

# The Kokkos Lectures

## Module 1: Introduction, Building and Parallel Dispatch

June 17, 2024

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 8+, Clang 8+, NVCC 11+, ROCM 5.2, Intel 19+

## 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](mailto:crtrott@sandia.gov)

- ▶ 07/17 Module 1: Introduction, Building and Parallel Dispatch
- ▶ 07/24 Module 2: Views and Spaces
- ▶ 07/31 Module 3: Data Structures + MultiDimensional Loops
- ▶ 08/07 Module 4: Hierarchical Parallelism
- ▶ 08/14 Module 5: Tasking, Streams and SIMD
- ▶ 08/21 Module 6: Internode: MPI and PGAS
- ▶ 08/28 Module 7: Tools: Profiling, Tuning and Debugging
- ▶ 09/04 Module 8: Kernels: Sparse and Dense Linear Algebra
- ▶ 09/11 Reserve Day

## Lectures

- ▶ Typically 90 minutes of lecture
- ▶ Submodules have associated exercise as homework
- ▶ Typically 2-3 Exercises per lecture
- ▶ Exercises will be talked through at next meeting.

## Exercises

- ▶ Exercises are small codes with places to do modifications.
- ▶ Access to GPUs helpful for most of them, but most can be done on pure CPU systems.
- ▶ Only dependent on standard compilers (e.g. Clang, NVCC)
- ▶ Office hours on Tuesdays 3-5 PM Eastern Time (potentially with AWS access).
- ▶ Ongoing support at <https://kokkosteam.slack.com>

## Introduction

What is Kokkos? Who is behind it? Why should you use it?

## Parallel Dispatch

Pattern, Policy and Body: how to parallelize simple code with Kokkos.

## Building

What do you need to build Kokkos and Apps? How to integrate into your build system?

# Introduction

## Learning objectives:

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

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



**LANL/SNL Trinity**  
Intel Haswell / Intel KNL  
*OpenMP 3*



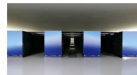
**LLNL SIERRA**  
IBM Power9 / NVIDIA Volta  
*CUDA / OpenMP<sup>(a)</sup>*



**ORNL Summit**  
IBM Power9 / NVIDIA Volta  
*CUDA / OpenACC / OpenMP<sup>(a)</sup>*



**SNL Astra**  
ARM CPUs  
*OpenMP 3*



**Riken Fugaku**  
ARM CPUs with SVE  
*OpenMP 3 / OpenACC<sup>(b)</sup>*

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



**NERSC Perlmutter**  
AMD CPU / NVIDIA GPU  
*CUDA / OpenMP 5<sup>(c)</sup>*



**ORNL Frontier**  
AMD CPU / AMD GPU  
*HIP / OpenMP 5<sup>(d)</sup>*



**ANL Aurora**  
Xeon CPUs / Intel GPUs  
*DPC++ / OpenMP 5<sup>(e)</sup>*



**LLNL El Capitan**  
AMD CPU / AMD GPU  
*HIP / OpenMP 5<sup>(d)</sup>*

(a) Initially not working. Now more robust for Fortran than C++, but getting better.

(b) Research effort.

(c) OpenMP 5 by NVIDIA.

(d) OpenMP 5 by HPE.

(e) OpenMP 5 by Intel.



## Industry Estimate

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

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

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

## Industry Estimate

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

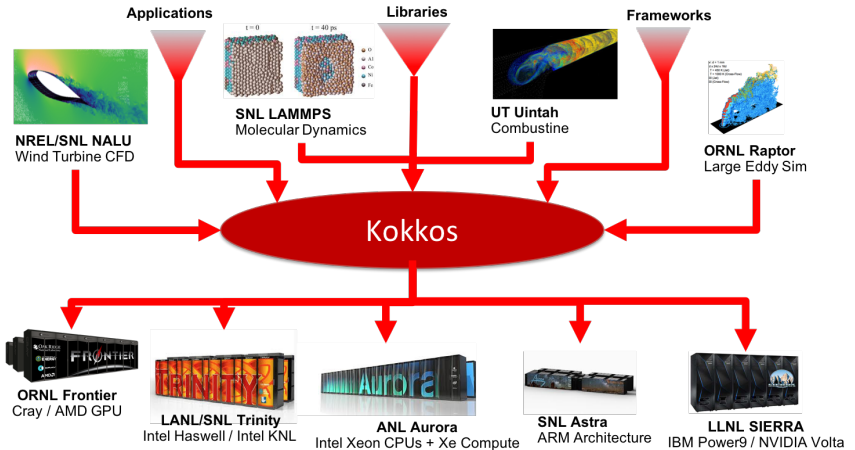
- ▶ Typical HPC production app: 300k-600k lines
  - ▶ Sandia alone maintains a few dozen
- ▶ Large Scientific Libraries:
  - ▶ E3SM: 1,000k lines
  - ▶ Trilinos: 4,000k lines

