155  0.277778
```

Understanding MemorySpace Utilization is critical

Three simple tools for understanding memory utilization:

- ▶ MemoryHighWaterMark: just the maximum utilization for each memory space.
- ▶ MemoryUsage: Timeline of memory usage.
- ▶ MemoryEvents: allocation, deallocation and deep_copy.
 - ▶ Name, Memory Space, Pointer, Size

```
# Memory Events
# Time      Ptr              Size      MemSpace  Op      Name
0.000776    0x7f095f600000          8000000
0.000910          0x1cb4680             8000000
0.001571 PushRegion Setup {
0.003754 } PopRegion
0.003756 PushRegion Iterate {
0.004100    0x7f0960000000          8000000          Cuda Allocate  Tmp
0.004451    0x7f0960000000         -8000000          Cuda DeAllocate Tmp
...
0.010350    0x7f0960000000          8000000          Cuda Allocate  Tmp
0.010605    0x7f0960000000         -8000000          Cuda DeAllocate Tmp
0.010753 } PopRegion
0.010753          0x1cb4680             -8000000          Host DeAllocate A_mirror
0.010766    0x7f095f600000         -8000000          Cuda DeAllocate A
```

Adding region markers to capture more code structure

Region Markers are helpful to:

- ▶ Find where time is spent outside of kernels.
- ▶ Group Kernels which belong together.
- ▶ Structure code profiles.
 - ▶ For example bracket *setup* or *solve* phase.

Adding region markers to capture more code structure

Region Markers are helpful to:

- ▶ Find where time is spent outside of kernels.
- ▶ Group Kernels which belong together.
- ▶ Structure code profiles.
 - ▶ For example bracket *setup* or *solve* phase.

Simple Push/Pop interface:

```
Kokkos::Profiling::pushRegion("Label");  
...  
Kokkos::Profiling::popRegion();
```

The simplest tool to leverage regions is the **Space Time Stack**:

- ▶ **Bottom Up** and **Top Down** data representation
- ▶ Can do MPI aggregation if compiled with MPI support
- ▶ Also aggregates memory utilization info.

```
BEGIN KOKKOS PROFILING REPORT:
TOTAL TIME: 0.0100131 seconds
TOP-DOWN TIME TREE:
<average time> <percent of total time> <percent time in Kokkos> <percent MPI imbalance> <remainder> <kernels per second> <number of calls> <name> [type]
=====
-> 6.90e-03 sec 68.9% 33.9% 0.0% 66.1% 4.35e+03 1 Iterate [region]
    |> 1.55e-03 sec 15.5% 100.0% 0.0% ----- 10 K_1 [scan]
    |> 4.04e-04 sec 4.0% 100.0% 0.0% ----- 10 Kokkos::View::initialization [Tmp] [for]
    |> 3.80e-04 sec 3.8% 100.0% 0.0% ----- 10 K_2 [reduce]
-> 1.84e-03 sec 18.4% 98.6% 0.0% 1.4% 1.09e+03 1 SetUp [region]
    |> 1.59e-03 sec 15.9% 100.0% 0.0% ----- 1 "A"="A_mirror" [copy]
    |> 2.21e-04 sec 2.2% 100.0% 0.0% ----- 1 Init_A [for]
-> 6.64e-04 sec 6.6% 100.0% 0.0% ----- 1 Kokkos::View::initialization [A_mirror] [for]
-> 6.68e-05 sec 0.7% 100.0% 0.0% ----- 1 Kokkos::View::initialization [A] [for]

BOTTOM-UP TIME TREE:
...

KOKKOS HOST SPACE:
=====
MAX MEMORY ALLOCATED: 7812.5 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  100.0% A_mirror

KOKKOS CUDA SPACE:
=====
MAX MEMORY ALLOCATED: 15625.0 kB
ALLOCATIONS AT TIME OF HIGH WATER MARK:
  50.0% A
  50.0% Iterate/Tmp

Host process high water mark memory consumption: 161668 kB

END KOKKOS PROFILING REPORT.
```

Non-Blocking Dispatch implies asynchronous error reporting!

