Kokkos: Performance Portability

Jonathan Rouzaud-Cornabas

INSA-Lyon – LIRIS – Inria Beagle

The Kokkos Lectures

Module 1: Introduction, Building and Parallel Dispatch

July 17, 2020

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2020-7263 PE

Kokkos is C++ Performance Portability

- Write a *single source* implementation using C++
- Use a *descriptive* Programming Model
- Compile for GPUs and CPUs

Kokkos is Ready for Use

- Well established project since 2012
- Major buy-in by DOE National Labs
- Well over 100 projects with over 500 developers use Kokkos
- Dedicated developer staff at 5 National Labs
- Robust support for software stacks: GCC 5+, Clang 4+, NVCC 9+, ROCM 3.5, XL16

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/LectureSeries:
  - Find these slides
- 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. Email: crtrott@sandia.gov

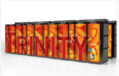
Introduction

Learning objectives:

- Why do we need Kokkos
- The Kokkos EcoSystem
- The Kokkos Team

The HPC Hardware Landscape

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

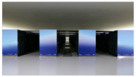


LANL/SNL Trinity
Intel Haswell / Intel KNL
OpenMP 3

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

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

SNL Astra
ARM CPUs
OpenMP 3



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

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

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

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

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

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

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

(e) OpenMP 5 by Intel.

Industry Estimate

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

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

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

Industry Estimate

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

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

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

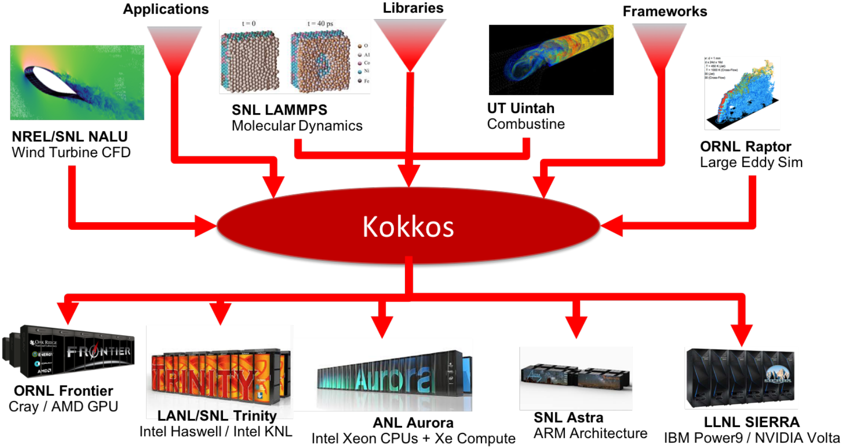
Software Cost Switching Vendors

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

What is Kokkos?

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

Kokkos at the Center



Applications

Libraries

Frameworks

NREL/SNL NALU
Wind Turbine CFD

SNL LAMMPS
Molecular Dynamics

UT Uintah
Combustine

ORNL Raptor
Large Eddy Sim

Kokkos

ORNL Frontier
Cray / AMD GPU

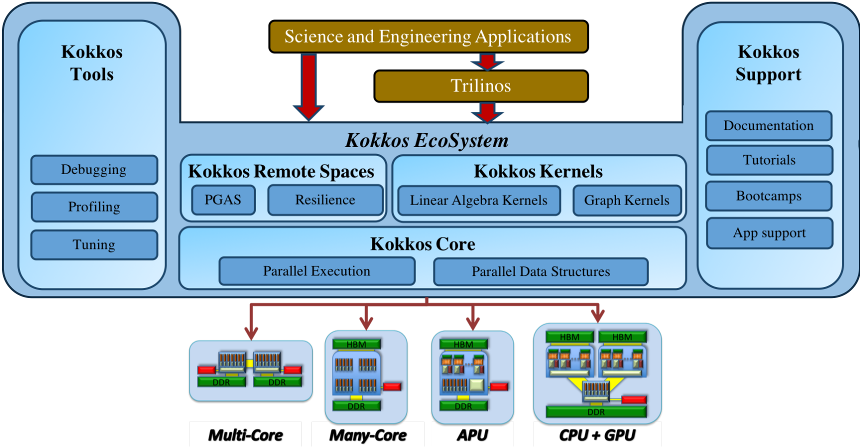
LANL/SNL Trinity
Intel Haswell / Intel KNL

