

Kokkos Tutorial: Fundamentals

Damien Lebrun-Grandié



ORNL is managed by UT-Battelle LLC
for the US Department of Energy



U.S. DEPARTMENT OF
ENERGY

A Condensed Short Tutorial

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

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

For the full lectures, with more capabilities covered, and more in-depth explanations visit:

<https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series>

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



LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3



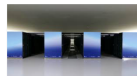
LLNL SIERRA
IBM Power9 / NVIDIA Volta
CUDA / OpenMP^(a)



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.

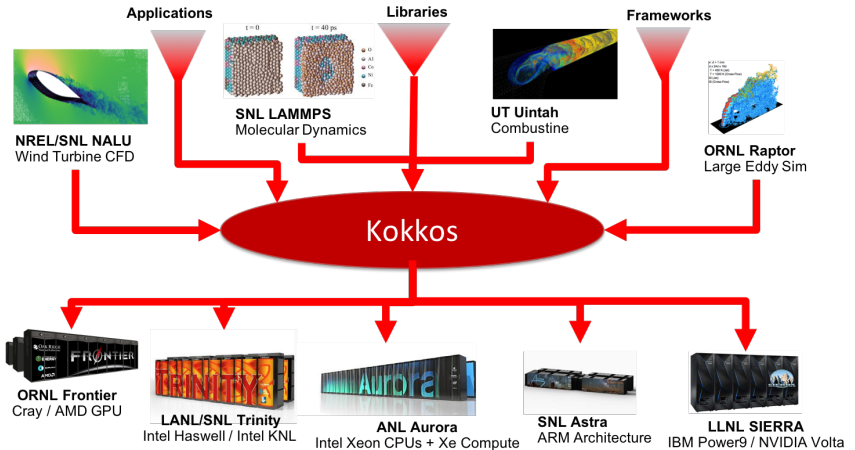
- ▶ Typical HPC production app: 300k-600k lines
 - ▶ Sandia alone maintains a few dozen
- ▶ Large Scientific Libraries:
 - ▶ E3SM: 1,000k lines
 - ▶ Trilinos: 4,000k lines

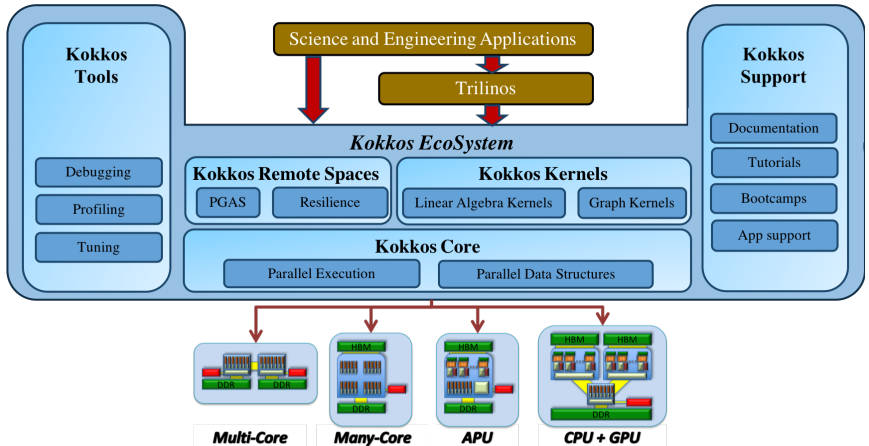
Conservative estimate: need to rewrite 10% of an app to switch Programming Model

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 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:**

