The Kokkos Lectures

The Fundamentals: A Condensed Short Tutorial

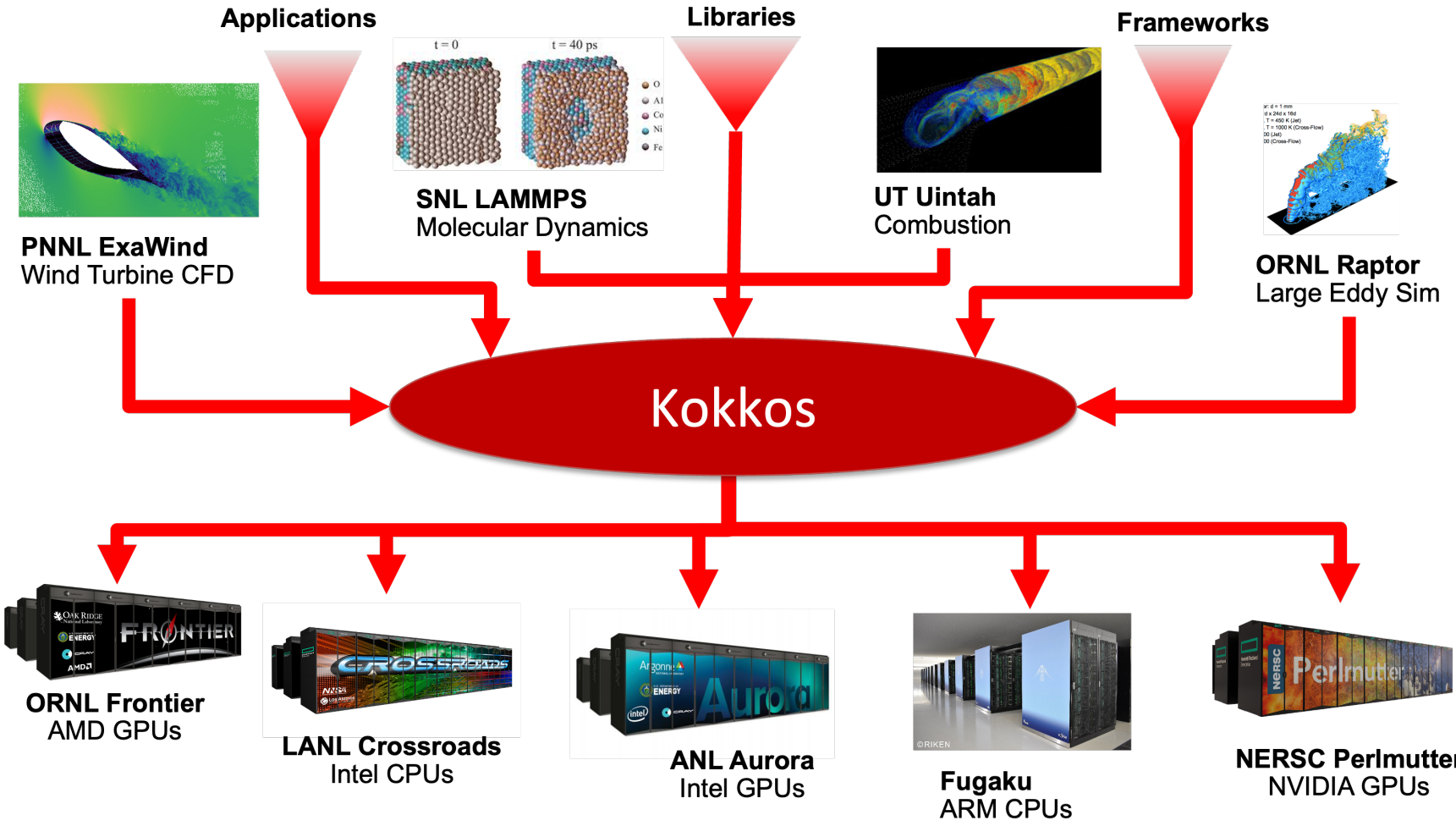
Jakob Bludau

June 24th 2024

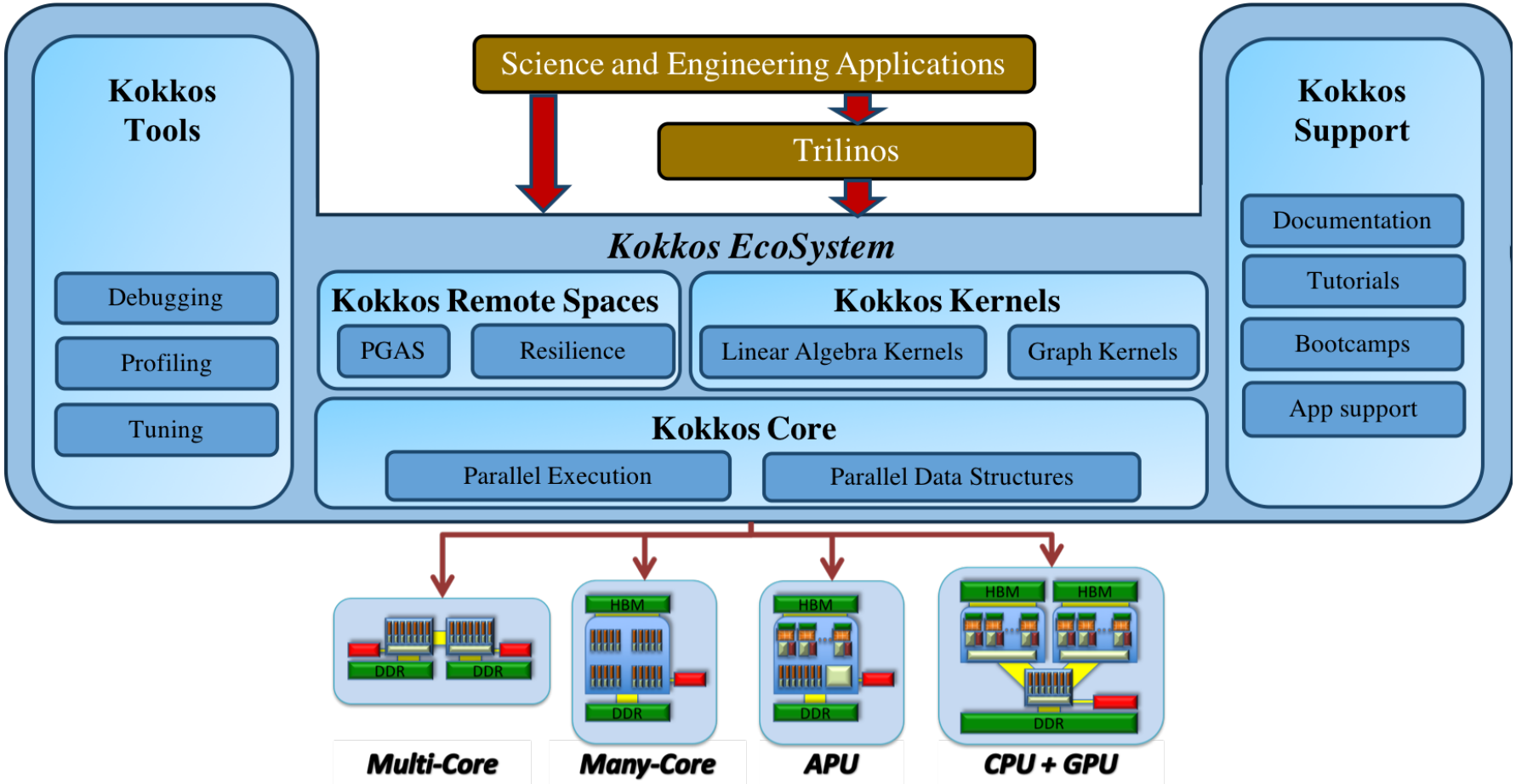
A Condensed Short Tutorial

This lecture covers fundamental concepts of Kokkos with Hands-On exercises.
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


Applications
Libraries
Frameworks
PNNL ExaWind
Wind Turbine CFD
t = 0
t = 40 ps
SNL LAMMPS
Molecular Dynamics
UT Uintah
Combustion
ORNL Raptor
Large Eddy Sim
Kokkos
ORNL Frontier
AMD GPUs
LANL Crossroads
Intel CPUs
ANL Aurora
Intel GPUs
Fugaku
ARM CPUs
NERSC Perlmutter
NVIDIA GPUs

The Kokkos EcoSystem




kokkos


Sandia
National
Laboratories
CSCS
OAK RIDGE
National Laboratory
Argonne
NATIONAL LABORATORY
Los Alamos
NATIONAL LABORATORY
EST.1943
BERKELEY LAB
cea
NGA
NexGen Analytics

Online Resources:

- https://kokkos.org:
  - Programming guide and API references.
  - All repositories in the ecosystem.
  - Links to tutorials.
  - etc.
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Everyone can invite others.