ANL Aurora
Intel Xeon CPUs + Xe Compute

SNL Astra
ARM Architecture

LLNL SIERRA
IBM Power9 / NVIDIA Volta

The Kokkos EcoSystem



The Kokkos Team

kokkos

Los Alamos NATIONAL LABORATORY EST.1943

Sandia National Laboratories

Argonne NATIONAL LABORATORY

OAK RIDGE National Laboratory

BERKELEY LAB

CSCS

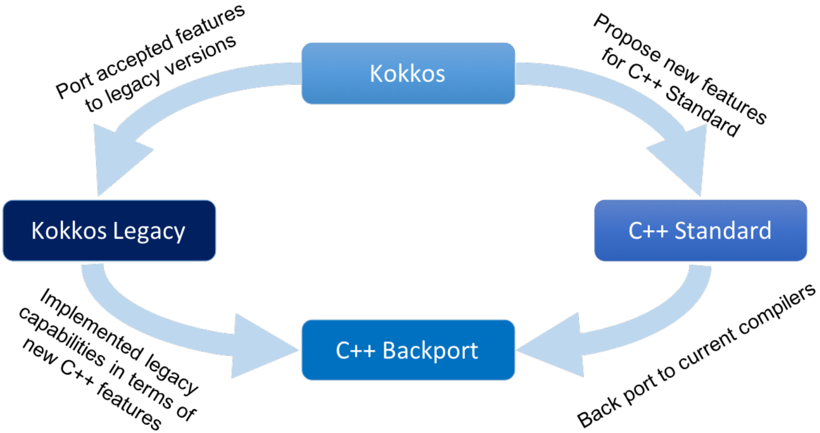
Kokkos Core: **C.R.Trott**, J. Ciesko, V. Dang, N. Ellingwood, D.S. Hollman, D. Ibanez, J. Miles, J. Wilke, , H. Finkel, N. Liber, D. Lebrun-Grandie, D. Arndt, B. Turcksin, J. Madsen, R. Gayatri
`former:` H.C. Edwards, D. Labreche, G. Mackey, S. Bova, D. Sunderland

Kokkos Kernels: **S. Rajamanickam**, L. Berger, V. Dang, N. Ellingwood, E. Harvey, B. Kelley, K. Kim, C.R. Trott, J. Wilke, S. Acer

Kokkos Tools **D. Poliakoff**, C. Lewis, S. Hammond, D. Ibanez, J. Madsen, S. Moore, C.R. Trott

Kokkos Support **C.R. Trott**, G. Shipmann, G. Womeldorff, and all of the above
`former:` H.C. Edwards, G. Lopez, F. Foertter

Kokkos helps improve ISO C++


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

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

C++20 std::atomic_ref

C++11 std::atomic insufficient for HPC

- Objects, not functions, with only atomic access
- Can't use non-atomic access in one operation, and then atomic access in the next

C++20 std::atomic_ref adds atomic capabilites as in Kokkos

- Can wrap standard allocations.
- Works also for sizes which can't be done lock-free (e.g. `complex<double>`)
- Atomic operations on reasonably arbitrary types

```
// Kokkos today
Kokkos::atomic_add(&a[i],5.0);

// atomic_ref in ISO C++20
std::atomic_ref(a[i]) += 5.0;
```

Important Point: Performance Portability

Important Point

There's a difference between *portability* and *performance portability*.

Example: implementations may target particular architectures and may not be *thread scalable*.

(e.g., locks on CPU won't scale to 100,000 threads on GPU)

Important Point: Performance Portability

Important Point

There's a difference between *portability* and *performance portability*.

Example: implementations may target particular architectures and may not be *thread scalable*.

(e.g., locks on CPU won't scale to 100,000 threads on GPU)

Goal: write **one implementation** which:

- compiles and **runs on multiple architectures**,
- obtains **performant memory access patterns** across architectures,
- can leverage **architecture-specific features** where possible.

Important Point: Performance Portability

Important Point

There's a difference between *portability* and *performance portability*.

Example: implementations may target particular architectures and may not be *thread scalable*.

(e.g., locks on CPU won't scale to 100,000 threads on GPU)

Goal: write **one implementation** which:

- compiles and **runs on multiple architectures**,
- obtains **performant memory access patterns** across architectures,
- can leverage **architecture-specific features** where possible.

Kokkos: performance portability across manycore architectures.

Concepts for Data Parallelism

Learning objectives:

- Terminology of pattern, policy, and body.
- The data layout problem.

Concepts: Patterns, Policies, and Bodies

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

Concepts: Patterns, Policies, and Bodies

Pattern

Policy

Body

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

Terminology:

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

⇒ The **pattern** and **policy** drive the computational **body**.

Threading “Parallel for”

What if we want to **thread** the loop?

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

What if we want to **thread** the loop?

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

(Change the *execution policy* from "serial" to "parallel.")

Threading “Parallel for”

What if we want to **thread** the loop?

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

(Change the *execution policy* from “serial” to “parallel.”)

OpenMP is simple for parallelizing loops on multi-core CPUs, but what if we then want to do this on **other architectures**?

Intel PHI *and* NVIDIA GPU *and* AMD GPU *and* ...

“Parallel for” on a GPU via pragmas

Option 1: OpenMP 4.5

```
#pragma omp target data map(...)
#pragma omp teams num_teams(...) num_threads(...) private(...)
#pragma omp distribute
for (element = 0; element < numElements; ++element) {
  total = 0
#pragma omp parallel for
  for (qp = 0; qp < numQPs; ++qp)
    total += dot(left[element][qp], right[element][qp]);
  elementValues[element] = total;
}
```

"Parallel for" on a GPU via pragmas

Option 1: OpenMP 4.5

```
#pragma omp target data map(...)
#pragma omp teams num_teams(...) num_threads(...) private(...)
#pragma omp distribute
for (element = 0; element < numElements; ++element) {
  total = 0
#pragma omp parallel for
  for (qp = 0; qp < numQPs; ++qp)
    total += dot(left[element][qp], right[element][qp]);
  elementValues[element] = total;
}
```

Option 2: OpenACC

```
#pragma acc parallel copy(...) num_gangs(...) vector_length(...)
#pragma acc loop gang vector
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp)
    total += dot(left[element][qp], right[element][qp]);
  elementValues[element] = total;
}
```

Portable, but not performance portable

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

But what about performance?