```

Profiling::pushRegion("Iterate");
for(int r=0; r<10; r++) {
    parallel_for("K_1",2*N, KOKKOS_LAMBDA(int i) {a(i) = i;});
    printf("Passed point A\n");
    double sum;
    parallel_reduce("K_2",N, KOKKOS_LAMBDA(int i, double& lsum) {
        lsum += a(i); },sum);
}
Profiling::popRegion();

```

Output of the run:

```

./test.cuda
Passed point A
terminate called after throwing an instance of 'std::runtime_error'
  what():  cudaStreamSynchronize(m_stream) error( cudaErrorIllegal
    an illegal memory access was encountered
      Kokkos/kokkos/core/src/Cuda/Kokkos_Cuda_Instance.cpp:312
Traceback functionality not available
Aborted (core dumped)

```

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

The KernelLogger is a tool to localize errors and check the actual runtime flow of a code.

- ▶ As other tools it inserts fences - which check for errors.
- ▶ Prints out Kokkos operations as they happen.

Debugging with Tools

Kokkos Tools can be used to implement Debugging functionality.

The KernelLogger is a tool to localize errors and check the actual runtime flow of a code.

- ▶ As other tools it inserts fences - which check for errors.
- ▶ Prints out Kokkos operations as they happen.

Output from the above test case with KernelLogger:

```
KokkosP: Allocate<Cuda> name: A pointer: 0x7f598b800000 size: 8000
KokkosP: Executing parallel-for kernel on device 0 with unique exe
KokkosP: Kokkos::View::initialization [A]
KokkosP: Execution of kernel 0 is completed.
KokkosP: Entering profiling region: Iterate
KokkosP: Executing parallel-for kernel on device 0 with unique exe
KokkosP: Iterate
KokkosP:    K_1
terminate called after throwing an instance of 'std::runtime_error'
  what():  cudaDeviceSynchronize() error( cudaErrorIllegalAddress)
Traceback functionality not available
```

The standard Kokkos profiling approach

Understand Kokkos Utilization (SimpleKernelTimer)

- ▶ Check how much time in kernels
- ▶ Identify HotSpot Kernels

Run Memory Analysis (MemoryEvents)

- ▶ Are there many allocations/deallocations - 5000/s is OK.
- ▶ Identify temporary allocations which might be able to hoisted

Identify Serial Code Regions (SpaceTimeStack)

- ▶ Add Profiling Regions
- ▶ Find Regions with low fraction of time spend in Kernels

Dive into individual Kernels

- ▶ Use connector tools (next subsection) to analyze kernels.
- ▶ E.g. use roof line analysis to find underperforming code.

Analyse a MiniMD variant with a serious performance issue.

Details:

- ▶ Location: Exercises/tools_minimd/
- ▶ Use standard Profiling Approach.
- ▶ Find the code location which causes the performance issue.
- ▶ Run with `miniMD.exe -s 20`

What should happen:

- ▶ Performance should be
- ▶ About 50% of time in a Force compute kernel
- ▶ About 25% in neighbor list creation

- ▶ Kokkos Tools provide an instrumentation interface **KokkosP** and **Tools** to leverage it.
- ▶ The interface is **always available** - even in release builds.
- ▶ Zero overhead if no tool is loaded during the run.
- ▶ Dynamically load a tool via setting `KOKKOS_PROFILE_LIBRARY` environment variable.
- ▶ Set callbacks directly in code for tools compiled into the executable.

Kokkos Kernels: performance portable BLAS, sparse, dense and graph algorithms

Topics:

- ▶ BLAS kernels.
- ▶ SPARSE math kernels.

BLAS and LAPACK

Learning objectives:

- ▶ Motivation for BLAS/LAPACK functions
- ▶ Algorithm Specialization for Applications
- ▶ Calling BLAS/LAPACK functions

KokkosKernels

- ▶ A single interface to vendor BLAS libraries on heterogenous computing platforms
- ▶ Support user-defined data type e.g., Automatic Differentiation, Ensemble, SIMD, types with Kokkos native implementation
- ▶ Customized performance solution for certain problem sizes
- ▶ Exploring new performance oriented interfaces

Vendor Libraries

- ▶ A user needs to write a different function interface for different computing platforms e.g., MKL vs. CUBLAS
- ▶ Built-in real/complex data types and column/row major data layouts are only supported
- ▶ Code is highly optimized; in practice, higher performance is obtained from larger problem sizes

Algorithm Specialization for Applications

- ▶ Dot-based GEMM
 - ▶ GEMM is used for orthogonalizing Krylov multi-vectors (long skinny matrix)
 - ▶ This particular problem shape does not perform well on CUBLAS
 - ▶ Algorithm is specialized for this shape performing multiple dot products instead of running standard GEMM algorithms
- ▶ Compact Batched BLAS
 - ▶ Application wants to solve many instances of tiny square block dense matrices; e.g., block dimensions of 3, 5, 7, 9, 11, etc.
 - ▶ Difficult to effectively use wide vector length such as AVX512 for this small problem size
 - ▶ A pack of block matrices are inter-leaved and solved simultaneously using vector instructions
 - ▶ Code is trivially vectorized 100% for the applied BLAS and LAPACK operations

Algorithm Specialization for Applications

- ▶ Extended Blas 1 interface: see axpby, update (a, c, b, y, g, z)
 - ▶ $y[i] = g * z[i] + b * y[i] + a * x[i]$
 - ▶ Trilinos Tpetra interface used in Belos iterative solvers
- ▶ See the wiki page for complete list of functions
 - ▶ <https://github.com/kokkos/kokkos-kernels/wiki>

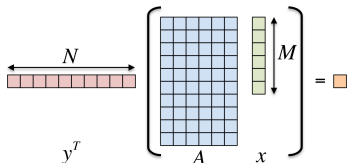
KokkosKernels interacts with application teams and provides custom performance solutions for their needs

Recall the Kokkos Inner Product exercise:

- ▶ Inner product $\langle y, A * x \rangle$

- ▶ y is $N \times 1$, A is $N \times M$,
 x is $M \times 1$

- ▶ Early exercise code looked like

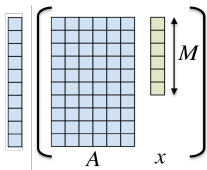