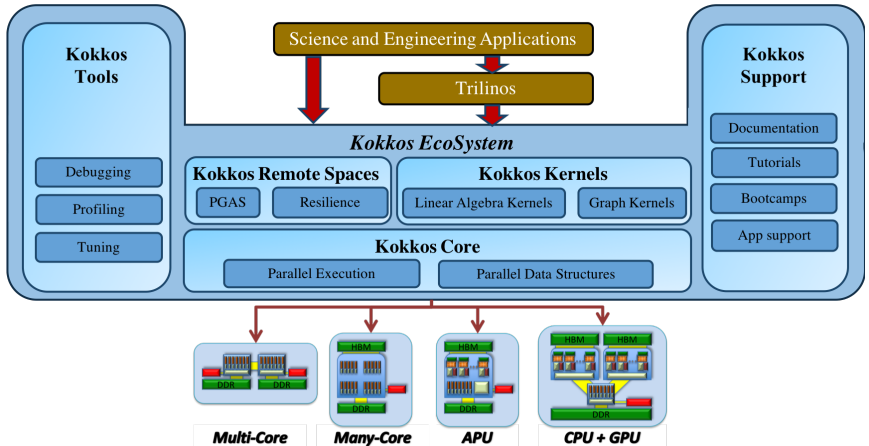
**Conservative estimate:** need to rewrite 10% of an app to switch Programming Model

## Software Cost Switching Vendors

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

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





**Kokkos Core:**

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

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

**Kokkos Kernels:**

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

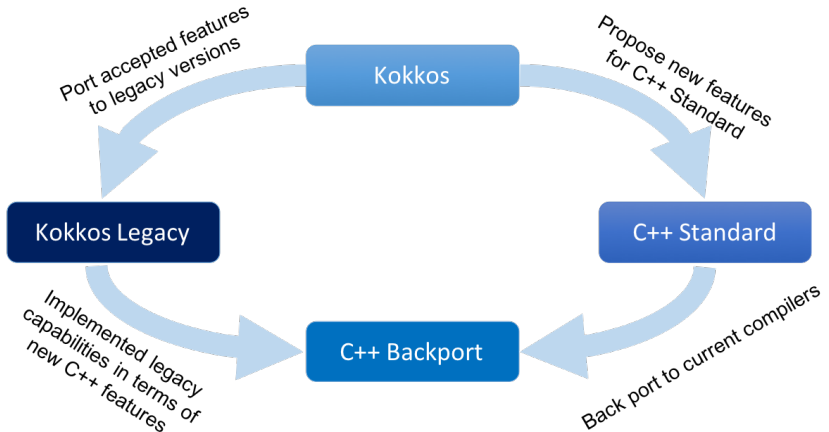
**Kokkos Tools**

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

**Kokkos Support**

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

## Kokkos helps improve ISO C++



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

## 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 capabilities 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;
```



## C++ does not provide multi dimensional arrays

- ▶ Every scientific programming language has them: Fortran, Matlab, Python, ...

## C++23 std::mdspan adds Kokkos::View like arrays

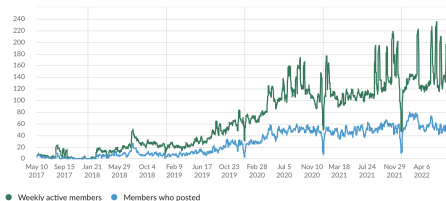
- ▶ Reference semantics.
- ▶ Compile time and runtime extents (also mixed)
- ▶ Data layouts to allow for adapting hardware specific access patterns.
- ▶ Subviews!

```
// Kokkos today
View<float**[5],LayoutLeft> a("A",10,12); a(3,5,1) = 5;