Portable, but not performance portable

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

But what about performance?

Performance depends upon the computation's **memory access pattern**.

Problem: memory access pattern

```
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp) {
    for (i = 0; i < vectorSize; ++i) {
      total +=
        left[element * numQPs * vectorSize +
             qp * vectorSize + i] *
        right[element * numQPs * vectorSize +
              qp * vectorSize + i];
    }
  }
  elementValues[element] = total;
}
```

Problem: memory access pattern

```
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp) {
    for (i = 0; i < vectorSize; ++i) {
      total +=
        left[element * numQPs * vectorSize +
             qp * vectorSize + i] *
        right[element * numQPs * vectorSize +
              qp * vectorSize + i];
    }
  }
  elementValues[element] = total;
}
```

Memory access pattern problem: CPU data layout reduces GPU performance by more than 10X.

Problem: memory access pattern

```
#pragma something, opencl, etc.
for (element = 0; element < numElements; ++element) {
  total = 0;
  for (qp = 0; qp < numQPs; ++qp) {
    for (i = 0; i < vectorSize; ++i) {
      total +=
        left[element * numQPs * vectorSize +
             qp * vectorSize + i] *
        right[element * numQPs * vectorSize +
              qp * vectorSize + i];
    }
  }
  elementValues[element] = total;
}
```

Memory access pattern problem: CPU data layout reduces GPU performance by more than 10X.

Important Point

For performance the memory access pattern
must depend on the architecture.

How does Kokkos address performance portability?

Kokkos is a *productive*, *portable*, *performant*, shared-memory programming model.

- is a C++ **library**, not a new language or language extension.
- provides **clear, concise, scalable** parallel patterns.
- lets you write algorithms once and run on **many architectures**
  e.g. multi-core CPU, GPUs, Xeon Phi, ...
- **minimizes** the amount of architecture-specific **implementation details** users must know.
- *solves the data layout problem* by using multi-dimensional arrays with architecture-dependent **layouts**

Data parallel patterns

Learning objectives:

- How computational bodies are passed to the Kokkos runtime.
- How work is mapped to execution resources.
- The difference between `parallel_for` and `parallel_reduce`.
- Start parallelizing a simple example.

Using Kokkos for data parallel patterns (0)

Data parallel patterns and work

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

Kokkos maps **work** to execution resources

Using Kokkos for data parallel patterns (0)

Data parallel patterns and work

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

Kokkos maps **work** to execution resources

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

Using Kokkos for data parallel patterns (0)

Data parallel patterns and work

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

Kokkos maps **work** to execution resources

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

Important concept: Work mapping

You give an **iteration range** and **computational body** (kernel) to Kokkos, and Kokkos decides how to map that work to execution resources.

Using Kokkos for data parallel patterns (2)

How are computational bodies given to Kokkos?

Using Kokkos for data parallel patterns (2)

How are computational bodies given to Kokkos?

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

Using Kokkos for data parallel patterns (2)

How are computational bodies given to Kokkos?

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

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

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

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

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

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

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

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

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

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

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

Using Kokkos for data parallel patterns (3)

How is work assigned to functor operators?

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

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

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

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

Warning: concurrency and order

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

Using Kokkos for data parallel patterns (4)

How is data passed to computational bodies?

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

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

Using Kokkos for data parallel patterns (4)

How is data passed to computational bodies?

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

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

How does the body access the data?

Important concept

A parallel functor body must have access to all the data it needs through the functor's **data members**.

Using Kokkos for data parallel patterns (5)

Putting it all together: the complete functor:

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

Using Kokkos for data parallel patterns (5)

Putting it all together: the complete functor:

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

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

Serial

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

Using Kokkos for data parallel patterns (5)

Putting it all together: the complete functor:

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

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

Serial

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

Functor

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

Using Kokkos for data parallel patterns (6)

The complete picture (using functors):

1. Defining the functor (operator+data):

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

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

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

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

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

Using Kokkos for data parallel patterns (7)

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

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

Using Kokkos for data parallel patterns (7)

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

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

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

Using Kokkos for data parallel patterns (7)

Functors are tedious ⇒ **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

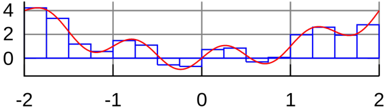
```
parallel_for(N, [=] (const int64_t i) {
  /* loop body */
});
```

Important concept

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

Riemann-sum-style numerical integration:

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

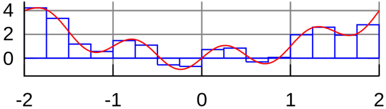


Wikipedia

Scalar integration (0)

Riemann-sum-style numerical integration:

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



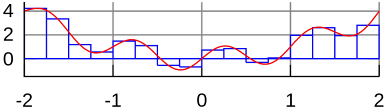
Wikipedia

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

Scalar integration (0)

Riemann-sum-style numerical integration:

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



Wikipedia

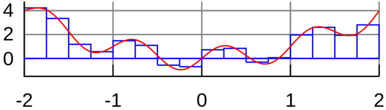
```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
  const double x =
    lower + (i/numberOfIntervals) * (upper - lower);
  const double thisIntervalsContribution = function(x);
  totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

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

Scalar integration (0)

Riemann-sum-style numerical integration:

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



Wikipedia

Pattern?

Policy?