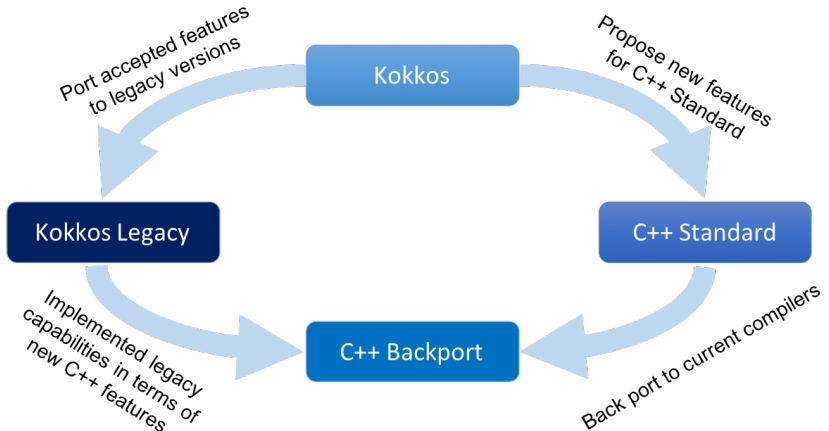
**C. Trott, D. Lebrun-Grandié, D. Arndt, J. Bludau, J. Ciesko, C. Cle-
venger, N. Ellingwood, R. Gayatri, D. Ibanez, D. Lee, S. Lee, N. Liber,
P. Miller, N. Morales, A. Powell, F. Rizzi, M. Simberg, C. Skrzyński,
B. Turcksin**

**former: H.C. Edwards, D. Labreche, G. Mackey, S. Bova, D. Sunder-
land, D.S. Hollman, J. Miles, J. Wilke, J. Madsen, H. Finkel**

Kokkos Kernels:

**S. Rajamanickam, L. Berger-Vergiat, V. Dang, N. Ellingwood, J.
Foucar, E. Harvey, B. Kelley, K. Liegeois, C. Pearson, E. Prudencio**

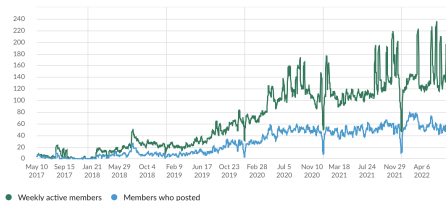
Kokkos helps improve ISO C++



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

Kokkos has a growing OpenSource Community

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



Online Resources:

- ▶ <https://github.com/kokkos>:
 - ▶ Primary Kokkos GitHub Organization
- ▶ <https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series>:
 - ▶ Slides, recording and Q&A for the Full Lectures
- ▶ <https://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

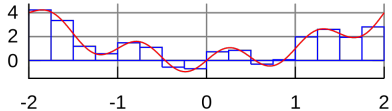
```
parallel_for(N, [=] (const int64_t i) {  
    /* loop body */  
});
```

Important concept

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

Riemann-sum-style numerical integration:

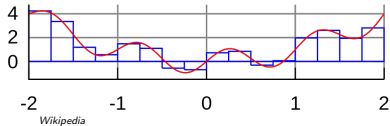
$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



Wikipedia

Riemann-sum-style numerical integration:

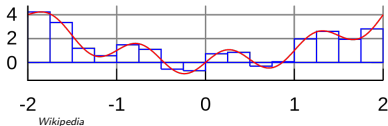
$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



```
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:

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

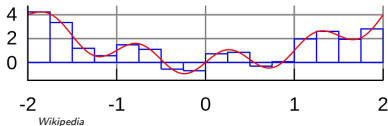


```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

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;

```

Body?

Policy?

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!

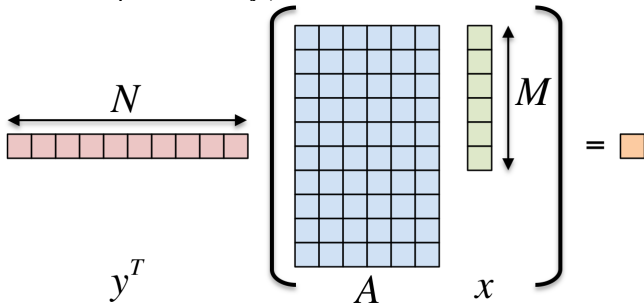
Giving unique names to each kernel is immensely helpful for debugging and profiling. You will regret it if you don't!

- ▶ 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

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

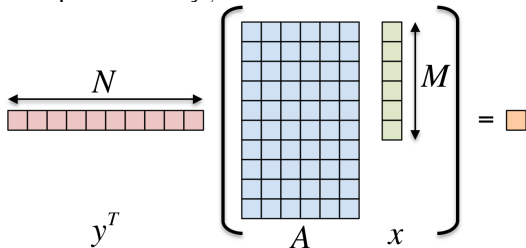
```
#include <Kokkos_Core.hpp>
int main(int argc, char* argv[]) {
    /* ... do any necessary setup (e.g., initialize MPI) ... */
    Kokkos::initialize(argc, argv);
    {
        /* ... do computations ... */
    }
    Kokkos::finalize();
    return 0;
}
```

(Optional) Command-line arguments or environment variables:

<code>--kokkos-num-threads=INT</code> or <code>KOKKOS_NUM_THREADS</code>	total number of threads
<code>--kokkos-device-id=INT</code> or <code>KOKKOS_DEVICE_ID</code>	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=...
```