// mdspan in ISO C++23
using ext = extents<int,dynamic_extent,dynamic_extent,5>;
mdspan<float,ext,layout_left> a(ptr,10,12); a[3,5,1]+=5;
```

## Kokkos has a growing OpenSource Community

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



**Knowledge of C++:** class constructors, member variables, member functions, member operators, template arguments

## Using your own $\${HOME}$

- ▶ Git
- ▶ CMake 3.16 (or newer)
- ▶ GCC 8.2 (or newer) *OR* Intel 19.0.5 (or newer) *OR* Clang 8.0 (or newer)
- ▶ CUDA nvcc 11.0 (or newer) *AND* NVIDIA compute capability 6.0 (or newer)
- ▶ git clone <https://github.com/kokkos/kokkos-tutorials>  
into  $\${HOME}/Kokkos/kokkos-tutorials$

Slides are in

$\${HOME}/Kokkos/kokkos-tutorials/LectureSeries$

Exercises are in

$\${HOME}/Kokkos/kokkos-tutorials/Exercises$

**Online Resources:**

- ▶ <https://github.com/kokkos>: Primary Kokkos GitHub Organization
- ▶ <https://kokkos.github.io/kokkos-core-wiki>: Wiki including API reference
- ▶ <https://github.com/kokkos/kokkos-tutorials>: Tutorial exercises
- ▶ <https://kokkosteam.slack.com>: Slack channel for Kokkos

**Kokkos' basic capabilities:**

- ▶ Simple 1D data parallel computational patterns
- ▶ Deciding where code is run and where data is placed
- ▶ Managing data access patterns for performance portability
- ▶ Multidimensional data parallelism

**Kokkos' advanced capabilities:**

- ▶ Thread safety, thread scalability, and atomic operations
- ▶ Hierarchical patterns for maximizing parallelism
- ▶ Task based programming with Kokkos

**Kokkos' tools and Kernels:**

- ▶ How to profile, tune and debug Kokkos code
- ▶ Interacting with Python and Fortran
- ▶ Using Kokkos Kernels math library

- ▶ Kokkos enables **Single Source Performance Portable Codes**
- ▶ **Simple things stay simple** - it is not much more complicated than OpenMP
- ▶ **Advanced performance optimizing capabilities** easier to use with Kokkos than e.g. CUDA or HIP
- ▶ Kokkos provides data abstractions critical for performance portability not available in other programming models  
**Controlling data access patterns is key for obtaining performance**
- ▶ The **Kokkos Ecosystem** comes with tools (profiling, debugging, tuning, math libraries, etc.) needed for application development in professional settings

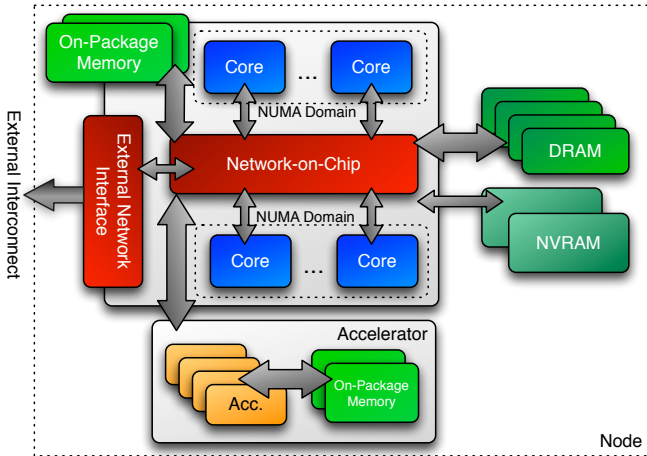
**Assume you are here because:**

- ▶ Want to use **all** HPC node architectures; including GPUs
- ▶ Are familiar with **C++**
- ▶ Want GPU programming to be **easier**
- ▶ Would like **portability**, as long as it doesn't hurt performance

**Helpful for understanding nuances:**

- ▶ Are familiar with **data parallelism**
- ▶ Are familiar with **OpenMP**
- ▶ Are familiar with **GPU architecture** and **CUDA**

### Target machine:





## 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

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

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.

```
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;  
}
```

Pattern

Policy

```
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;  
}
```

Body

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**.

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.”)



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* ...

## 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 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;
}
```

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

But what about performance?

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;  
}
```

```
#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.

```
#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.

## 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.

## 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 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**.

## Putting it all together: the complete functor:

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

## Putting it all together: the complete functor:

```
struct AtomForceFunctor {
    ForceType _atomForces;
    DataType _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);
}
```



## Putting it all together: the complete functor:

```

struct AtomForceFunctor {
    ForceType _atomForces;
    DataType _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);
}

```

## 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  $\Rightarrow$  C++11 Lambdas are concise

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

Functors are tedious  $\Rightarrow$  C++11 Lambdas are concise

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

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

Functors are tedious  $\Rightarrow$  C++11 Lambdas are concise

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

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

### Warning: Lambda capture and C++ containers

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

## How does this compare to OpenMP?

Serial

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

OpenMP

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

Kokkos

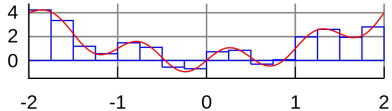
```
parallel_for(N, [=] (const int64_t i) {  
    /* loop body */  
});
```

### Important concept

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

## Riemann-sum-style numerical integration:

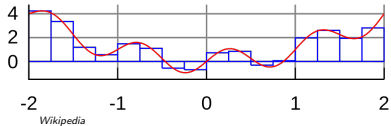
$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



Wikipedia

Riemann-sum-style numerical integration:

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

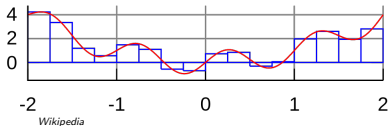