```
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?

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

Scalar integration (1)

An (incorrect) attempt:

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

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

Scalar integration (2)

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;
```

Scalar integration (2)

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

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

Second problem: race condition

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

Scalar integration (3)

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

Scalar integration (3)

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

Scalar integration (3)

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

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

Scalar integration (3)

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

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

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

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

Scalar integration (4)

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.

Amdahl's Law (1)

Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

Amdahl's Law (1)

Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

Simplistic data-parallel performance model: Time $= \alpha + \frac{\beta * N}{P}$

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

Amdahl's Law (1)

Warning: Parallelism is NOT free

Dispatching (launching) parallel work has non-negligible cost.

Simplistic data-parallel performance model: Time $= \alpha + \frac{\beta * N}{P}$

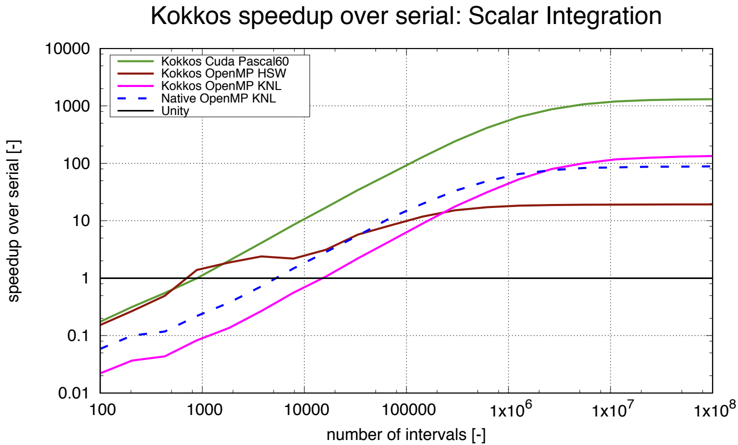
- $\alpha =$ dispatch overhead
- $\beta =$ time for a unit of work
- $N =$ number of units of work
- $P =$ available concurrency

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

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

Amdahl's Law (2)

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



Note: log scale

Always name your kernels!

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

- Non-nested parallel patterns can take an optional string argument.
- The label doesn't need to be unique, but it is helpful.
- Anything convertible to "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);
```

View motivation

Example: running daxpy on the GPU:

Lambda

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

Functor

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

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.

View motivation

Example: running daxpy on the GPU:

Lambda

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

Functor

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

Problem: x and y reside in CPU memory.

Solution: We need a way of storing data (multidimensional arrays) which can be communicated to an accelerator (GPU).

⇒ **Views**

Views (0)

View abstraction

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

High-level example of Views for daxpy using lambda:

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

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

Views (0)

View abstraction

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

High-level example of Views for daxpy using lambda:

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

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

Important point

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

Views (1)

View overview:

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

View overview:

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

Example:

```
View<double***> data("label", N0, N1, N2); //3 run, 0 compile
View<double**[N2]> data("label", N0, N1);  //2 run, 1 compile
View<double*[N1][N2]> data("label", N0);   //1 run, 2 compile
View<double[N0][N1][N2]> data("label");    //0 run, 3 compile
//Access
data(i,j,k) = 5.3;
```

Note: runtime-sized dimensions must come first.

Views (2)

View life cycle:

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

Views (2)

View life cycle:

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

Example:

```
View<double*[5]> a("a", 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?

Views (2)

View life cycle:

- Allocations only happen when *explicitly* specified.

  i.e., there are **no hidden allocations**.
- Copy construction and assignment are **shallow** (like pointers).

  so, you pass `Views` by value, *not* by reference
- Reference counting is used for **automatic deallocation.**
- They behave like `shared_ptr`

Example:

```
View<double*[5]> a("a", 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
```

Views (3)

View Properties:

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

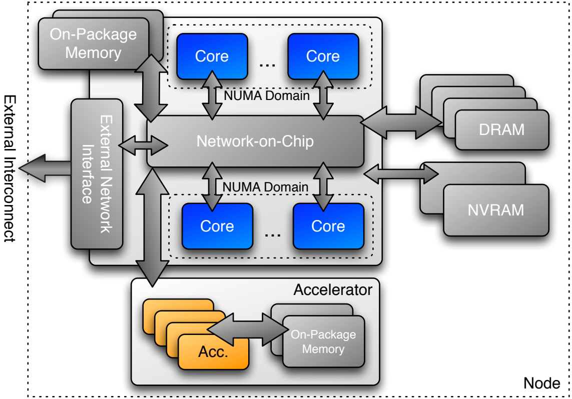
Example:

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

Execution Space

a homogeneous set of cores and an execution mechanism

(i.e., “place to run code”)



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

Execution spaces (2)

Host

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

Parallel

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

Execution spaces (2)

Host

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

Parallel

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

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

  ⇒ Always in the **host process**

Execution spaces (2)

Host

Parallel

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

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

Execution spaces (2)

Host

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

Parallel

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

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

Execution spaces (3)

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 ... */
  });
```

Execution spaces (3)

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.