Some (good) reasons:

- It is used in Trilinos
- Get performance out of any accelerator without starting over
- It helps you avoid repeating the common mistakes when porting to an accelerator
- It is descriptive (mostly)

To describe to Kokkos what you want there are a few building blocks:

- Parallel patterns (How?)
- Functors (What?)
- Views (Which data?)
- Spaces (Where?)
- Policies (Which indices?)
- ...

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;
  DataType _atomData;

  AtomForceFunctor(ForceType atomForces, DataType 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 ⇒ **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 ⇒ **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 ⇒ **C++11 Lambdas** are concise

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

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

Warning: Lambda capture and C++ containers

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

How does this compare to OpenMP?

Serial

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

OpenMP

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

Kokkos

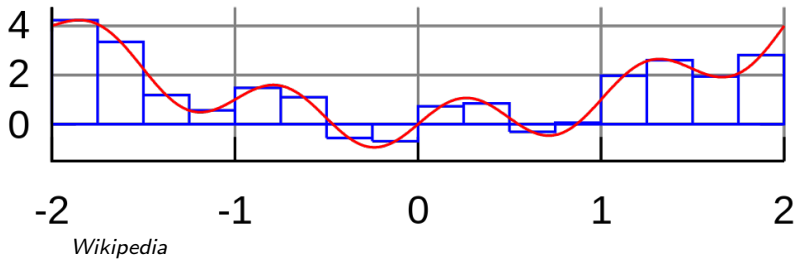
```
parallel_for(N, [=] (const int64_t i) {
  /* loop body */
});
```

Important concept

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

Riemann-sum-style numerical integration:

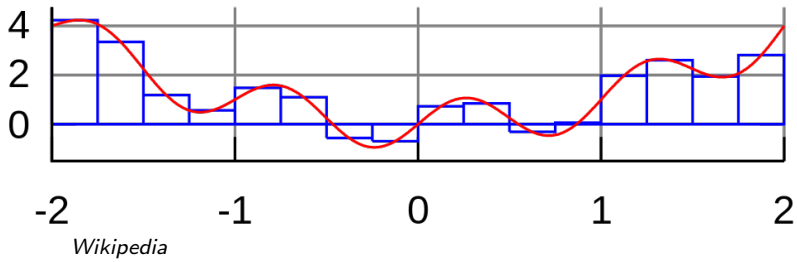
$$y = \int_{lower}^{upper} function(x)\, dx$$



Wikipedia

Riemann-sum-style numerical integration:

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

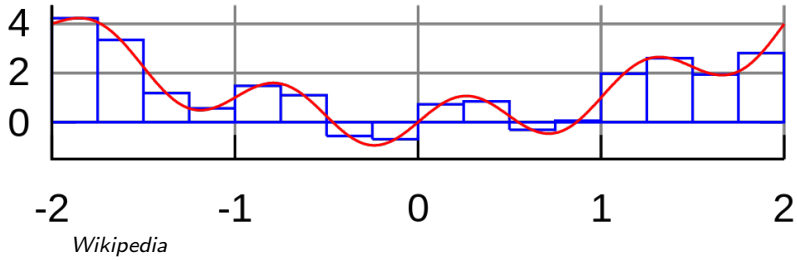


Wikipedia

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

Riemann-sum-style numerical integration:

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

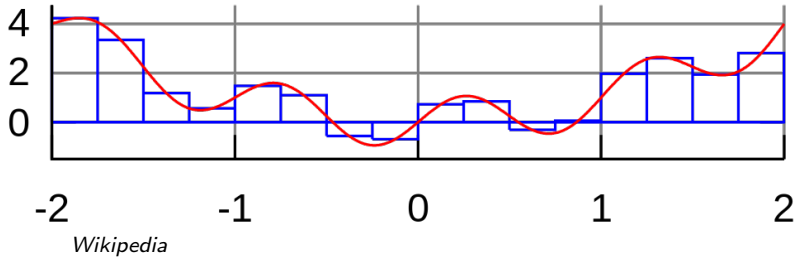



```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  const double x =
    lower + (i/numberOfIntervals) * (upper - lower);
  const double thisIntervalsContribution = function(x);
  totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

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

Riemann-sum-style numerical integration:

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



Pattern?

Policy?

Body?

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

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

An (incorrect) attempt:

```
double totalIntegral = 0;
Kokkos::parallel_for(numberOfIntervals,
  [=] (const int64_t index) {
    const double x =
      lower + (index/numberOfIntervals) * (upper - lower);
    totalIntegral += function(x);},
  );
totalIntegral *= dx;
```

First problem: compiler error; cannot increment `totalIntegral`
(lambdas capture by value and are treated as const!)

An (incorrect) solution to the (incorrect) attempt:

```
double totalIntegral = 0;
double * totalIntegralPointer = &totalIntegral;
Kokkos::parallel_for(numberOfIntervals,
  [=] (const int64_t index) {
    const double x =
      lower + (index/numberOfIntervals) * (upper - lower);
    *totalIntegralPointer += function(x);},
  );
totalIntegral *= dx;
```

An (incorrect) solution to the (incorrect) attempt:

```
double totalIntegral = 0;
double * totalIntegralPointer = &totalIntegral;
Kokkos::parallel_for(numberOfIntervals,
  [=] (const int64_t index) {
    const double x =
      lower + (index/numberOfIntervals) * (upper - lower);
    *totalIntegralPointer += function(x);},
  );
totalIntegral *= dx;
```

Second problem: race condition

step	thread 0	thread 1
0	load	
1	increment	load
2	write	increment
3		write

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!

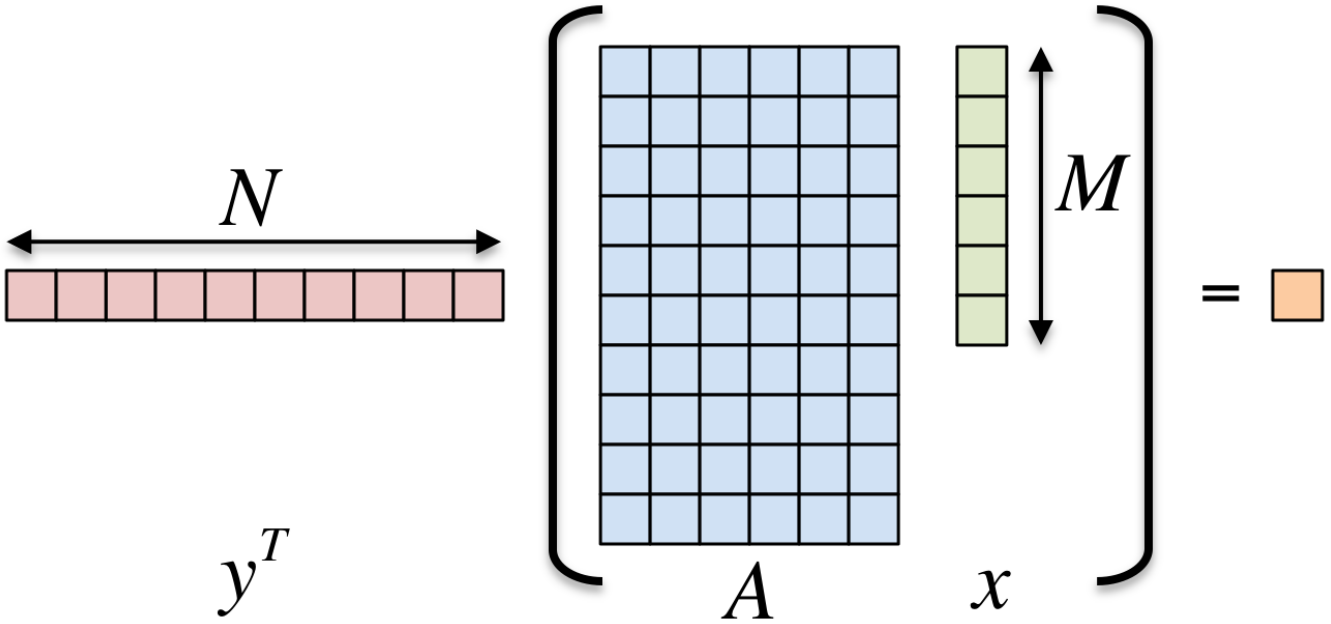
Giving unique names to each kernel is immensely helpful for debugging and profiling. You will regret it if you don't!

- 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 $< y, A * x >$



Details:

- y is Nx1, A is NxM, x is Mx1
- We'll use this exercise throughout the tutorial

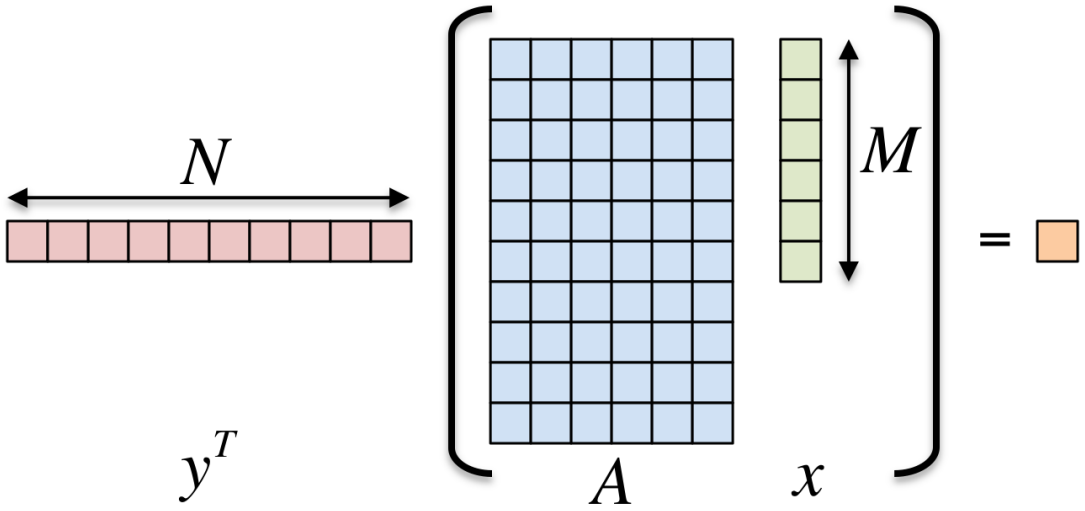
The **first step** in using Kokkos is to include, initialize, and finalize:

```
#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:

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

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



Details:

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

Compiling for CPU

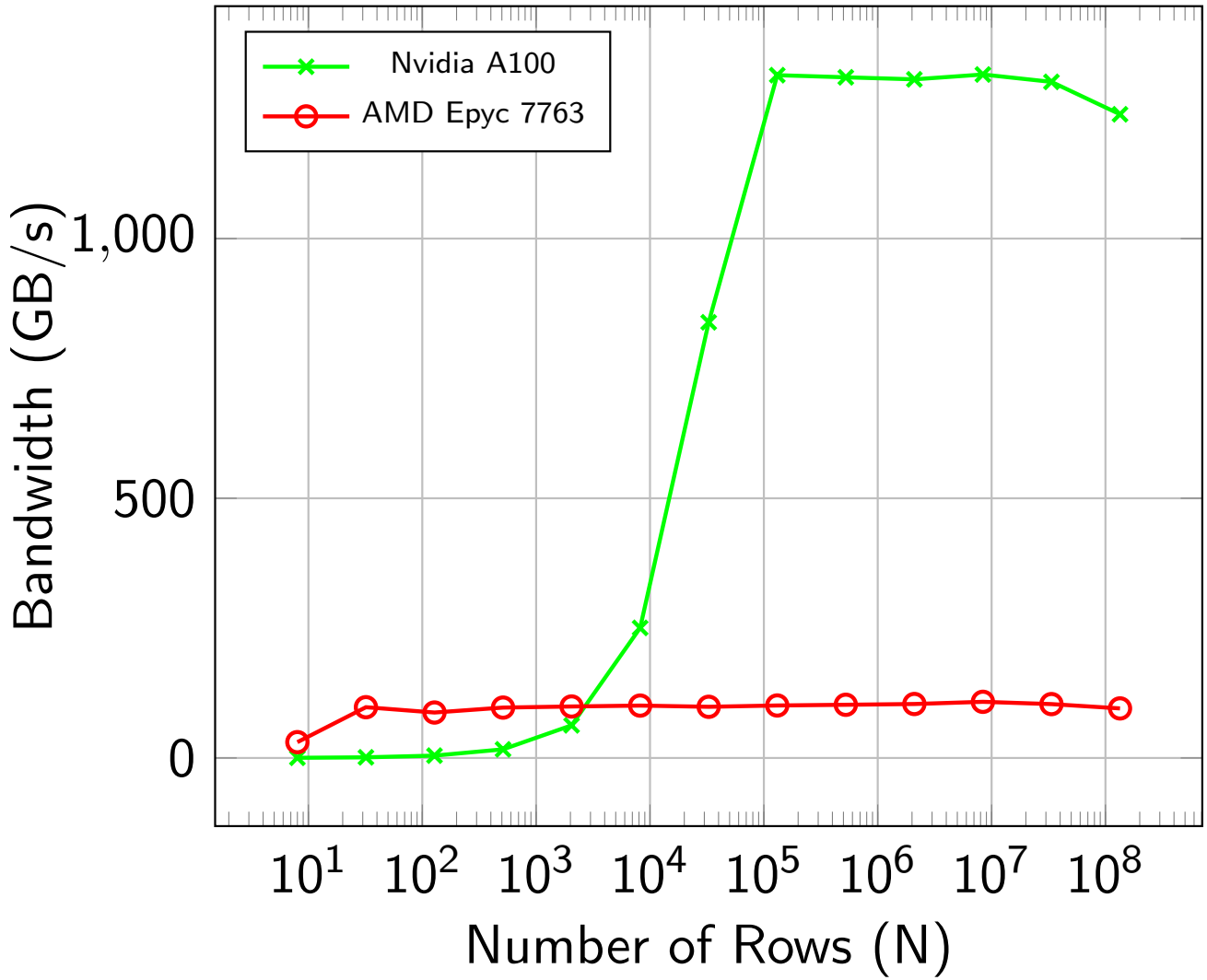
```
cmake -B build -DKokkos_ENABLE_OPENMP=ON \
    -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

Running on CPU with OpenMP backend

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

Things to try:

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


Nvidia A100
AMD Epyc 7763
Bandwidth (GB/s)
1,000
500
0
10^1 10^2 10^3 10^4 10^5 10^6 10^7 10^8
Number of Rows (N)

- **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.

C++ Performance Portability - A Decade of Lessons Learned
https://www.youtube.com/watch?v=jNGGKFkt4lA

- Loops are par-unseq.→*Kokkos::parallel_∗*
- Data structures need reference semantics.

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).

$\Rightarrow$ **Views**

View abstraction

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

High-level example of Views for daxpy using lambda:

```
View<double*, ...> x(...), y(...);
...populate x, y...

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

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 with Views

- Location: `Exercises/02/Begin/`
- Assignment: Change data storage from arrays to Views.

```
# CPU-only using OpenMP
cmake -B build-openmp -DKokkos_ENABLE_OPENMP=ON
cmake --build build-openmp
# Run exercise
./build-openmp/02_Exercise -S 26
# Note the warnings, set appropriate environment variables
```