```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```



Riemann-sum-style numerical integration:

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

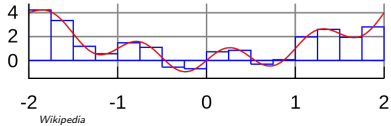


```
double totalIntegral = 0;
for (int64_t i = 0; i < numberOfIntervals; ++i) {
    const double x =
        lower + (i/numberOfIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;
```

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

Riemann-sum-style numerical integration:

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



Pattern?

```

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

```

Body?

Policy?

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

## An (incorrect) attempt:

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

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

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

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

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

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

**Second problem:** race condition

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

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

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

Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

## Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

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



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

## Important concept: Reduction

Reductions combine the results contributed by parallel work.

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

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

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

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

## Example: Scalar integration

OpenMP

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

Kokkos

```
double totalIntegral = 0;
parallel_reduce(numberOfIntervals,
    [=] (const int64_t i, double & valueToUpdate) {
        valueToUpdate += function(...);
    },
    totalIntegral);
```

- ▶ The operator takes **two arguments**: a work index and a value to update.
- ▶ The second argument is a **thread-private value** that is managed by Kokkos; it is not the final reduced value.

Warning: Parallelism is NOT free

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

**Warning: Parallelism is NOT free**

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

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

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

**Warning: Parallelism is NOT free**

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

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

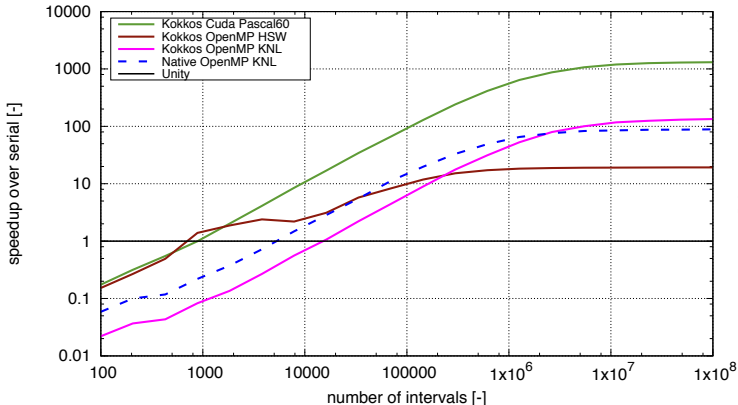
- ▶  $\alpha$  = dispatch overhead
- ▶  $\beta$  = time for a unit of work
- ▶  $N$  = number of units of work
- ▶  $P$  = available concurrency

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

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

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

### Kokkos speedup over serial: Scalar Integration



Note: log scale

## Always name your kernels!

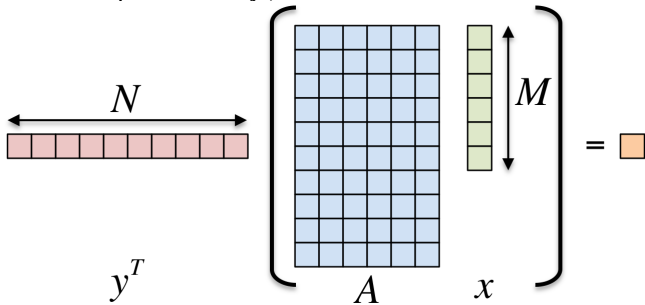
Giving unique names to each kernel is immensely helpful for debugging and profiling. You will regret it if you don't!