Execution spaces (5)

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 */
```

Execution spaces (5)

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

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

Lambda annotation with KOKKOS_LAMBDA macro

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

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

Memory space motivating example: summing an array

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

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

Memory space motivating example: summing an array

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

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

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

Memory space motivating example: summing an array

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

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

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

Memory Space Motivation

Memory space motivating example: summing an array

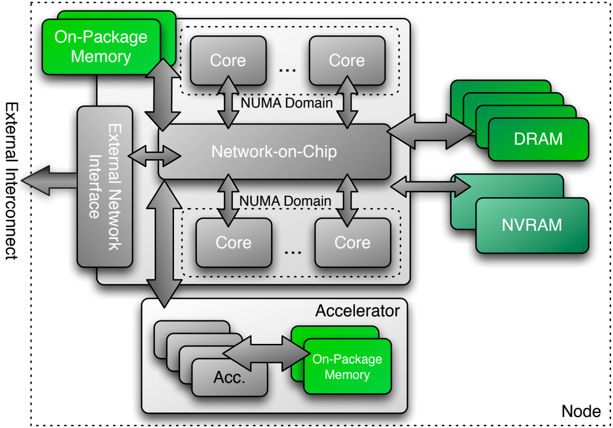
```
View<double*> data("data", size);
for (int64_t i = 0; i < size; ++i) {
  data(i) = ...read from file...
}

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

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

$\Rightarrow$ **Memory Spaces**

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



Memory spaces (1)

Important concept: Memory spaces

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

Memory spaces (1)

Important concept: Memory spaces

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

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

Memory spaces (1)

Important concept: Memory spaces

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

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

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

Memory spaces (1)

Important concept: Memory spaces

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

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

  `HostSpace, CudaSpace, CudaUVMSpace, ...` more
- Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space

Important concept: Memory spaces

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

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

  `HostSpace, CudaSpace, CudaUVMSpace, ...` more
- Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space
- If no `Space` is provided, the view's data resides in the **default memory space** of the **default execution space**.

Memory spaces (1)

Important concept: Memory spaces

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

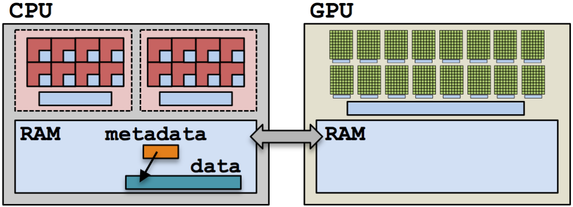
- `View<double***,`*Memory***Space**`> data(...);`
- Available **memory spaces**:

  `HostSpace, CudaSpace, CudaUVMSpace, ...` more
- Each **execution space** has a default memory space, which is used if **Space** provided is actually an execution space
- If no `Space` is provided, the view's data resides in the **default memory space** of the **default execution space**.

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

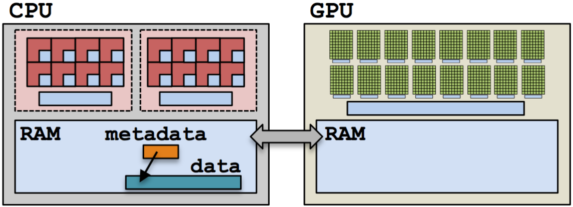
Example: HostSpace

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



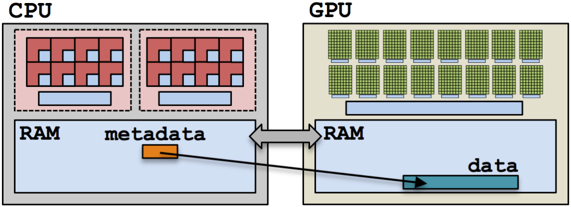
Example: HostSpace

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



Example: CudaSpace

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



Anatomy of a kernel launch:

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

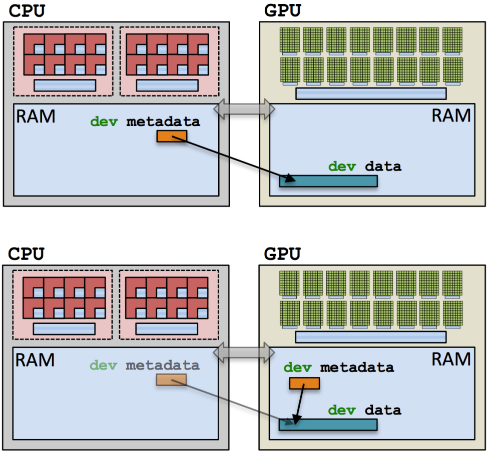
```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev(...);
parallel_for("Label",N,
  KL (int i) {
    dev(i) = ...;
  });
```

