

Kokkos Tutorial

H. Carter Edwards ¹, Christian R. Trott ¹, Jeff Amelang ²

¹Sandia National Laboratories

²Harvey Mudd College

September 1-2, 2015

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

SAND2015-7205 PE

Understand Kokkos Programming Model Abstractions

- ▶ What, how and why of *performance portability*
- ▶ Hope for future-proofing (DOE's Sierra ATS-2)

Understand Kokkos Programming Model Abstractions

- ▶ What, how and why of *performance portability*
- ▶ Hope for future-proofing (DOE's Sierra ATS-2)

Morning:

- ▶ Simple data parallel computations
- ▶ Deciding where code is run and where data is placed
- ▶ Managing data access patterns for performance portability

Understand Kokkos Programming Model Abstractions

- ▶ What, how and why of *performance portability*
- ▶ Hope for future-proofing (DOE's Sierra ATS-2)

Morning:

- ▶ Simple data parallel computations
- ▶ Deciding where code is run and where data is placed
- ▶ Managing data access patterns for performance portability

Afternoon:

- ▶ Thread safety and *thread scalability*
- ▶ Thread-teams for maximizing parallelism

- ▶ Computers are becoming heterogenous and **MPI-only is no longer sufficient.**
- ▶ For **portability**, you need to use OpenMP, OpenACC, or Kokkos.
- ▶ Only Kokkos obtains performant memory access patterns via **architecture-aware** arrays and work mapping.
*i.e., not just portable, *performance portable*.*
- ▶ With Kokkos, **simple things stay simple** (parallel-for, etc.).
*i.e., it's *no more difficult* than OpenMP.*
- ▶ **Advanced patterns are simpler** with Kokkos than with native versions.
*i.e., you're *not missing out* on advanced features.*

Setting the Stage: Our Evolving HPC Environment and the Many-core Revolution

Context and motivation

Learning objectives:

- ▶ Understand multi-core and many-core revolutions.
- ▶ Difference between CPUs and GPUs.
- ▶ Portability versus performance portability.

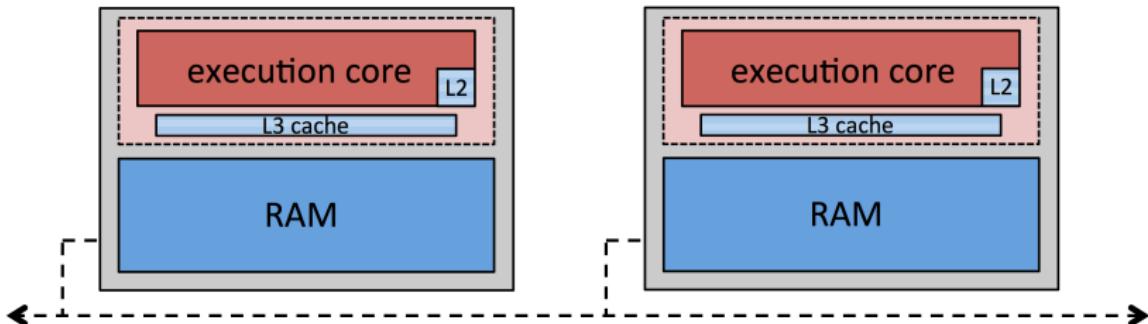
Recipe for a contemporary cluster of multi-core CPUs:

1. Use the correct **algorithm** (e.g., factorization instead of inverse, domain decomposition).
2. Implement **distributed-memory** parallelization (MPI) to use multiple nodes.
3. Choose correct **data structures** to minimize time spent on cache misses.
4. Accelerate number crunching (**vectorization**, etc.).