```
double result = 0;
Kokkos::parallel_reduce("yAz", N,
    KOKKOS_LAMBDA (int j, double &update) {
        double temp2 = 0;
        for (int i = 0; i < M; ++i) {
            temp2 += A(j, i) * x(i);
        }
        update += y(j) * temp2;
    }, result);
```

This can be naturally expressed as two BLAS operations:

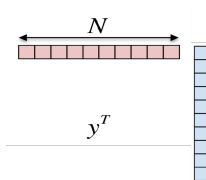
In Matlab notation:

```
// 1. gemv:
Ytmp = A * x
```



```
// 2. dot:
result = y'*Ytmp
```

□□



Different function signatures and APIs are used by different vendors
e.g., on Cuda: cublasDgemv and cublasDdot

`KokkosBlas::gemv (mode, alpha, A, x, beta, y);`

Interface:

- ▶ mode [in]
 - ▶ "N" for non-transpose
 - ▶ "T" for transpose
 - ▶ "C" for conjugate transpose.
- ▶ alpha [in] Input coefficient of $A \cdot x$
- ▶ A [in] Input matrix, as a 2-D Kokkos::View
- ▶ x [in] Input vector, as a 1-D Kokkos::View
- ▶ beta [in] Input coefficient of y
- ▶ y [in/out] Output vector, as a nonconst 1-D Kokkos::View