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

Execution and Memory spaces (1)

Example: one view

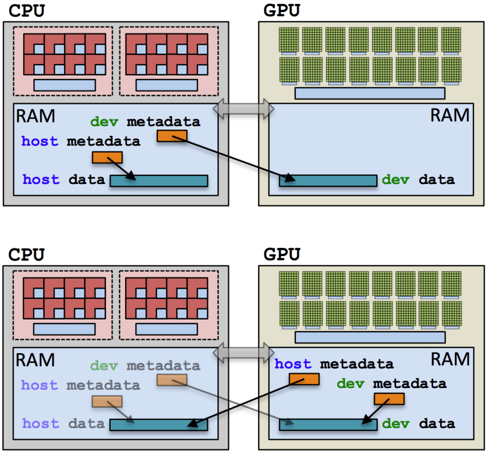
```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
parallel_for("Label",N,
  KL (int i) {
    dev(i) = ...;
  });
```



Execution and Memory spaces (2)

Example: two views

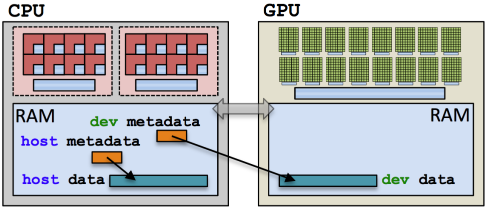
```
#define KL KOKKOS_LAMBDA
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for("Label",N,
  KL (int i) {
    dev(i)  = ...;
    host(i) = ...;
  });
```

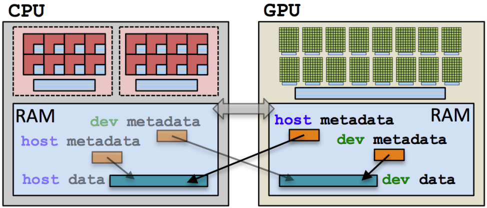


Execution and Memory spaces (2)

Example: two views

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





Execution and Memory spaces (3)

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);
```

Execution and Memory spaces (3)

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);
```

Execution and Memory spaces (4)

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);
```

Execution and Memory spaces (4)

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);
```

Execution and Memory spaces (4)

Example (redux): summing an array with the GPU

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

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

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

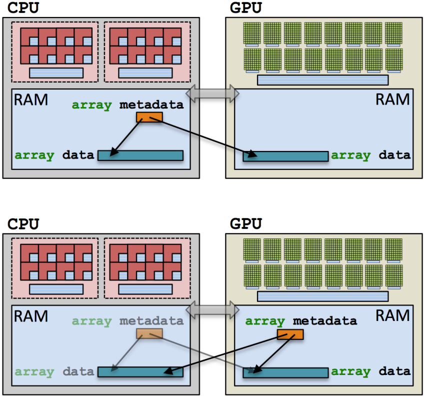
What's the solution?

- CudaUVMSpace
- CudaHostPinnedSpace (skipping)
- Mirroring

Execution and Memory spaces (5)

CudaUVMSpace

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



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

Important concept: Mirrors

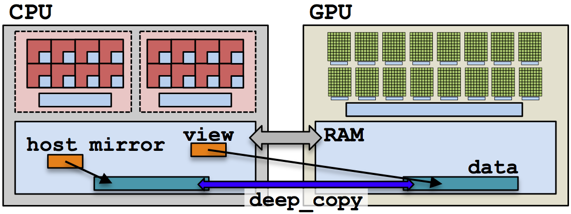
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

Important concept: Mirrors

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

Mirroring schematic

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



Mirroring pattern

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

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

Mirroring pattern

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

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

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

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

Mirroring pattern

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

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

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

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

3. **Populate** `hostView` on the host (from file, etc.).

Mirroring pattern

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

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

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

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

3. **Populate** `hostView` on the host (from file, etc.).

4. **Deep copy** `hostView`'s array to `view`'s array.

   ```
   Kokkos::deep_copy(view, hostView);
   ```

Mirroring pattern

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

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

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

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

3. **Populate** `hostView` on the host (from file, etc.).

4. **Deep copy** `hostView`'s array to `view`'s array.

```
Kokkos::deep_copy(view, hostView);
```

5. **Launch** a kernel processing the `view`'s array.

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

Mirroring pattern

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

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

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

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

3. **Populate** hostView on the host (from file, etc.).

4. **Deep copy** hostView's array to view's array.

   ```
   Kokkos::deep_copy(view, hostView);
   ```

5. **Launch** a kernel processing the view's array.