Recipe for a contemporary cluster of multi-core CPUs:

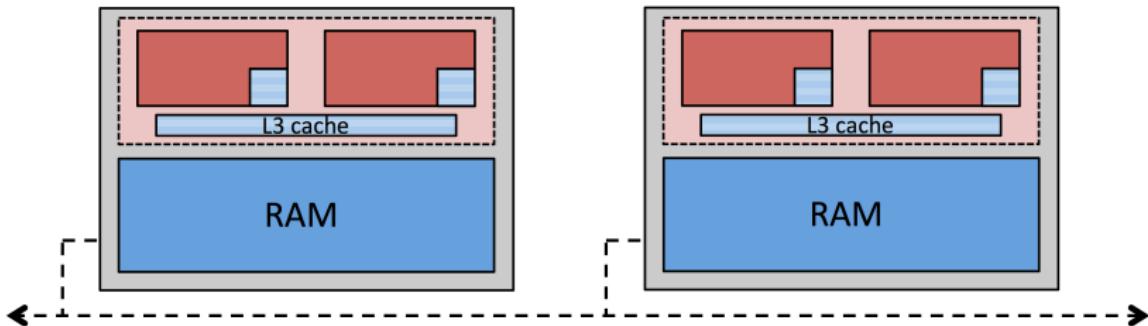
1. Use the correct **algorithm** (e.g., factorization instead of inverse, domain decomposition).
2. Implement **distributed-memory** parallelization (MPI) to use multiple nodes.
3. Choose correct **data structures** to minimize time spent on cache misses.
4. Accelerate number crunching (**vectorization**, etc.).
5. Implement **shared-memory** thread-parallelism to use all cores of a node more effectively.

2003:



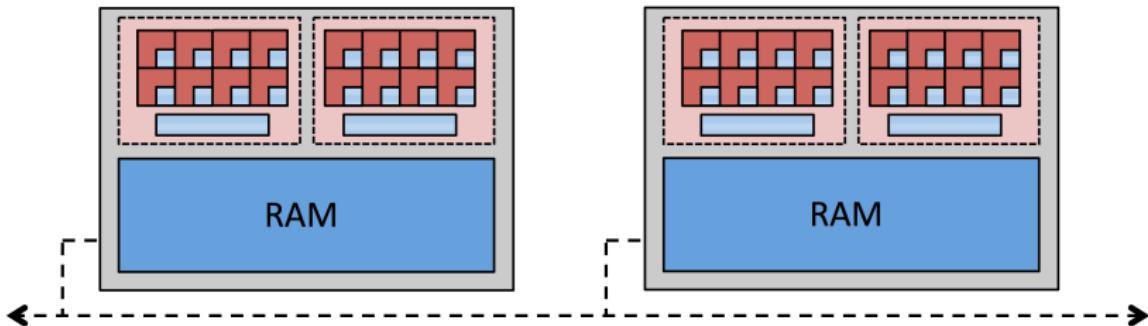
- ▶ Intel **Gallatin** architecture
 - 1 core, 3.0 GHz, 8 KB L1, 512 KB L2, 4 MB L3
- ▶ **MPI-only:**
`mpirun -np numNodes ./program`

2006:

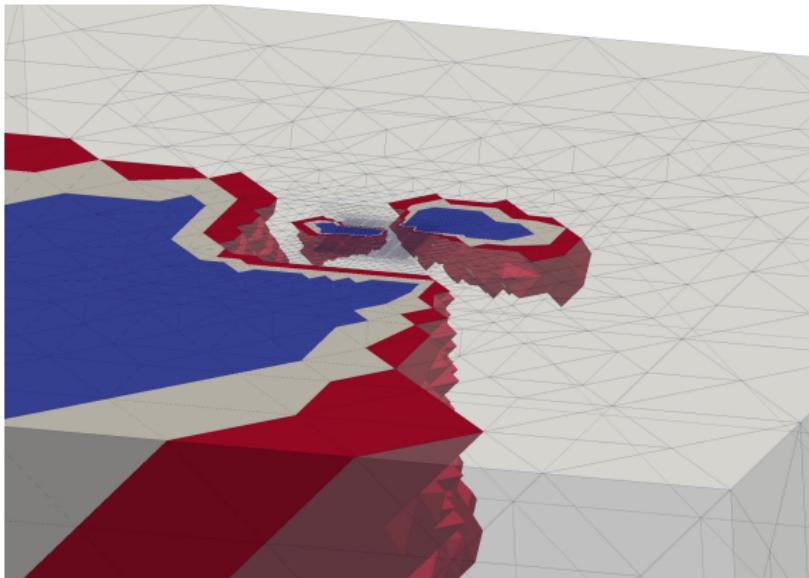


- ▶ Intel **Tulsa**
2 cores, 3.0 GHz, 32 KB L1, 1 MB L2, 8 MB L3
- ▶ **MPI-only:**
`mpirun -np 2*numNodes ./program`
- ▶ **MPI+threading ("hybrid"):**
`mpirun -np numNodes ./program -numThreads=2`

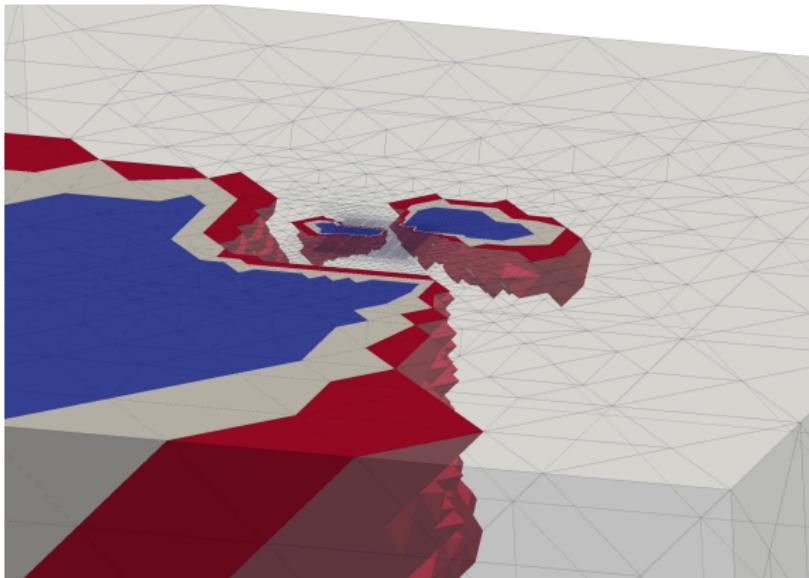
2014:



- ▶ **Intel Ivy Bridge**
8 cores, 2.0 GHz, 32 KB L1, 256 KB L2, 16 MB L3
- ▶ **MPI-only:**
`mpirun -np 16*numNodes ./program`
- ▶ **MPI+threading (“hybrid”):**
`mpirun -np numNodes ./program -numThreads=16`



- ▶ MPI-only results in **more subdomain boundary**, communication, and synchronization.



- ▶ MPI-only results in **more subdomain boundary**, communication, and synchronization.
- ▶ MPI-only requires that **all cores support** MPI processes.

Moore's law ⇒ “what do we do with all these transistors?”

Moore's law ⇒ “what do we do with all these transistors?”

CPU strategy: make a single thread go faster

- ▶ 1982: instruction cache
- ▶ 1985: data cache
- ▶ 1989: pipeline
- ▶ 1993: superscalar
- ▶ 1995: out of order execution, branch prediction
- ▶ 1999: SIMD (vector) instructions
- ▶ 2002: hyperthreading

Moore's law ⇒ “what do we do with all these transistors?”

CPU strategy: make a single thread go faster

- ▶ 1982: instruction cache
- ▶ 1985: data cache
- ▶ 1989: pipeline
- ▶ 1993: superscalar
- ▶ 1995: out of order execution, branch prediction
- ▶ 1999: SIMD (vector) instructions
- ▶ 2002: hyperthreading

Then what? 2005 finally brings **multi-core**.

Moore's law ⇒ “what do we do with all these transistors?”