```
result = KokkosBlas::dot(x,y);
```

Single Interface:

- ▶ x [in] Input vector, as a 1-D Kokkos::View
- ▶ y [in] Input vector, as a 1-D Kokkos::View
- ▶ result [out] Scalar result on host
- ▶ This interface calls Kokkos::fence on all execution spaces

```
KokkosBlas::dot(r,x,y);
```

Single and Multi-vector Interface:

- ▶ x [in] Input (multi-)vector, as a 1-D or 2-D Kokkos::View
- ▶ y [in] Input (multi-)vector, as a 1-D or 2-D Kokkos::View
- ▶ r [in/out] Output result, as a rank-0 or 1-D Kokkos::View
- ▶ This interface is non-blocking.

KokkosKernels:

```
Kokkos::View<double*> tmp("tmp", N);
KokkosBlas::gemv("N", alpha, A, x, beta,
    tmp);

double result = 0;

result = KokkosBlas::dot(y, tmp);
```

- ▶ Uses two BLAS functions
- ▶ Optionally interface to optimized vendor libraries
- ▶ For certain matrix shapes may choose specialized code path for performance

User implementation:

```
double result = 0;
Kokkos::parallel_reduce("yAx", N,
    KOKKOS_LAMBDA (int j, double &
        update) {
    double temp2 = 0;
    for (int i = 0; i < M; ++i) {
        temp2 += A(j, i) * x(i);
    }
    update += y(j) * temp2;
}, result);
```

- ▶ Exploits a single level of parallelism only i.e., internal temp2 is summed sequentially
- ▶ Matrix-vector multiplication and dot product are fused in a single kernel

Related exercise available at: [Exercises/kokkoskernels/InnerProduct](#)

Summary: BLAS/LAPACK

- ▶ Single interface for heterogeneous computing platforms
- ▶ Optimized vendor library interface when it is available
- ▶ Specialization of algorithms corresponding to application needs
- ▶ Native implementation supports strided data layout of a matrix

- ▶ Location: Exercises/kokkoskernels/InnerProduct/Begin/
- ▶ Assignment: We return to the Inner Product example with two separate sub-exercises:
 1. Use Kokkos Kernels BLAS routines (gemv, dot) to compute the inner product
 2. Use Kokkos Kernels team-level BLAS routines (dot) within the TeamPolicy implementation within the matrix-vector product calculation
- ▶ Compile and run on CPU, GPU; test with NVIDIA's cuBLAS tpl enabled

Exercise Inner Product Using Kokkos Kernels

For convenience, use the provided script to install KokkosKernels and generate a Makefile for the exercise

```
./run_installlibs_cmake.sh  
make -j
```

Update path and configuration variables in the script as needed based on your based on your setup:

- ▶ **KOKKOS PATH:** Point to your Kokkos source directory
- ▶ **KOKKOSKERNELS PATH:** KokkosKernels source directory
- ▶ **KOKKOS DEVICES:** Enabled execution spaces
- ▶ **TPLS:** Blank by default, optionally set to cublas with Cuda builds

Run exercise