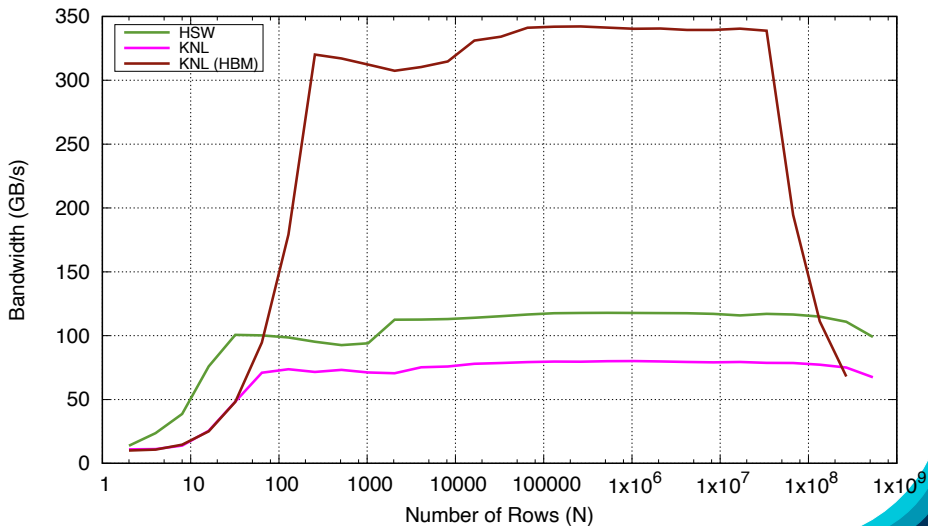
Running on CPU with OpenMP back-end

```
# 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 (shallow 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 (shallow 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 `std::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 `std::shared_ptr`

Example:

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

What gets printed?

View life cycle:

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

Example:

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

What gets printed?

3.0

View Properties:

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

Example:

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

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

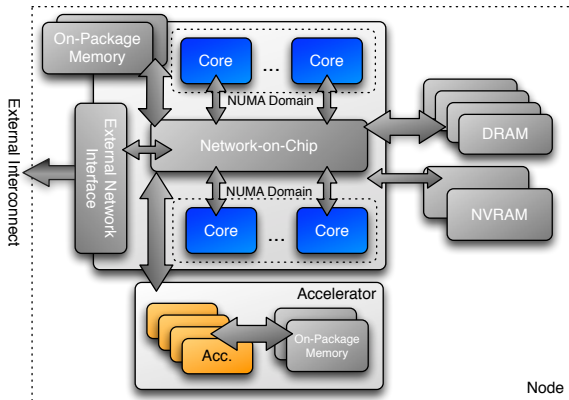
- ▶ Location: Exercises/02/Begin/
- ▶ Assignment: Change data storage from arrays to Views.
- ▶ Compile and run on CPU, and then on GPU with UVM

```
make -j KOKKOS_DEVICES=OpenMP # CPU-only using OpenMP
make -j KOKKOS_DEVICES=Cuda   # GPU - note UVM in Makefile
# Run exercise
./02_Exercise.host -S 26
./02_Exercise.cuda -S 26
# Note the warnings, set appropriate environment variables
```

- ▶ Vary problem size: **-S #**
- ▶ Vary number of rows: **-N #**
- ▶ Vary repeats: **-nrepeat #**
- ▶ Compare performance of CPU vs GPU

Execution Space

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



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

Changing the parallel execution space:

Custom

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

Default

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

Changing the parallel execution space:

Custom

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

Default

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

Requirements for enabling execution spaces:

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

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

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


Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```

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