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

6. If needed, **deep copy** the view's updated array back to the hostView's array to write file, etc.

   ```
   Kokkos::deep_copy(hostView, view);
   ```

Mirrors of Views in HostSpace

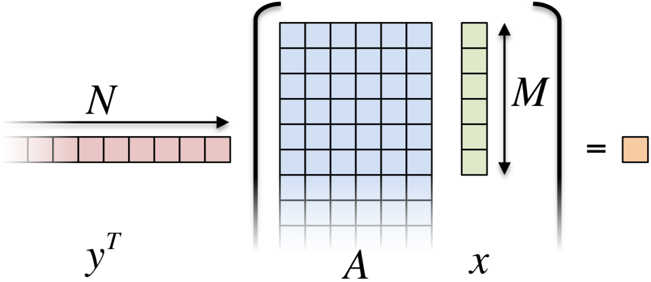
What if the View is in HostSpace too? Does it make a copy?

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

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

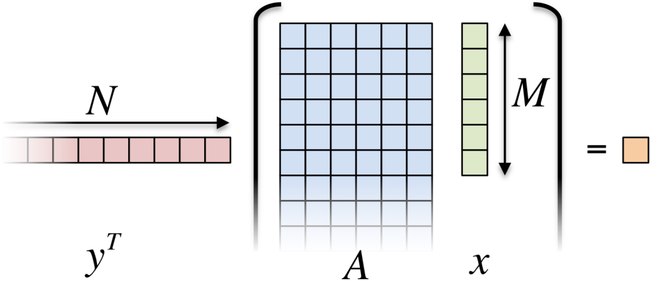
Example: inner product (0)

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



Example: inner product (0)

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

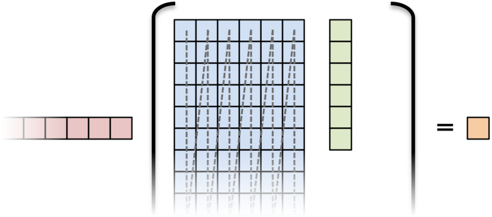


Driving question: How should A be laid out in memory?

Layout is the mapping of multi-index to memory:

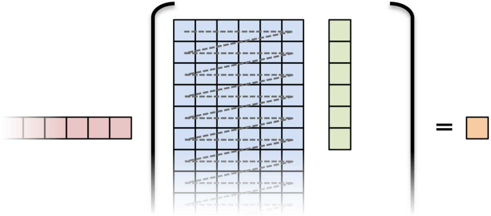
LayoutLeft

in 2D, "column-major"



LayoutRight

in 2D, "row-major"



Important concept: Layout

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

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

Important concept: Layout

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

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

- Most-common layouts are `LayoutLeft` and `LayoutRight`.

  `LayoutLeft`: left-most index is stride 1.

  `LayoutRight`: right-most index is stride 1.
- If no layout specified, default for that memory space is used.

  `LayoutLeft` for `CudaSpace`, `LayoutRight` for `HostSpace`.
- Layouts are extensible: $\approx$ 50 lines
- Advanced layouts: `LayoutStride`, `LayoutTiled`, ...

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?

Caching and coalescing (0)

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.

Caching and coalescing (0)