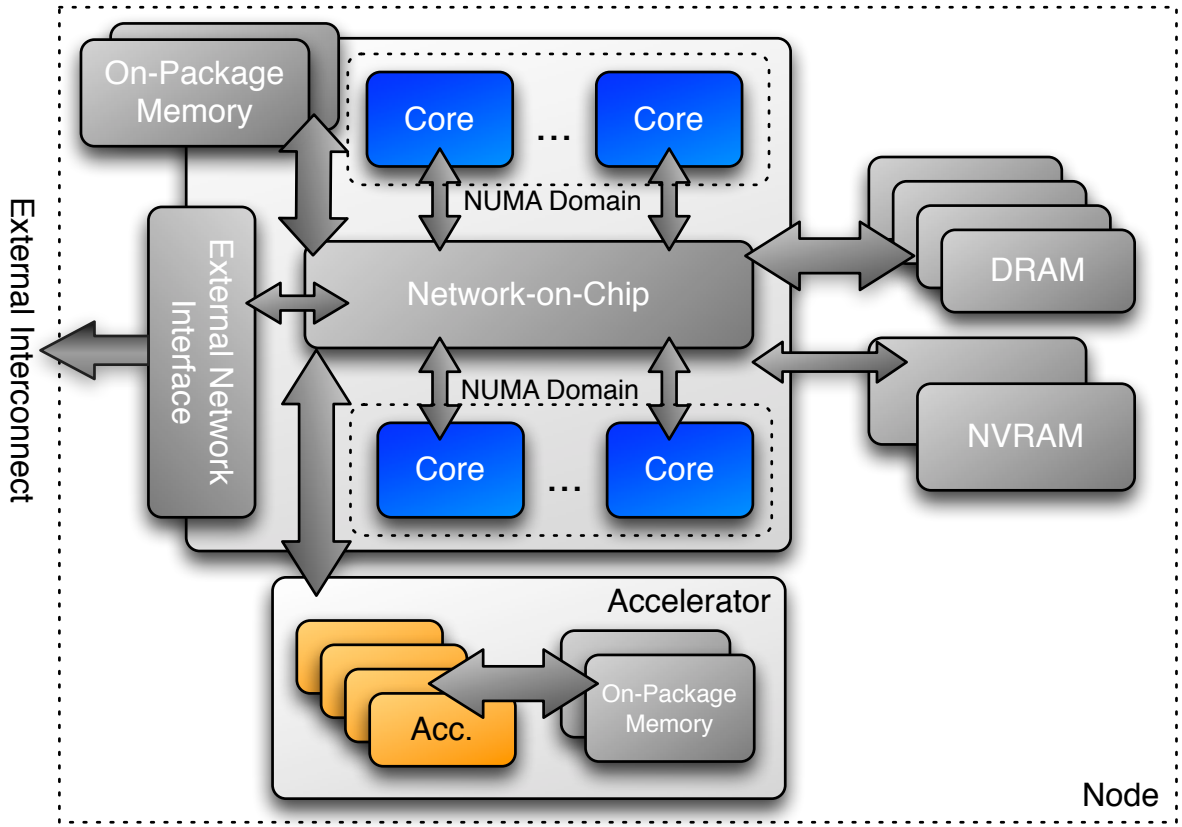
- Vary problem size: -**S #**
- Vary number of rows: -**N #**
- Vary repeats: -**nrepeat #**

C++ Performance Portability - A Decade of Lessons Learned
https://www.youtube.com/watch?v=jNGGKFkt4lA

- Loops are par-unseq.→*Kokkos::parallel_∗*
- Data structures need reference semantics.→*Kokkos::View*

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

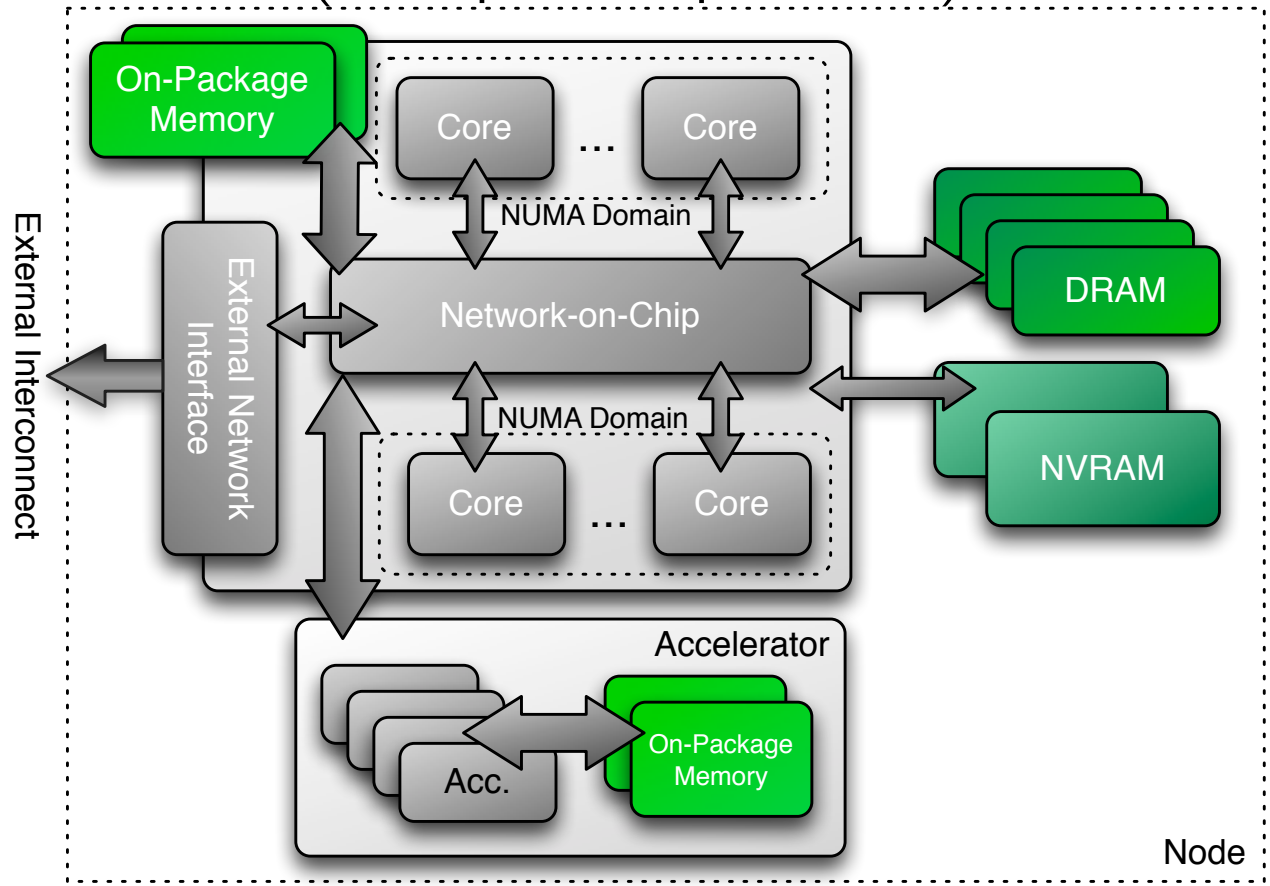
```
struct ParallelFunctor {
  KOKKOS_INLINE_FUNCTION
  double helperFunction(const int64_t s) const {...}
  KOKKOS_INLINE_FUNCTION
  void operator()(const int64_t index) const {
    helperFunction(index);
  }
}
// Where kokkos defines:
#define KOKKOS_INLINE_FUNCTION inline                          /* #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***,`*Memory***Space**`> data(...);`

Important concept: Memory spaces

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

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

  `HostSpace`, `CudaSpace`, `CudaUVMSpace`, `...` more

  Portable: `SharedSpace`, `SharedHostPinnedSpace`

Important concept: Memory spaces

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

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

  `HostSpace`, `CudaSpace`, `CudaUVMSpace`, ... more

  Portable: `SharedSpace`, `SharedHostPinnedSpace`
- Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space

Important concept: Memory spaces

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

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

  `HostSpace`, `CudaSpace`, `CudaUVMSpace`, ... more

  Portable: `SharedSpace`, `SharedHostPinnedSpace`
- Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space
- If no `Space` is provided, the view's data resides in the **default memory space** of the **default execution space**.

Important concept: Memory spaces

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

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

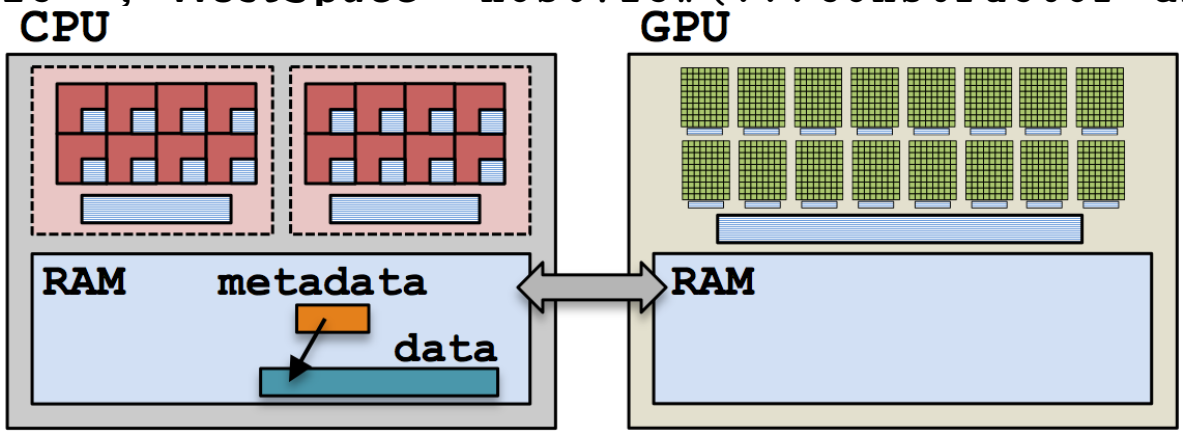
  `HostSpace`, `CudaSpace`, `CudaUVMSpace`, ... more

  Portable: `SharedSpace`, `SharedHostPinnedSpace`
- 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);
```