```

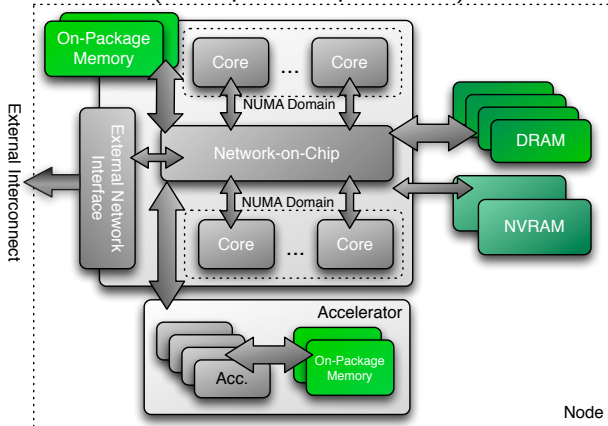
Lambda annotation with KOKKOS_LAMBDA macro

```

Kokkos::parallel_for("Label", numberOfIterations,
  KOKKOS_LAMBDA (const int64_t index) {...});
// Where Kokkos defines:
#define KOKKOS_LAMBDA [=] /* #if CPU-only */
#define KOKKOS_LAMBDA [=] __device__ __host__ /* #if CPU+Cuda */

```

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



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

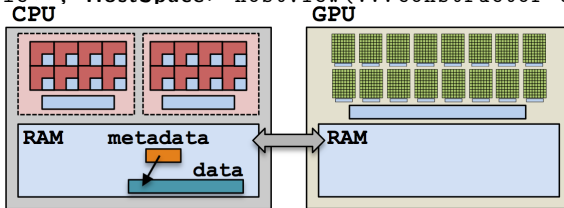
Every view stores its data in a **memory space** set at compile time.

- ▶ `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);
```

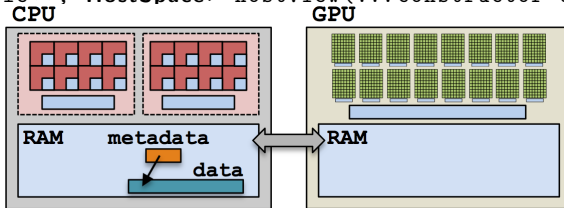

Example: HostSpace

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



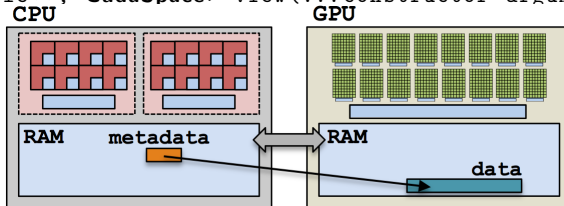
Example: HostSpace

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



Example: CudaSpace

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



Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 1: View lives in CudaSpace

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2: View lives in HostSpace

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

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

What's the solution?

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

Important concept: Mirrors

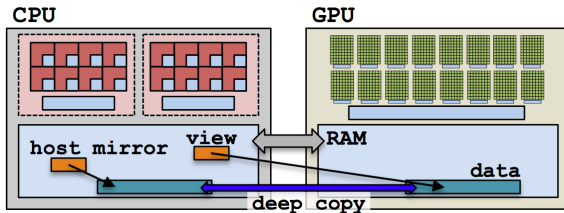
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

Important concept: Mirrors

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

Mirroring schematic

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



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?

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

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

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.

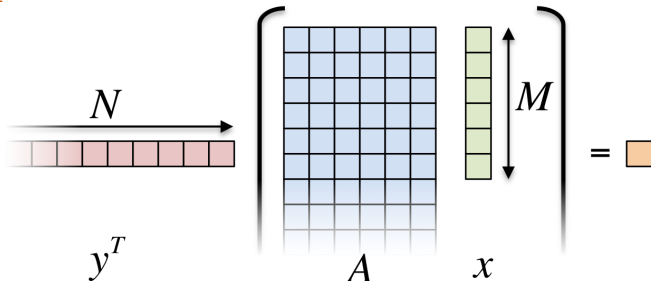
Managing memory access patterns for performance portability

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.

Example: inner product (0)

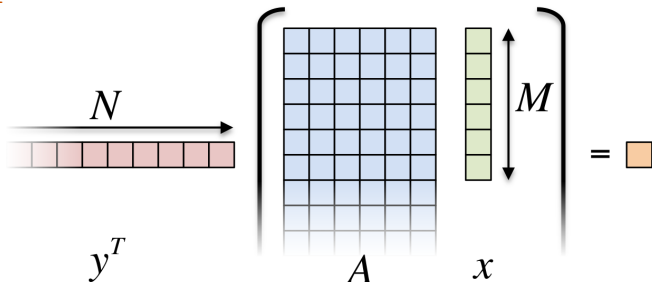
```
Kokkos::parallel_reduce("Label",  
  RangePolicy<ExecutionSpace>(0, N),  
  KOKKOS_LAMBDA (const size_t row, double & valueToUpdate) {  
    double thisRowsSum = 0;  
    for (size_t entry = 0; entry < M; ++entry) {  
      thisRowsSum += A(row, entry) * x(entry);  
    }  
    valueToUpdate += y(row)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



```

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);