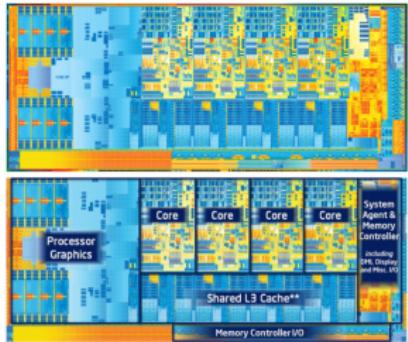
CPU strategy: make a single thread go faster

- ▶ 1982: instruction cache
- ▶ 1985: data cache
- ▶ 1989: pipeline
- ▶ 1993: superscalar
- ▶ 1995: out of order execution, branch prediction
- ▶ 1999: SIMD (vector) instructions
- ▶ 2002: hyperthreading

Then what? 2005 finally brings **multi-core**.

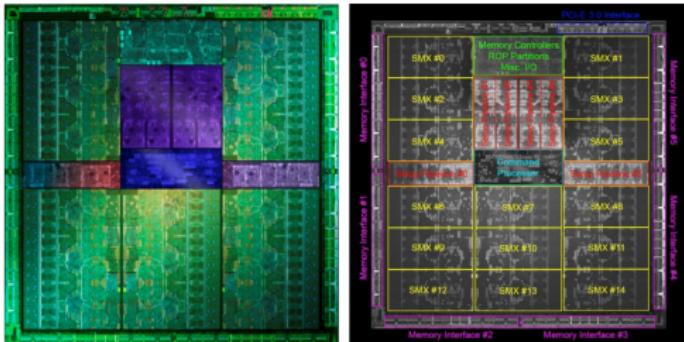
But what if instead we support many (>1000) simultaneous (slower) threads? ⇒ the **many-core revolution**

Ivy Bridge



anantech.com

Kepler



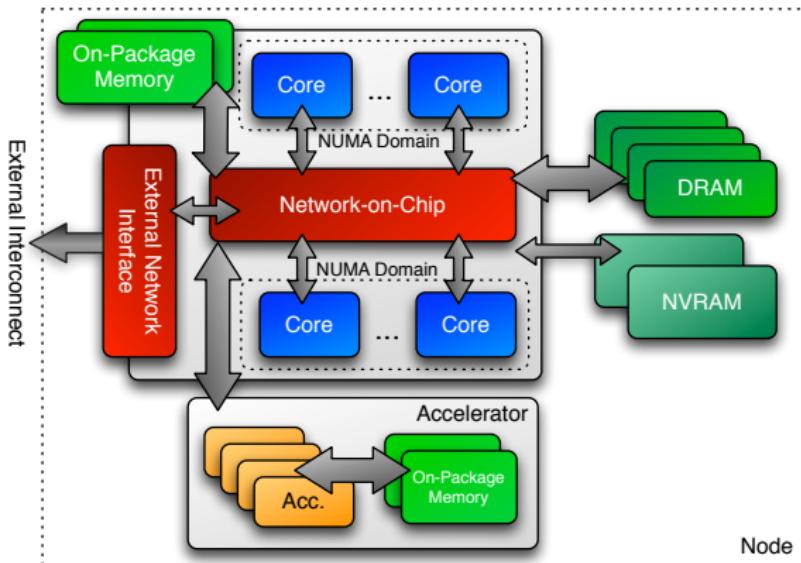
pcper.com

	Ivy Bridge	Kepler
Die size	212 mm^2	551 mm^2
Transistors	1.16B	7.1B
Core count	4	2880
Caches	big	tiny
Specialty	latency	throughput