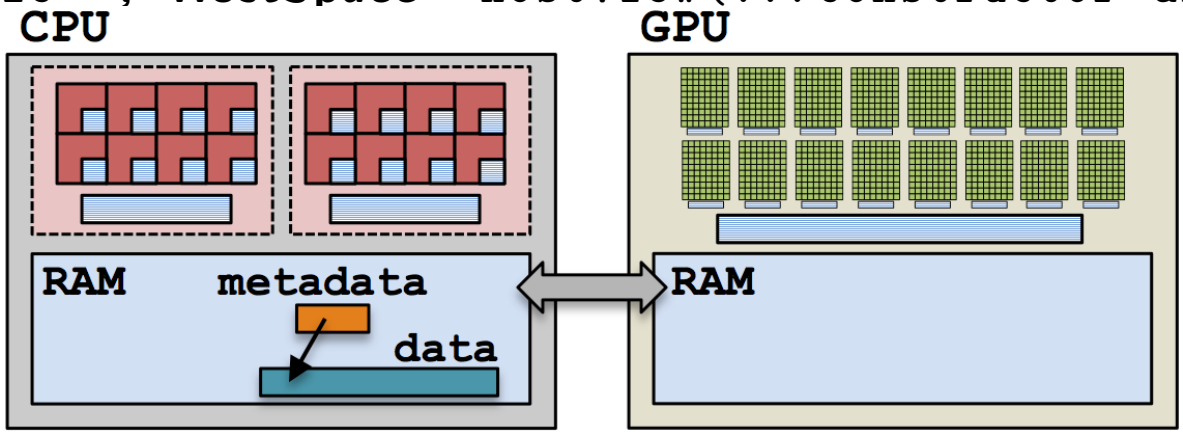
Example: HostSpace

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



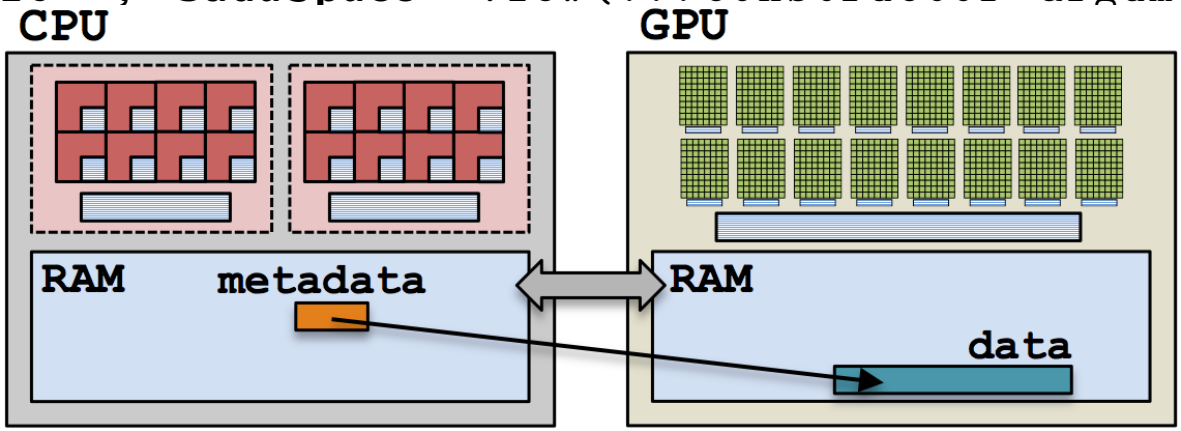
Example: HostSpace

```
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?

- SharedSpace
- SharedHostPinnedSpace
- Mirroring

Important concept: Mirrors

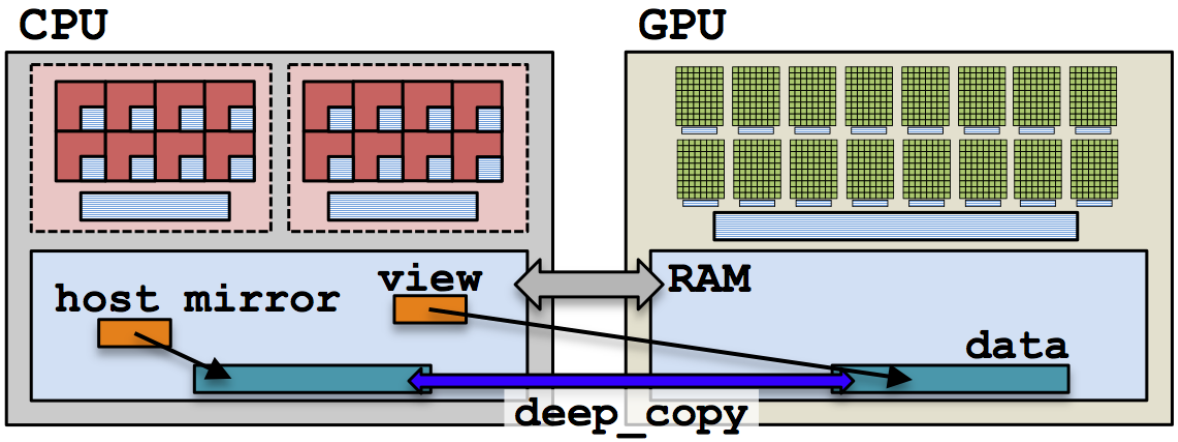
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

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
cmake -B build-openmp -DKokkos_ENABLE_OPENMP=ON
cmake --build build-openmp
# Compile for GPU
cmake -B build-cuda -DKokkos_ENABLE_CUDA=ON
cmake --build build-cuda
# Run on CPU
./build-openmp/03_Exercise -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.

C++ Performance Portability - A Decade of Lessons Learned

https://www.youtube.com/watch?v=jNGGKFkt4lA

- Loops are par-unseq.→*Kokkos::parallel_∗*
- Data structures need reference semantics.→*Kokkos::View*
- Functions might need to be annotated.→*KOKKOS_FUNCTION,LAMBDA,...*
- Execution and memory resources need to know about each other.→*Kokkos::ExecutionSpace,MemorySpace*

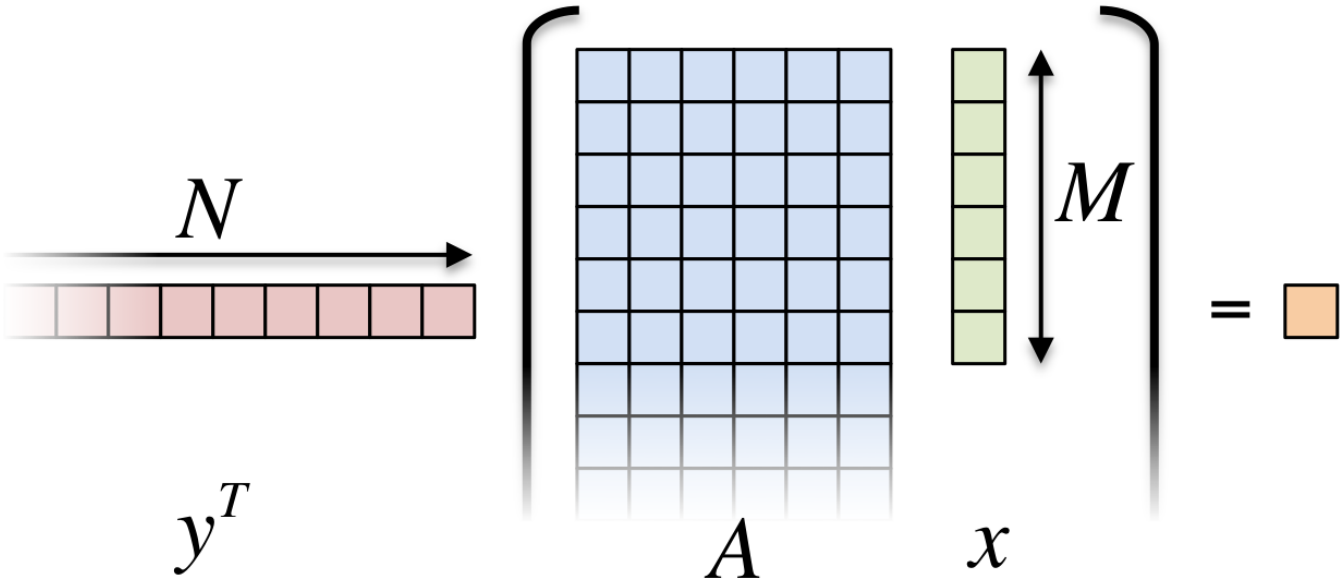
Managing memory access patterns for performance portability

Learning objectives:

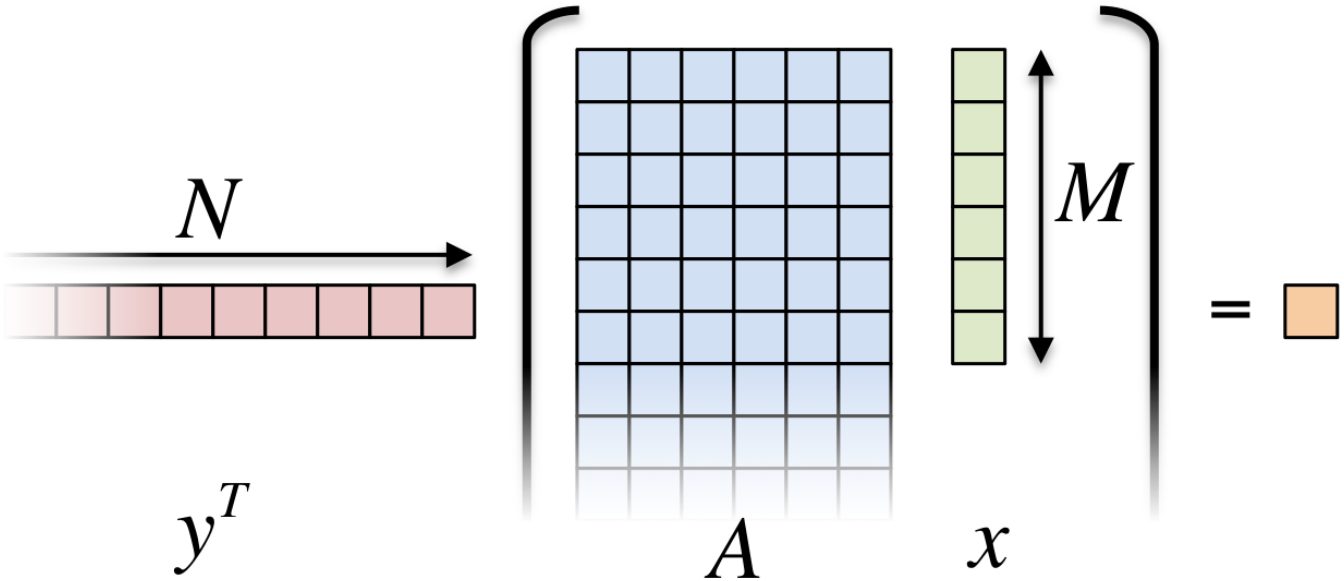
- How the `View`'s `Layout` parameter controls data layout.
- How memory access patterns result from Kokkos mapping parallel work indices **and** layout of multidimensional array data
- Why memory access patterns and layouts have such a performance impact (caching and coalescing).
- See a concrete example 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);
```