```

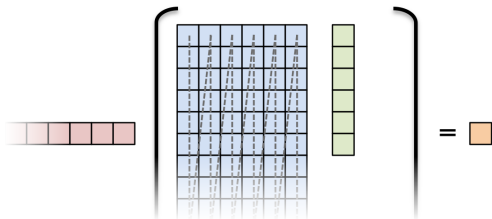


Driving question: How should A be laid out in memory?

Layout is the mapping of multi-index to memory:

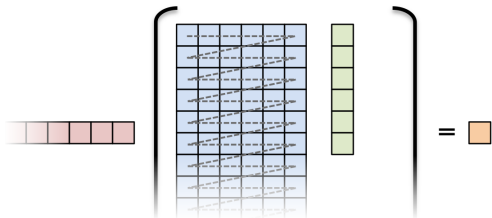
LayoutLeft

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

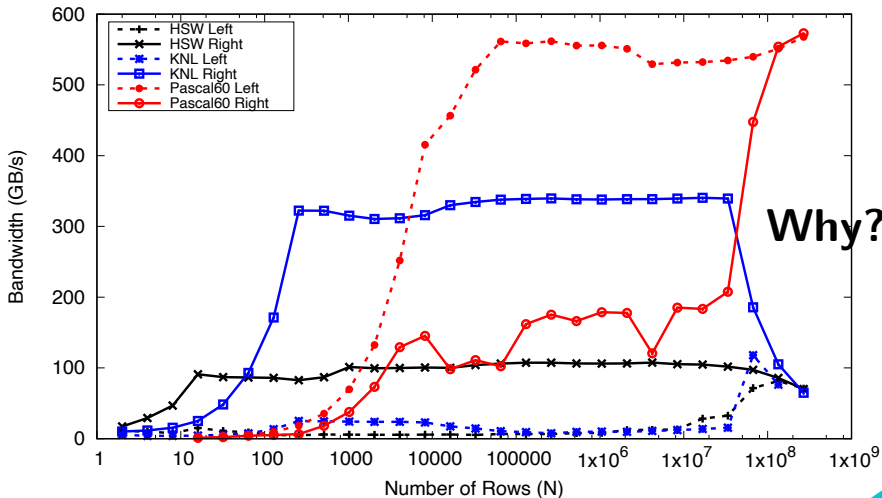
Things to try:

- ▶ 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

$\langle y | Ax \rangle$ 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 finish 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?

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)

Rule of Thumb

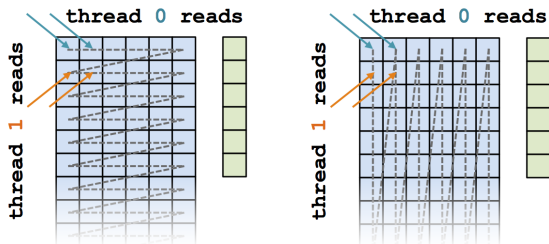
Kokkos index mapping and default layouts provide efficient access if **iteration indices** correspond to the **first index** of array.