```
./innerproduct.exe -S 26  
# Note the warnings, set appropriate environment variables
```

- ▶ Vary problem size: **-S #**
- ▶ Vary number of rows: **-N #**
- ▶ Vary repeats: **-nrepeat #**

Sparse Linear Algebra

Sparse linear algebra data structures and functions.

Learning objectives:

- ▶ Key characteristics algorithms
- ▶ Containers: CrsMatrix, StaticCrsGraph, Vector
- ▶ SpMV
- ▶ SpADD
- ▶ SpGEMM

Support for important class of applications

- ▶ Representation of choices for discrete PDE systems (FEM, FD, CVFEM, ...)
- ▶ Natural use for network representation
 - ▶ Electrical grid, electronic circuit
 - ▶ Social network

Unique format supported: Compressed row sparse

Sparse matrices can be stored in various format, currently only Crs format is fully supported, BlockCrs is partially supported

Constraints from Crs format

- ▶ hard to optimize memory access patterns
- ▶ often multi-pass algorithms required
 1. compute storage
 2. compute column index and actual values
- ▶ typically algorithms can be split in symbolic and numeric phases

Symbolic/Numeric split

While extremely useful for reuse it is potentially slower for single use case depending on implementation

Handle: hiding important details!

- ▶ What the handles does for you:
 - ▶ stores user parameters
 - ▶ keeps temporary data needed in numeric of solve/apply phases
 - ▶ cleans up temporary data at destruction
 - ▶ contains kernel specific "sub-handle"
 - ▶ specifies required data types
- ▶ Usage: `KokkosKernels::Experimental::KokkosKernelsHandle<size_type, index_type, scalar_type, ExecutionSpace, TempMemSpace, PermMemSpace>()`

One dense structure:

- ▶ View (of rank 1): represents a "vector"
- ▶ View (of rank 2): represents a "multi-vector"

Two sparse structures:

- ▶ StaticCrsGraph: encodes the sparsity pattern in `row_map` and `entries`
- ▶ CrsMatrix: contains a `StaticCrsGraph` and `values`

Example:

`example/wiki/sparse/KokkosSparse_wiki_crsmatrix.cpp`

Two interfaces for one kernel?

1. Simplified interface

- ▶ uses high level containers
- ▶ reduced number of parameters and templates
- ▶ allocates memory

2. Expert interface

- ▶ uses low level container (i.e. views)
- ▶ allows for finer memory management

Simplified/Expert interface

For clarity we will focus on the simplified interface in the rest of the lecture

SpMV: a mixed sparse/dense kernel

$$0.5 * \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} + 1.0 \begin{bmatrix} 1 & & 2 \\ & 3 & \\ 4 & & 5 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 9 \\ 8.5 \\ 22 \end{bmatrix}$$

- ▶ Computes: $y = \beta * y + \alpha * A * x$
- ▶ Output is a dense vector
 - ▶ single pass algorithm since no CrsGraph needs to be computed
 - ▶ good amount of parallelism exploitable
- ▶ Usage:
`KokkosSparse::spmv(mode, alpha, A, x, beta, y);`
- ▶ Example:
`example/wiki/sparse/KokkosSparse_wiki_spmv.cpp`

SpADD: Sparse Matrix Addition

$$2.0 \begin{bmatrix} 1 & & 2 \\ & 3 & 4 \\ 5 & & \end{bmatrix} + 0.5 \begin{bmatrix} 6 & 7 & \\ & 8 & \\ & & 9 \end{bmatrix} = \begin{bmatrix} 5 & 3.5 & 4 \\ & 10 & 8 \\ 10 & & 4.5 \end{bmatrix}$$

- ▶ Computes: $C = \alpha A + \beta B$ given A and B two CrsMatrices
- ▶ Sorted inputs speeds-up the kernel and reduces memory consumption

- ▶ Usage:

```
KokkosSparse::spadd_symbolic(handle, A, B, C);  
KokkosSparse::spadd_numeric(handle, alpha, A,  
beta, B, C);
```

- ▶ Example:

```
example/wiki/sparse/KokkosSparse_wiki_spadd.cpp
```

SpGEMM: Sparse General Matrix Matrix Multiply

- ▶ Compute $A \times B = C$ for given sparse matrices A and B

$$\begin{bmatrix} 1 & & 2 \\ & 3 & 4 \\ 5 & & \end{bmatrix} \times \begin{bmatrix} 6 & & 7 & \\ 8 & 9 & & \\ & & 10 & 11 \end{bmatrix} = \begin{bmatrix} 6 & & 27 & 22 \\ 24 & 27 & 40 & 41 \\ 30 & 35 & & \end{bmatrix}$$

- ▶ Sparsity structure of C is not known in advance!
- ▶ We have a two-phase implementation:
 - ▶ This allows determining the sparsity of C efficiently

SpGEMM: Sparse General Matrix Matrix Multiply

► Symbolic phase:

- `KokkosSparse::spgemm_symbolic(handle, A, isTrnspsdA, B, isTrnspsdB, C);`
- determines number of nonzeros in each row of *C* and
- allocates memory for column indices and values of the nonzeros

► Numeric phase

- `KokkosSparse::spgemm_numeric(handle, A, isTrnspsdA, B, isTrnspsdB, C);`
- computes column indices and values of the nonzeros of *C*

► Example

`example/wiki/sparse/KokkosSparse_wiki_spgemm.cpp`

SpGEMM: Sparse General Matrix Matrix Multiply

- ▶ We follow Gustavson's algorithm:
 - for** each row index $i \leftarrow 0$ **to** $nrowsA$ **do**
 - for** each column index $j \in A(i, :)$ **do**
 - //accumulate partial row results
 - $C(i, :) \leftarrow C(i, :) + A(i, j)B(j, :)$
- ▶ Our implementation exploits hierarchical paralelism
 - ▶ Teams are assigned contiguous row chunks in A
 - ▶ Threads are assigned individual rows of A
 - ▶ Vector lanes are assigned the nonzeros of rows of B

SpGEMM: Sparse General Matrix Matrix Multiply

- ▶ We follow Gustavson's algorithm:
 for each row index $i \leftarrow 0$ **to** $nrowsA$ **do**
 for each column index $j \in A(i, :)$ **do**
 //accumulate partial row results
 $C(i, :) \leftarrow C(i, :) + A(i, j)B(j, :)$
- ▶ Our thread-scalable hashmap accumulator implementation
 - ▶ is used in both symbolic and numeric phases
 - ▶ supports both sparse and dense accumulators
 - ▶ has a two-level structure: Level-1 (L_1) and Level-2 (L_2)
 - ▶ L_1 hashmap lives in the fast shared memory
 - ▶ L_2 hashmap is created only if L_1 hashmap runs out of memory
 - ▶ L_2 hashmap lives in the large global memory
- ▶ For more details see: M. Deveci, C. Trott, S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures", Parallel Computing 78, 33-46, 2018.

Summary: Sparse Linear Algebra

- ▶ Main difficulties: finding sparsity patterns and memory access
- ▶ Containers: `View`, `StaticCrsGraph` and `CrsMatrix`
- ▶ Kernels: `SpMV`, `SpADD` and `SpGEMM`

This was a short introduction

Didn't cover many things:

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.
- ▶ Advanced data structures.

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.
- ▶ Advanced data structures.
- ▶ Atomic operations and Scatter Contribute patterns.

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.
- ▶ Advanced data structures.
- ▶ Atomic operations and Scatter Contribute patterns.
- ▶ Team Scratch memory (GPU shared memory).

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.
- ▶ Advanced data structures.
- ▶ Atomic operations and Scatter Contribute patterns.
- ▶ Team Scratch memory (GPU shared memory).
- ▶ SIMD vectorization.

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.
- ▶ Advanced data structures.
- ▶ Atomic operations and Scatter Contribute patterns.
- ▶ Team Scratch memory (GPU shared memory).
- ▶ SIMD vectorization.
- ▶ MPI and PGAS integration.

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.
- ▶ Advanced data structures.
- ▶ Atomic operations and Scatter Contribute patterns.
- ▶ Team Scratch memory (GPU shared memory).
- ▶ SIMD vectorization.
- ▶ MPI and PGAS integration.
- ▶ All Tools for Profiling, Debugging and Tuning.

This was a short introduction

Didn't cover many things:

- ▶ Full BuildSystem integration.
- ▶ Advanced data structures.
- ▶ Atomic operations and Scatter Contribute patterns.
- ▶ Team Scratch memory (GPU shared memory).
- ▶ SIMD vectorization.
- ▶ MPI and PGAS integration.
- ▶ All Tools for Profiling, Debugging and Tuning.
- ▶ All Math Kernels (Batched BLAS, Graphs, Sparse Solvers).

The Kokkos Lectures

Join 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

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://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.