- ▶ 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);
```

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



**Details:**

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



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

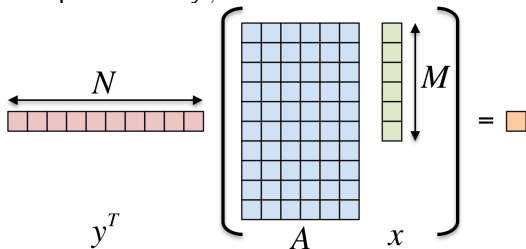
```
#include <Kokkos_Core.hpp>
int main(int argc, char* argv[]) {
    /* ... do any necessary setup (e.g., initialize MPI) ... */
    Kokkos::initialize(argc, argv);
    {
        /* ... do computations ... */
    }
    Kokkos::finalize();
    return 0;
}
```

(Optional) Command-line arguments or environment variables:

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

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

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



**Details:**

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

## Compiling for CPU

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

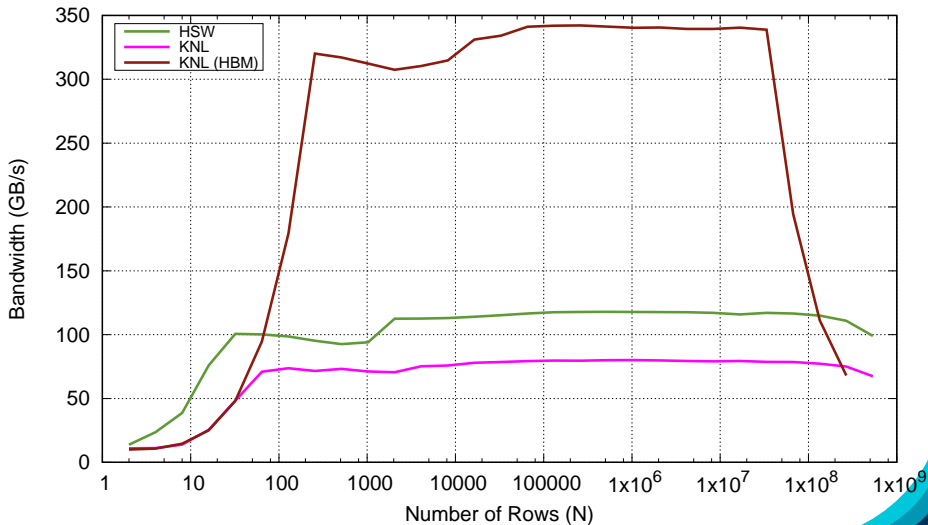
## Running on CPU with OpenMP backend

```
# Set OpenMP affinity  
export OMP_NUM_THREADS=8  
export OMP_PROC_BIND=spread OMP_PLACES=threads  
# Print example command line options:  
./build_openmp/01_Exercise -h  
# Run with defaults on CPU  
./build_openmp/01_Exercise  
# Run larger problem  
./build_openmp/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 =  $2^s == 2^{n+m}$

## &lt;y,Ax&gt; Exercise 01, Fixed Size



## More things to try: port your solution to work on the device

- ▶ You will need to update the dynamic memory allocation.
- ▶ Replace `std::malloc` and `std::free` with `Kokkos::kokkos_malloc` and `Kokkos::kokkos_free`.
- ▶ Bonus question: Why does this perform so poorly? (hint: the answer is in this slide deck somewhere)
- ▶ Note that this is just for learning purposes and by no mean a recommended way to manage the lifetime of your arrays. We will see a better way to do this soon.

## Compiling for GPU

```
cmake -B build_cuda -DKokkos_ENABLE_CUDA=ON \  
      -DKokkos_ARCH_VOLTA70=ON \  
      -DCMAKE_BUILD_TYPE=Release  
cmake --build build_cuda
```

- ▶ Customizing `parallel_reduce` data type and reduction operator  
e.g., minimum, maximum, ...
- ▶ `parallel_scan` pattern for exclusive and inclusive prefix sum
- ▶ Using *tag dispatch* interface to allow non-trivial functors to have multiple “operator()” functions.  
very useful in large, complex applications

- ▶ **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.

# Building Applications with Kokkos

## Learning objectives:

- ▶ Install Kokkos via CMake
- ▶ Build Kokkos inline via CMake
- ▶ Using Spack
- ▶ Build Kokkos inline via GNU Makefiles



# Building Applications with Kokkos

## Learning objectives:

- ▶ Install Kokkos via CMake
- ▶ Build Kokkos inline via CMake
- ▶ Using Spack
- ▶ Build Kokkos inline via GNU Makefiles

## Ignore This For Tutorial Only

The following details on options to integrate Kokkos into your build process are NOT necessary to know if you just want to do the tutorial.

- ▶ **Install Kokkos via CMake:** For large projects with multiple dependencies installing Kokkos via CMake and then building against it is the best option.
- ▶ **Build Kokkos inline via CMake:** This is an option suited for applications which have few dependencies (and no one depending on them) and want to build Kokkos inline with their application.
- ▶ **Using Spack:** For projects which largely rely on components provided by the Spack package manager.
- ▶ **Build Kokkos inline via GNU Makefiles:** The option for projects which don't want to use CMake. Only inline builds are supported via Makefiles though. Often this works well for small applications, with few if any dependencies.