Example:

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

Analysis: Kokkos architecture-dependent

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



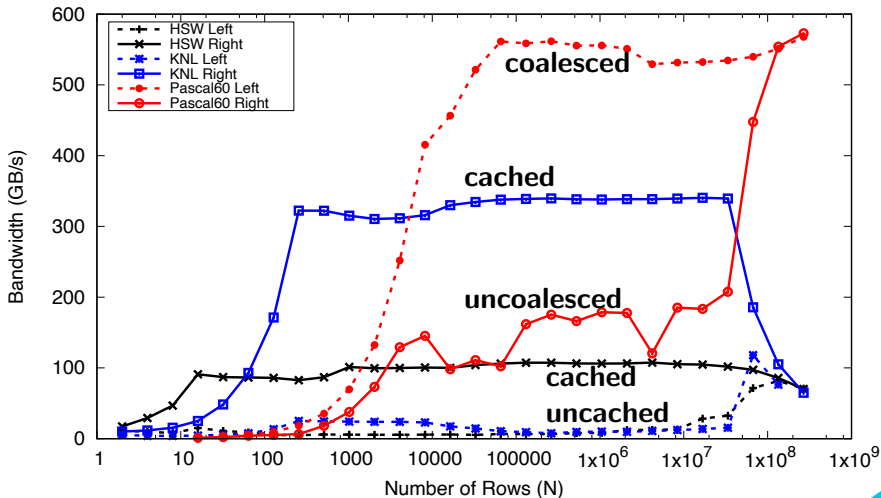
(a) OpenMP

(b) Cuda

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

<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.

The Kokkos Lectures

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

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

<https://kokkos.link/the-lectures>

Online Resources:

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

Kokkos EcoSystem:

- ▶ C++ Performance Portability Programming Model.
- ▶ The Kokkos Ecosystem provides capabilities needed for serious code development.
- ▶ Kokkos is supported by multiple National Laboratories with a sizeable dedicated team.

Data Parallelism:

- ▶ Simple things stay simple!
- ▶ You use **parallel patterns** and **execution policies** to execute **computational bodies**
- ▶ Simple parallel loops use the `parallel_for` pattern:

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

- ▶ Reductions combine contributions from loop iterations

```
int result;  
parallel_reduce("Label", N, [=] (int64_t i, int& lres) {  
    /* loop body */  
    lres += /* something */  
}, result);
```

Kokkos View

- ▶ Multi Dimensional Array.
- ▶ Compile and Runtime Dimensions.
- ▶ Reference counted like a `std::shared_ptr` to an array.

```
Kokkos::View<int*[5]> a("A", N);  
a(3,2) = 7;
```

Execution Spaces

- ▶ Parallel operations execute in a specified **Execution Space**
- ▶ Can be controlled via template argument to **Execution Policy**
- ▶ If no Execution Space is provided use
`DefaultExecutionSpace`

```
// Equivalent:  
parallel_for("L", N, functor);  
parallel_for("L",  
    RangePolicy<DefaultExecutionSpace>(0, N), functor);
```

Memory Spaces

- ▶ Kokkos Views store data in **Memory Spaces**.
- ▶ Provided as template parameter.
- ▶ If no Memory Space is given, use `Kokkos::DefaultExecutionSpace::memory_space`.
- ▶ `deep_copy` is used to transfer data: no hidden memory copies by Kokkos.

```
View<int*, CudaSpace> a("A", M);  
// View in host memory to load from file  
auto h_a = create_mirror_view(a);  
load_from_file(h_a);  
// Copy  
deep_copy(a, h_a);
```

Layouts

- ▶ Kokkos Views use an index mapping to memory determined by a **Layout**.
- ▶ Provided as template parameter.
- ▶ If no **Layout** is given, derived from the execution space associated with the memory space.
- ▶ Defaults are good if you parallelize over left most index!

```
View<int**, LayoutLeft> a("A", N, M);
View<int**, LayoutRight> b("B", N, M);

parallel_for("Fill", N, KOKKOS_LAMBDA(int i) {
    for(int j = 0; j < M; j++) {
        a(i,j) = i * 1000 + j; // coalesced
        b(i,j) = i * 1000 + j; // cached
    }
});
```

Hierarchical Parallelism

- ▶ How to leverage more parallelism through nested loops.
- ▶ The concept of Thread-Teams and Vectorlength.

Kokkos Tools

- ▶ Debugging
- ▶ Profiling