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;
}
```

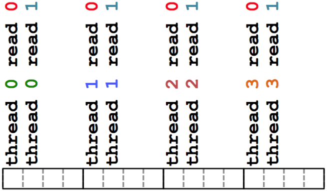
Question: once a thread reads d, does it need to wait?

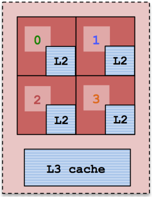
- **CPU** threads are independent.
  - i.e., threads may execute at any rate.
- **GPU** threads execute synchronized.
  - i.e., threads in groups can/must execute instructions together.

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?

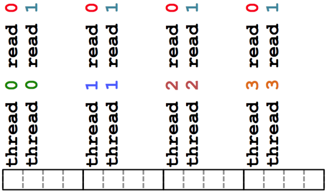
CPUs: few (independent) cores with separate caches:

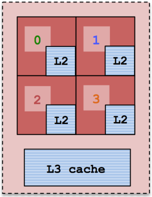




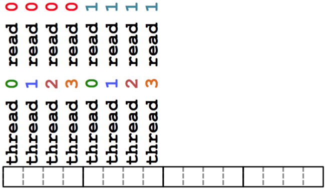
Caching and coalescing (1)

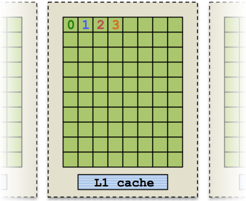
CPUs: few (independent) cores with separate caches:





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





Caching and coalescing (2)

Important point

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

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

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

Caching and coalescing (2)

Important point

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

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

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

Warning

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

Mapping indices to cores (0)

Consider the array summation example:

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

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

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

Mapping indices to cores (0)

Consider the array summation example:

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

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

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

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

Contiguous:	Strided:
`0, 1, 2, ..., N/P`	`0, N/P, 2*N/P, ...`

Mapping indices to cores (0)

Consider the array summation example:

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

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

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

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

Contiguous:
`0, 1, 2, ..., N/P`
CPU

Strided:
`0, N/P, 2*N/P, ...`
GPU

Why?

Mapping indices to cores (1)

Iterating for the execution space:

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

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

Mapping indices to cores (1)

Iterating for the execution space:

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

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

Important point

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

Mapping indices to cores (2)

Rule of Thumb

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

Example:

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

Example: inner product (2)

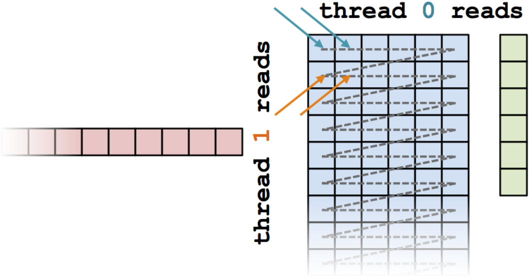
Important point

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

Important point

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

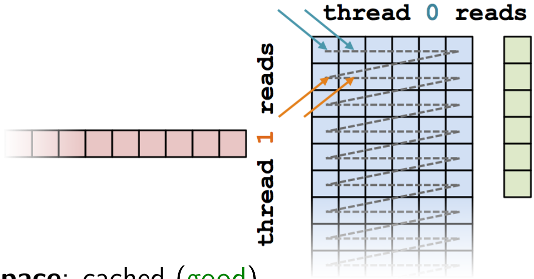
Analysis: row-major (`LayoutRight`)



Important point

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

Analysis: row-major (`LayoutRight`)

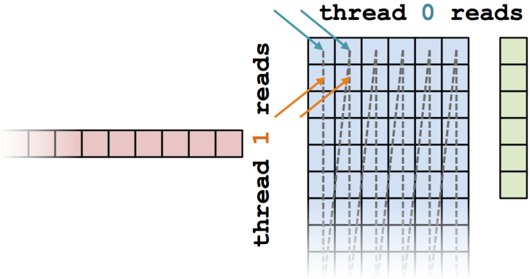


- **HostSpace**: cached (good)
- **CudaSpace**: uncoalesced (bad)

Important point

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

Analysis: column-major (`LayoutLeft`)

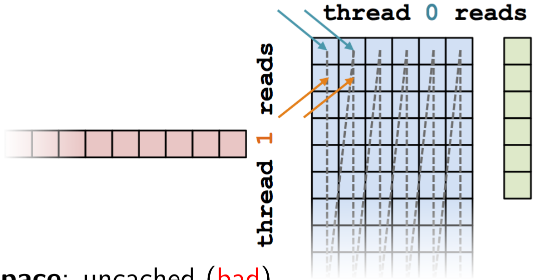


Example: inner product (3)

Important point

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

Analysis: column-major (`LayoutLeft`)

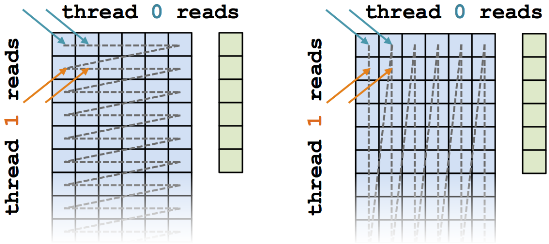


- **HostSpace**: uncached (bad)
- **CudaSpace**: coalesced (good)

Example: inner product (4)

Analysis: Kokkos architecture-dependent

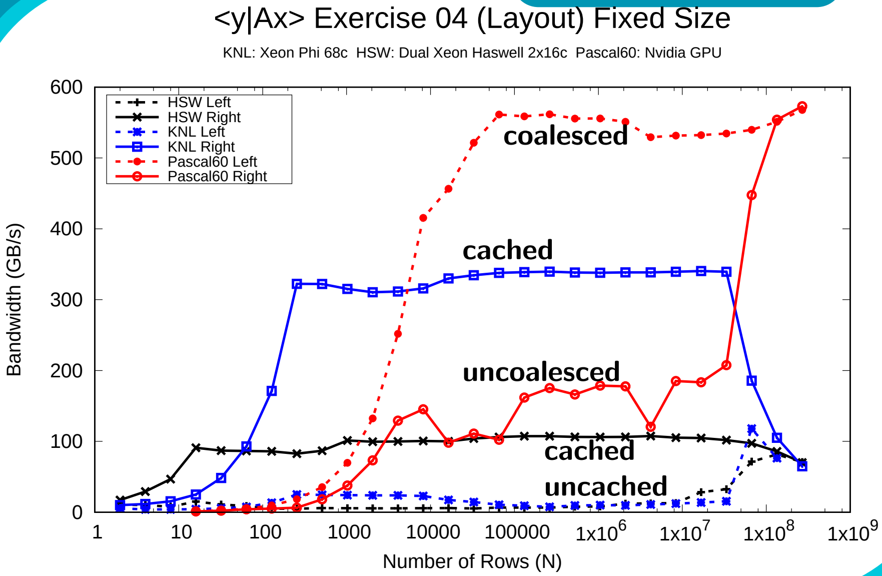
```
View<double**, ExecutionSpace> A(N, M);
parallel_for(RangePolicy< ExecutionSpace>(0, N),
  ... thisRowsSum += A(j, i) * x(i);
```



(a) OpenMP (b) Cuda

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

Example: inner product (5)



Memory Access Pattern Summary

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

  ⇒ You'll need multiple versions of code or pay the performance penalty.