- ▶ In the spirit of C++ for *code* performance portability, modern CMake aims for *build system* portability
- ▶ Projects that depend on Kokkos should be agnostic to the exact build configuration of Kokkos
- ▶ No CUDA details in C++! No CUDA details in CMake!
- ▶ Single build system call in your project should configure all compiler/linker flags:

```
add_library(myLib goTeamVenture.cpp)
target_link_libraries(myLib PUBLIC Kokkos::kokkos)
```
- ▶ Kokkos configure options are enabled/disabled via CMake as:

```
cmake -DKokkos_XYZ=ON
```

- ▶ Numerous backends can be activated
  - ▶ Only one GPU, one parallel CPU, and Serial at the same time!
- ▶ `-DKokkos_ENABLE_CUDA=ON`
- ▶ `-DKokkos_ENABLE_HIP=ON`
- ▶ `-DKokkos_ENABLE_SYCL=ON`
- ▶ `-DKokkos_ENABLE_OPENMP=ON`
- ▶ `-DKokkos_ENABLE_OPENMPTARGET=ON`

- ▶ Numerous backends can be activated
  - ▶ Only one GPU, one parallel CPU, and Serial at the same time!
- ▶ `-DKokkos_ENABLE_CUDA=ON`
- ▶ `-DKokkos_ENABLE_HIP=ON`
- ▶ `-DKokkos_ENABLE_SYCL=ON`
- ▶ `-DKokkos_ENABLE_OPENMP=ON`
- ▶ `-DKokkos_ENABLE_OPENMPTARGET=ON`
- ▶ Verify execution spaces in CMake Output, e.g. CUDA

```
-- The project name is: Kokkos
...
-- Execution Spaces:
--   Device Parallel: CUDA
--   Host Parallel: NONE
--   Host Serial: SERIAL
```

- ▶ Device backends *require* architecture be specified (CUDA , OpenMPTarget, and HIP)
  - ▶ `-DKokkos_ARCH_VOLTA70=ON`
  - ▶ `-DKokkos_ARCH_AMD_GFX90A=ON`: MI250X
- ▶ Host backends *recommend* architecture be specified to enable architecture-specific optimizations
  - ▶ `-DKokkos_ARCH_HSW=ON`: Haswell
  - ▶ `-DKokkos_ARCH_ZEN2=ON`: Ryzen (2nd gen)
- ▶ Architecture flags will automatically propagate to your project via transitive CMake properties

- ▶ Device backends *require* architecture be specified (CUDA , OpenMPTarget, and HIP)
  - ▶ `-DKokkos_ARCH_VOLTA70=ON`
  - ▶ `-DKokkos_ARCH_AMD_GFX90A=ON`: MI250X
- ▶ Host backends *recommend* architecture be specified to enable architecture-specific optimizations
  - ▶ `-DKokkos_ARCH_HSW=ON`: Haswell
  - ▶ `-DKokkos_ARCH_ZEN2=ON`: Ryzen (2nd gen)
- ▶ Architecture flags will automatically propagate to your project via transitive CMake properties
- ▶ Verify architectures in CMake Output, e.g. Volta 7.0

```
-- The project name is: Kokkos
...
-- Architectures:
-- VOLTA70
```

- ▶ Kokkos is a *C++* performance portability layer, but CUDA is usually built as a separate language with `nvcc`.
- ▶ `nvcc` doesn't accept all *C++* compiler flags
- ▶ Kokkos' solution for now is to provide `nvcc_wrapper` that converts `nvcc` into a full *C++* compiler.



- ▶ Kokkos is a *C++* performance portability layer, but CUDA is usually built as a separate language with `nvcc`.
- ▶ `nvcc` doesn't accept all *C++* compiler flags
- ▶ Kokkos' solution for now is to provide `nvcc_wrapper` that converts `nvcc` into a full *C++* compiler.
- ▶ Set CMake *C++* compiler to `nvcc_wrapper`

```
> cmake ${KOKKOS_SRC}  
-DCMAKE_CXX_COMPILER=${KOKKOS_SRC}/bin/nvcc_wrapper  
-DKokkos_ENABLE_CUDA=ON
```

- ▶ Kokkos is a C++ performance portability layer, but CUDA is usually built as a separate language with `nvcc`.
- ▶ `nvcc` doesn't accept all C++ compiler flags
- ▶ Kokkos' solution for now is to provide `nvcc_wrapper` that converts `nvcc` into a full C++ compiler.
- ▶ Set CMake C++ compiler to `nvcc_wrapper`
- ▶ CMake will report compiler as host C++ compiler

```
> cmake ${KOKKOS_SRC}
-DMAKE_CXX_COMPILER=${KOKKOS_SRC}/bin/nvcc_wrapper
-DKokkos_ENABLE_CUDA=ON
-- The CXX compiler identification is GNU 8.2.0
-- Check for working CXX compiler: bin/nvcc_wrapper
```

- ▶ Kokkos is a C++ performance portability layer, but CUDA is usually built as a separate language with `nvcc`.
- ▶ `nvcc` doesn't accept all C++ compiler flags
- ▶ Kokkos' solution for now is to provide `nvcc_wrapper` that converts `nvcc` into a full C++ compiler.
- ▶ Set CMake C++ compiler to `nvcc_wrapper`
- ▶ CMake will report compiler as host C++ compiler

```
> cmake ${KOKKOS_SRC}
-DMAKE_CXX_COMPILER=${KOKKOS_SRC}/bin/nvcc_wrapper
-DKokkos_ENABLE_CUDA=ON
-- The CXX compiler identification is GNU 8.2.0
-- Check for working CXX compiler: bin/nvcc_wrapper
```

- ▶ Or simply use `clang++` as your compiler...