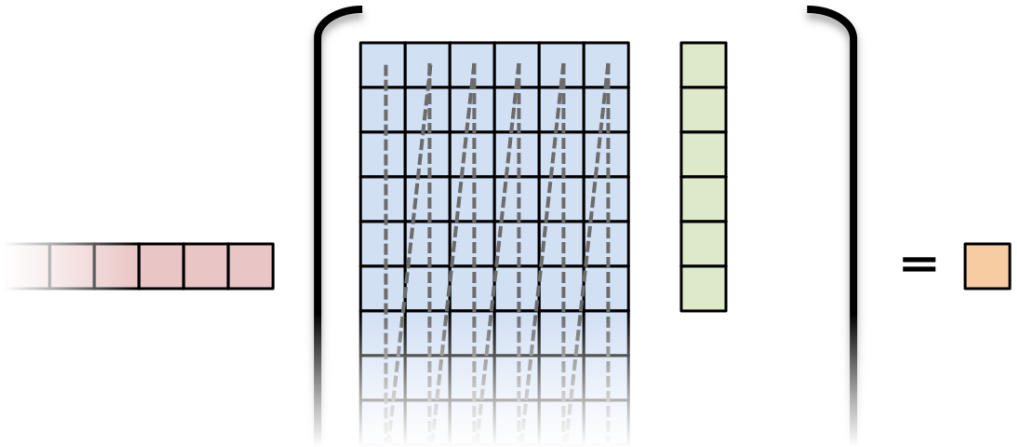


Driving question: How should A be laid out in memory?

Layout is the mapping of multi-index to memory:

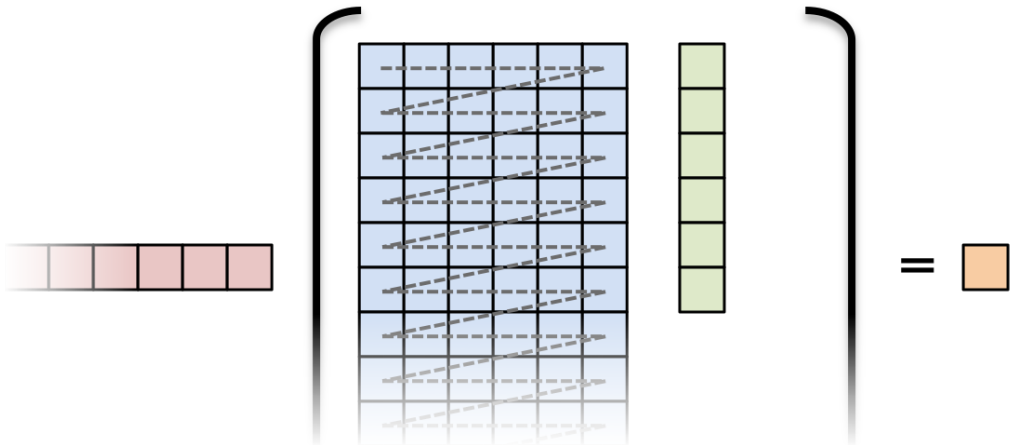
LayoutLeft

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(...);
```

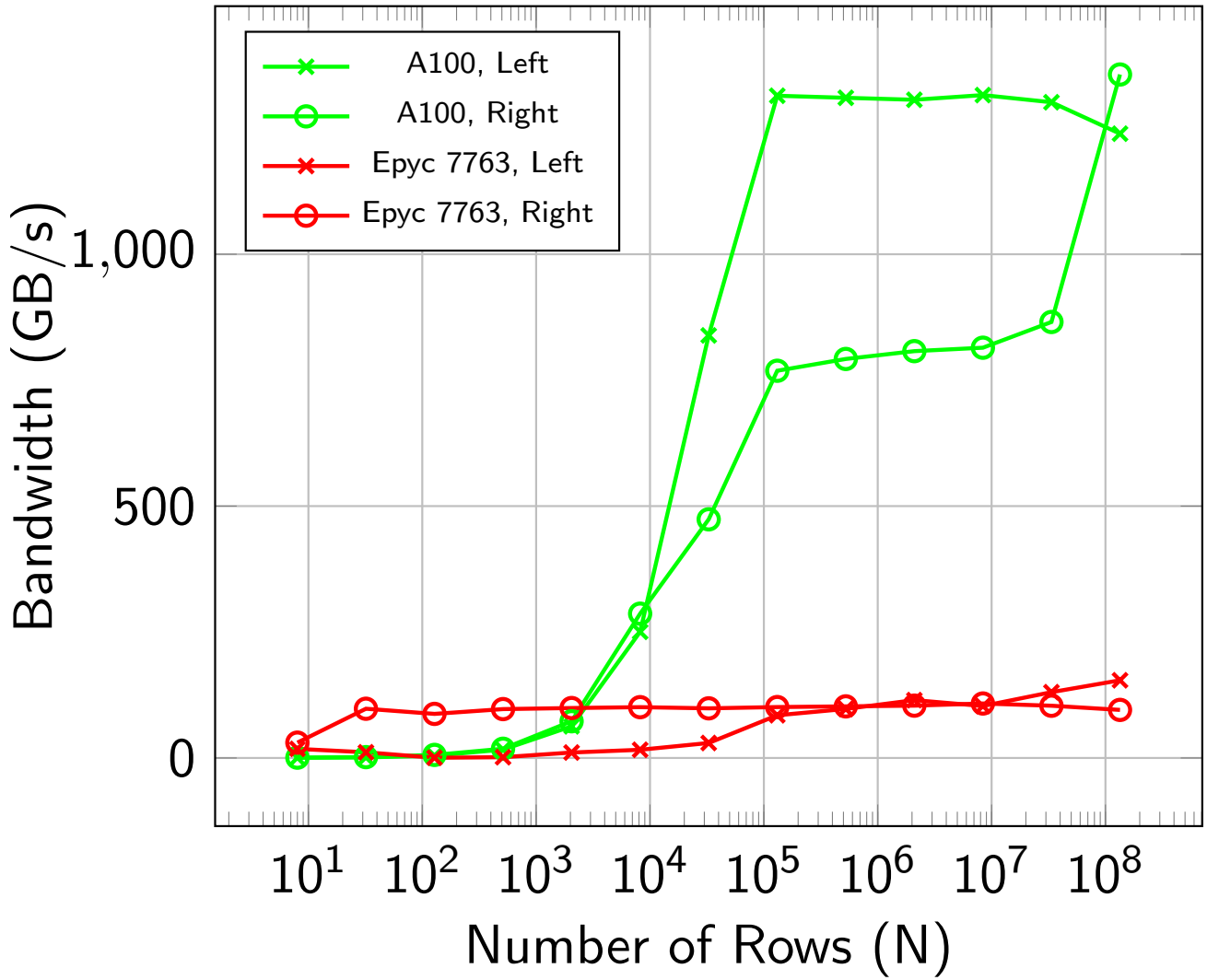
- Most-common layouts are `LayoutLeft` and `LayoutRight`.
  - `LayoutLeft`: left-most index is stride 1.
  - `LayoutRight`: right-most index is stride 1.
- If no layout specified, default for that memory space is used.
  - `LayoutLeft` for `CudaSpace`, `LayoutRight` for `HostSpace`.
- Layouts are extensible: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

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 ...)
- Check what happens if `MemSpace` and `ExecSpace` do not match.


A100, Left
A100, Right
Epyc 7763, Left
Epyc 7763, Right
Bandwidth (GB/s)
1,000
500
0
10^1 10^2 10^3 10^4 10^5 10^6 10^7 10^8
Number of Rows (N)

Why?

Thread independence:

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

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

Thread independence:

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

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

- **CPU** threads are independent.
  - i.e., threads may execute at any rate.

Thread independence:

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

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

- **CPU** threads are independent.
  - i.e., threads may execute at any rate.
- **GPU** threads execute synchronized.
  - i.e., threads in groups can/must execute instructions together.

Thread independence:

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

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

- **CPU** threads are independent.
  - i.e., threads may execute at any rate.
- **GPU** threads execute synchronized.
  - i.e., threads in groups can/must execute instructions together.

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

Thread independence:

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

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

- **CPU** threads are independent.
  - i.e., threads may execute at any rate.
- **GPU** threads execute synchronized.
  - i.e., threads in groups can/must execute instructions together.

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

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

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

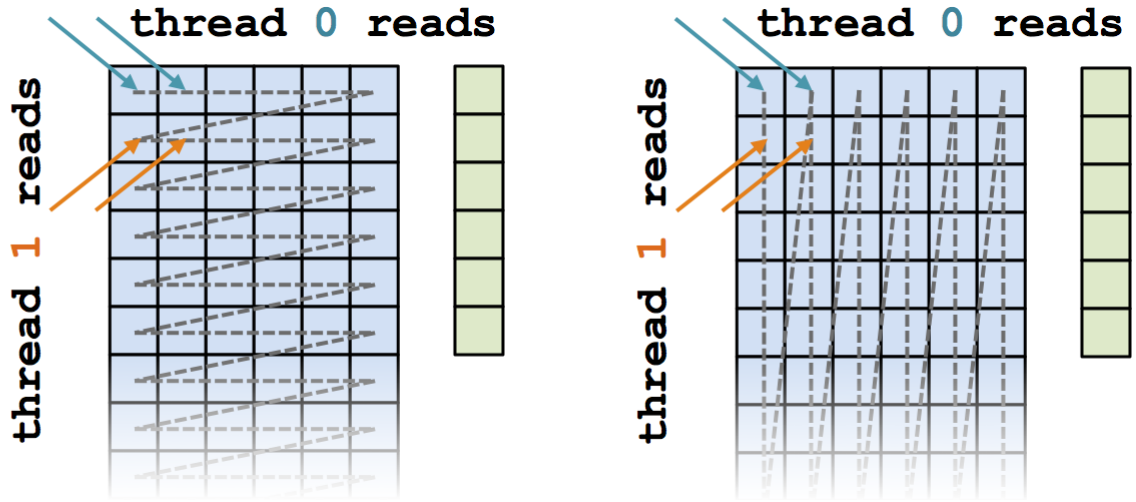
Kokkos index mapping and default layouts provide efficient access if **iteration indices** correspond to the **first index** of array.

Example:

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

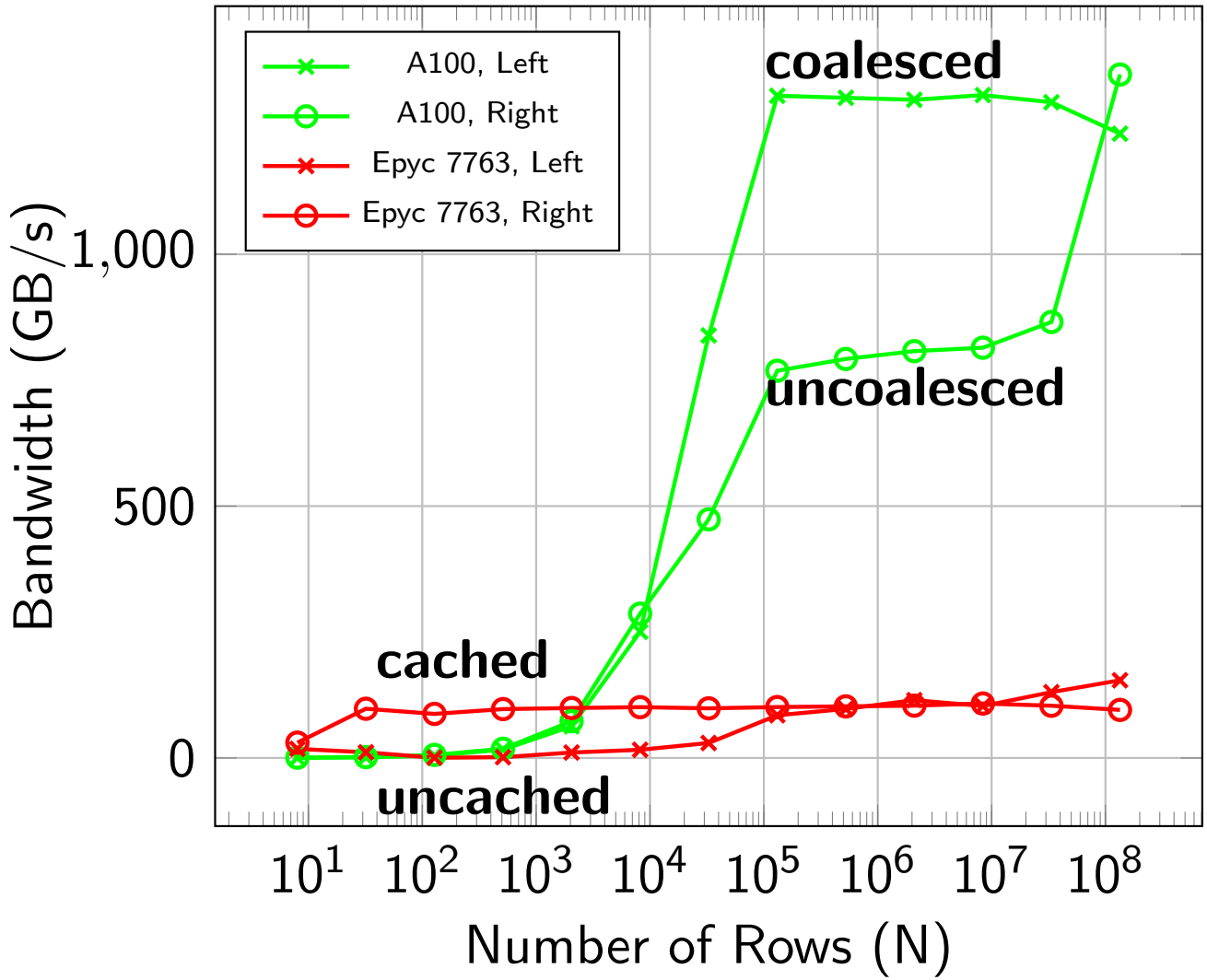
Analysis: Kokkos architecture-dependent