**Enable HIP backend** Configure with:

```
-DKokkos_ENABLE_HIP=ON
```

**Compiler** Need to explicitly set hipcc or amdclang++ as C++ compiler:

```
-DCMAKE_CXX_COMPILER=hipcc
```

**Architecture flags** Chose one from:

```
-DKokkos_ARCH_AMD_GFX908=ON # for AMD Radeon Instinct MI100
```

```
-DKokkos_ARCH_AMD_GFX90A=ON # for AMD Radeon Instinct MI200 series
```

**Enable SYCL backend** Configure with:

```
-DKokkos_ENABLE_SYCL=ON
```

**Compiler** Need to explicitly set `icpx` as C++ compiler:

```
-DCMAKE_CXX_COMPILER=icpx
```

**Architecture flags** Chose one from:

```
-DKokkos_ARCH_INTEL_GEN=ON # JIT compiler
```

```
-DKokkos_ARCH_INTEL_PVC=ON # for GPU Max 1550/Ponte Vecchio
```

- ▶ Similar configuration as CUDA/HIP backends, but use:  
`cmake -DKokkos_ENABLE_OPENMPTARGET=ON`
- ▶ Still requires target device architecture to be given:  
`cmake -DKokkos_ARCH_VOLTA70=ON`
- ▶ Currently very sensitive to exact compiler/STL combination
  - ▶ Clang15+
  - ▶ GCC9 Toolchain
  - ▶ See [scripts/docker/Dockerfile.openmptarget](#) for recipe
- ▶ C++17 is required
- ▶ Working on Spack packages to handle complex version dependencies

Find exported Kokkos configuration (include dirs, libraries to link against, compile options, etc.) and generate my project's build system accordingly.

**Basic starting point** Create a CMakeLists.txt file.

```
cmake_minimum_required(VERSION 3.16)
project(myProject CXX) # C++ needed to build my project

find_package(Kokkos REQUIRED) # fail if Kokkos not found

# build my executable from the specified source code
add_executable(myExe source.cpp)
# declare dependency on Kokkos
target_link_libraries(myExe PRIVATE Kokkos::kokkos)
```

**Working with a library**

```
find_package(Kokkos 4.0 REQUIRED) # request Kokkos minimum version
add_library(myLib ${SOURCES})
target_link_libraries(myLib PUBLIC Kokkos::kokkos)
```

**Finding Kokkos** Add Kokkos installation prefix to the list of directories searched by CMake:

```
cmake .. -DKokkos_ROOT=<prefix> -DCMAKE_CXX_COMPILER=<...>
```

**Kokkos package introspection** Assert that support for `__host__`, `__device__` annotations in lambdas declaration is enabled

```
# (optional) assume my project uses lambdas
if(Kokkos_ENABLE_CUDA)
    # fatal error if not enabled
    kokkos_check(OPTIONS CUDA_ENABLE_LAMBDA)
endif()
```

or query that generation of relocatable device code is enabled

```
kokkos_check(
    DEVICES CUDA
    OPTIONS CUDA_RELOCATABLE_DEVICE_CODE
    RESULT_VARIABLE KOKKOS_HAS_CUDA_RDC)
if(KOKKOS_HAS_CUDA_RDC)
    ...
```



Build Kokkos as part of your own project (as opposed to finding a pre-installed Kokkos)

```
add_subdirectory(<kokkos source dir>)  
  
# identical as when finding an installed Kokkos package  
add_executable(myExe ${SOURCES})  
target_link_libraries(myExe PRIVATE Kokkos::kokkos)
```

Pass Kokkos options along with app-specific options at configuration time

```
cmake .. -DCMAKE_CXX_COMPILER=<kokkos dir>/bin/nvcc_wrapper \  
-DKokkos_ENABLE_CUDA=ON -DKokkos_ENABLE_CUDA_LAMBDA=ON \  
-DmyApp_ENABLE_FOO=ON -DmyApp_ENABLE_BAR=ON
```

- ▶ Spack provides a package manager that automatically downloads, configures, and installs package dependencies
- ▶ Kokkos itself can be easily installed with specific variants (+) and compilers (%)

```
spack install kokkos@develop +openmp %gcc@8.3.0
```

- ▶ Good practice is to define “best variant” in your packages.yaml directory, e.g. for Volta system

```
packages:  
  kokkos:  
    variants: +cuda +openmp +cuda_lambda +wrapper \  
              ^cuda@12.0 cuda_arch=70  
    compiler: [gcc@8.3.0]
```

- ▶ Build rules in package.py automatically map Spack variants to correct CMake options
- ▶ Run `spack info kokkos` to see full list of variants

- ▶ Build rules created in a `package.py` file
- ▶ Step 1: Declare dependency on specific version of kokkos (3.x, master, or develop)

```
class myLib(CMakePackage):  
    depends_on('kokkos@3.2')
```

- ▶ Step 2: Add build rule pointing to Spack-installed Kokkos and same C++ compiler Kokkos uses

```
def cmake_args(self):  
    options = []  
    ...  
    options.append('-DCMAKE_CXX_COMPILER={}'.format(  
        self.spec['kokkos'].kokkos_cxx)  
    options.append('-DKokkos_ROOT={}'.format(  
        self.spec['kokkos'].prefix)  
    return options
```

- ▶ Full details can be found in `Spack.md` in Kokkos repo.

### Building Kokkos inline with GNU Makefiles in three steps:

- ▶ Set Kokkos Options e.g. `KOKKOS_DEVICES`, `KOKKOS_ARCH`
- ▶ Include `Makefile.kokkos`
- ▶ Add `KOKKOS_CXXFLAGS`, `KOKKOS_LDFLAGS` etc. to build rules

### Most Important Settings:

- ▶ `KOKKOS_DEVICES`: What backends to enabled. Comma separated list: `Serial,OpenMP,Cuda,HIP,OpenMPTarget`
- ▶ `KOKKOS_ARCH`: Set architectures. Comma separated list: `HSW,Volta70,Power9,...`

### Building Kokkos inline with GNU Makefiles in three steps:

- ▶ Set Kokkos Options e.g. `KOKKOS_DEVICES`, `KOKKOS_ARCH`
- ▶ Include `Makefile.kokkos`
- ▶ Add `KOKKOS_CXXFLAGS`, `KOKKOS_LDFLAGS` etc. to build rules

### Most Important Settings:

- ▶ `KOKKOS_DEVICES`: What backends to enabled. Comma separated list: `Serial,OpenMP,Cuda,HIP,OpenMPTarget`
- ▶ `KOKKOS_ARCH`: Set architectures. Comma separated list: `HSW,Volta70,Power9,...`

### Order Matters!

Add default target, Kokkos settings, and CXXFLAGS before including `Makefile.kokkos`!

```
KOKKOS_PATH = ${HOME}/Kokkos/kokkos
SRC = $(wildcard *.cpp)
KOKKOS_DEVICES=OpenMP,Cuda
KOKKOS_ARCH = SKX,Volta70

default: test
    echo "Start Build"

CXX = clang++
CXXFLAGS = -O3 -g
LINK = ${CXX}

OBJ = $(SRC:.cpp=.o)

include $(KOKKOS_PATH)/Makefile.kokkos

test: $(OBJ) $(KOKKOS_LINK_DEPENDS)
    $(LINK) $(KOKKOS_LDFLAGS) $(OBJ) $(KOKKOS_LIBS) -o test

%.o:%.cpp $(KOKKOS_CPP_DEPENDS)
    $(CXX) $(KOKKOS_CPPFLAGS) $(KOKKOS_CXXFLAGS) $(CXXFLAGS)
        -c $<
```

- ▶ Kokkos' primary build system is CMAKE.
- ▶ Kokkos options are transitively passed on, including many necessary compiler options.
- ▶ The Spack package manager does support Kokkos.
- ▶ If you write an application, and have few if any dependencies, building Kokkos as part of your code is an option with both CMake and GNU Makefiles.

## Kokkos EcoSystem:

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

## Building Kokkos

- ▶ Kokkos' primary build system is CMAKE.
- ▶ Kokkos options are transitively passed on, including many necessary compiler options.
- ▶ The Spack package manager does support Kokkos.
- ▶ For applications with few if any dependencies, building Kokkos as part of your code is an option with CMake and GNU Makefiles.



## Data Parallelism:

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

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

- ▶ Reductions combine contributions from loop iterations

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

## **Kokkos::View:**

- ▶ Solving the data-layout issue.
- ▶ Controlling data life-time.

## **Execution and Memory Spaces:**

- ▶ How to control where data lives.
- ▶ How to control where code executes.
- ▶ How to manage data transfers.

**Don't Forget:** Join the Slack Channel and drop into our office hours on Monday.

## **Updates at:**

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