```
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)


A100, Left
A100, Right
Epyc 7763, Left
Epyc 7763, Right
coalesced
uncoalesced
cached
uncached
Bandwidth (GB/s)
1,000
500
0
10^1 10^2 10^3 10^4 10^5 10^6 10^7 10^8
Number of Rows (N)

Memory Access Pattern Summary

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

C++ Performance Portability - A Decade of Lessons Learned

https://www.youtube.com/watch?v=jNGGKFkt4lA

- Loops are par-unseq.→*Kokkos::parallel_∗*
- Data structures need reference semantics.→*Kokkos::View*
- Functions might need to be annotated.→*KOKKOS_FUNCTION,LAMBDA,...*
- Execution and memory resources need to know about each other.→*Kokkos::ExecutionSpace,MemorySpace*
- Data layouts need to be adaptable for portable performance.→*Kokkos::Layouts*

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.
- Non-Sum reductions / multiple reductions.

This was a short introduction

Didn't cover many things:

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

This was a short introduction

Didn't cover many things:

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

This was a short introduction

Didn't cover many things:

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

This was a short introduction

Didn't cover many things:

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

This was a short introduction

Didn't cover many things:

- Full BuildSystem integration.
- Non-Sum reductions / multiple reductions.
- Multidimensional loops.
- Advanced data structures.
- Subviews.
- 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.
- Non-Sum reductions / multiple reductions.
- Multidimensional loops.
- Advanced data structures.
- Subviews.
- 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.
- Non-Sum reductions / multiple reductions.
- Multidimensional loops.
- Advanced data structures.
- Subviews.
- 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.
- Non-Sum reductions / multiple reductions.
- Multidimensional loops.
- Advanced data structures.
- Subviews.
- Atomic operations and Scatter Contribute patterns.
- Team Scratch memory (GPU shared memory).
- SIMD vectorization.
- MPI and PGAS integration.
- Tools for Profiling, Debugging and Tuning.

This was a short introduction

Didn't cover many things:

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

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

C++ Performance Portability - A Decade of Lessons Learned

https://www.youtube.com/watch?v=jNGGKFkt4lA

- Loops are par-unseq.→*Kokkos::parallel_∗*
- Data structures need reference semantics.→*Kokkos::View*
- Functions might need to be annotated.→*KOKKOS_FUNCTION,LAMBDA,...*
- Execution and memory resources need to know about each other.→*Kokkos::ExecutionSpace,MemorySpace*
- Data layouts need to be adaptable for portable performance.→*Kokkos::Layouts*
- Use specialized hardware features.→*Kokkos::SIMD,Atomics,ScratchSpace,...*
- Nested parallel patterns for complex cases.→*Kokkos::NestedPolies*

Online Resources:

- https://kokkos.org:
  - Programming guide and API references.
  - All repositories in the ecosystem.
  - Links to tutorials.
  - etc.
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Everyone can invite others.

DualView

Learning objectives:

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

Motivation and Value-added

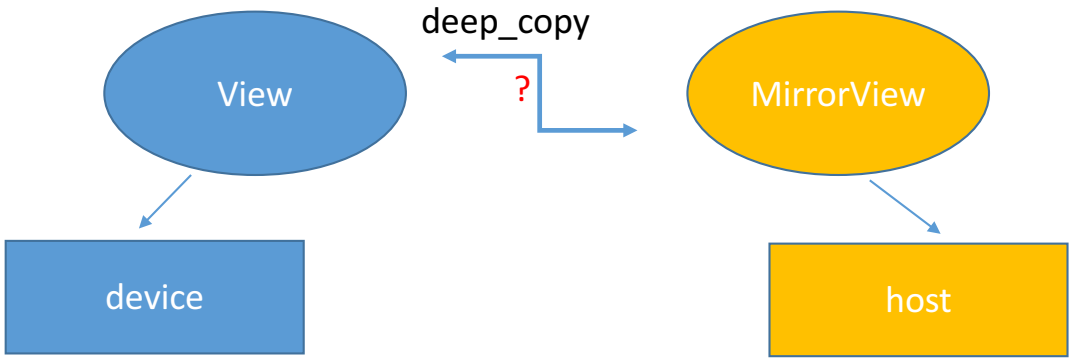
- DualView was designed to help transition codes to Kokkos.

Motivation and Value-added

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

Motivation and Value-added

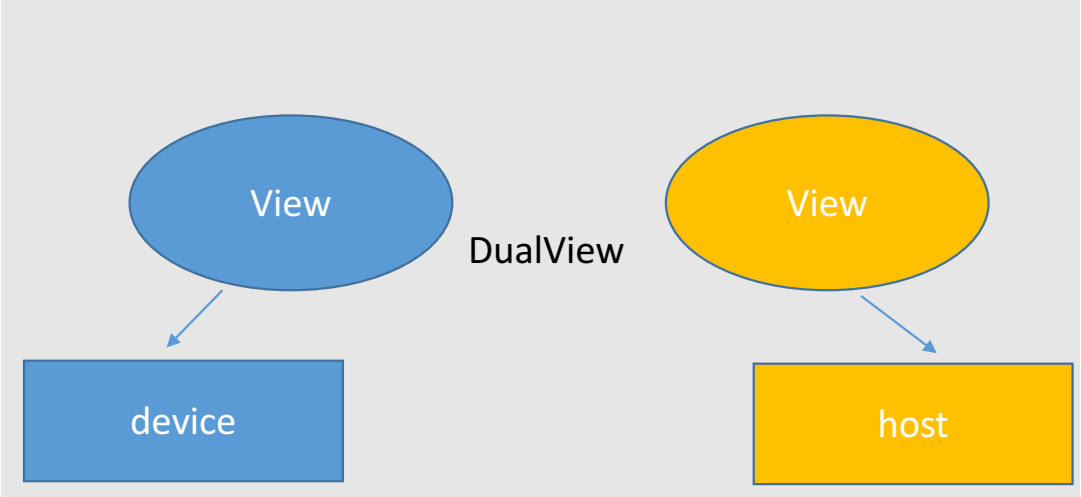
- DualView was designed to help transition codes to Kokkos.
- DualView simplifies the task of managing data movement between memory spaces, e.g., host and device.
- When converting a typical app to use Kokkos, there is usually no holistic view of such data transfers.



Without DualView, could use MirrorViews, but

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

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



There is no automatic tracking of data freshness:

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

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

- Data members for the two views

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

- Retrieve data members

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

- Mark data as modified

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

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

- Sync data in a direction if not in sync

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

- Check sync status

```
bool need_sync_host();
bool need_sync_device();
```

DualView has templated functions for generic use in templated code

- Retrieve data members

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

- Mark data as modified

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

- Sync data in a direction if not in sync

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

- Check sync status

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

DualView Example

```
class Foo {
DualView<...> data;
void run_a() {
  data.sync_device(); data.modify_device();
  auto d_data = data.view_device();
  parallel_for(N, KOKKOS_LAMBDA(int i) { d_data(i)+=/*mod d_d*/});
}
void run_b() {
  data.sync_host();
  auto h_data = data.view_host();
  for(int i=0; i<N; i++) { h_data(i) += /* modify h_data */ });
  data.modify_host();
}
void run_c() {
  data.sync_device();
  auto d_data = data.view_device();
  parallel_for(N, KOKKOS_LAMBDA(int i) { /* read d_data */ });
}
void do_operations(bool a, bool b, bool c) {
  if(a) run_a();
  if(b) run_b();
  if(c) run_c();
}
};
```

Details:

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

```
# Compile for CPU
cmake -B build-openmp -DKokkos_ENABLE_OPENMP=ON
cmake --build build-openmp
# Compile for GPU
cmake -B build-cuda -DKokkos_ENABLE_CUDA=ON
cmake --build build-cuda
# Run
./build-openmp/03_Exercise -S 26
./build-cuda/03_Exercise -S 26
```