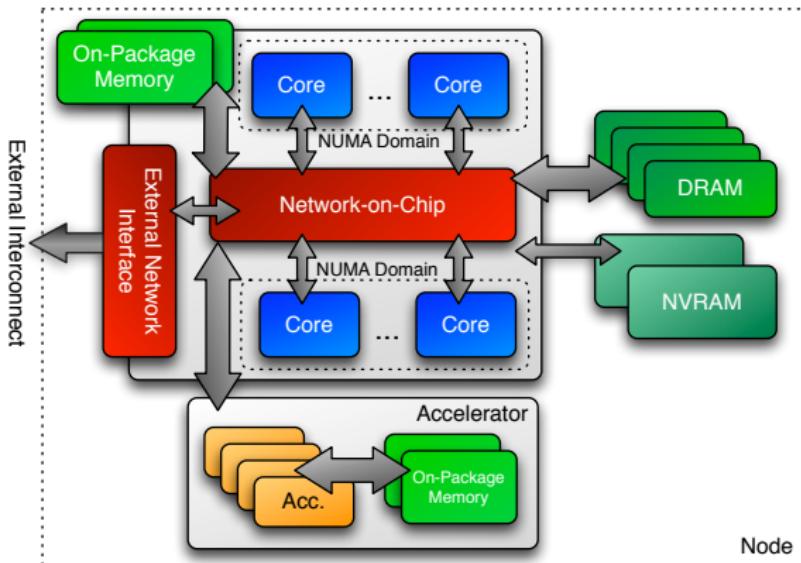
Key characteristics of GPUs:

- ▶ GPUs support **thousands** of simultaneously-executing threads.
- ▶ Cores are “**simple**” - no transistors are dedicated to branch prediction, out of order execution, etc. Instead, more cores.
- ▶ Current GPUs can't access CPU memory, have to **move data**
- ▶ GPUs have expensive memory which provides **5-10X the bandwidth** to GPU memory as CPUs have to CPU memory.
- ▶ GPUs rely on parallelism and fast context switching instead of big caches to **hide memory latency**.
- ▶ To use a GPU effectively, you need at least **$O(10,000)$ threads**.
- ▶ Four of top 10 and **90 of top 500** use accelerators.
- ▶ **Cores cannot run MPI processes.**

Compute nodes will be **heterogeneous** in cores *and* memory:

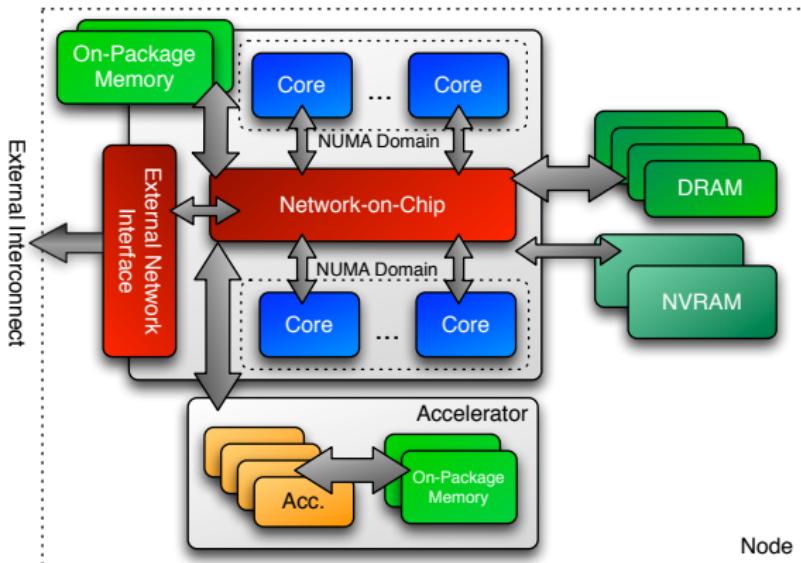


Compute nodes will be **heterogeneous** in cores *and* memory:



Many-core revolution: 20-year “just recompile” **free ride is over.**

Compute nodes will be **heterogeneous** in cores *and* memory:



Many-core revolution: 20-year “just recompile” **free ride is over.**

How much do I have to **learn and change** to use these nodes?

Kokkos

Learning objectives:

- ▶ Understand kokkos scope, goals, and philosophy.
- ▶ Difference between Kokkos and #pragma methods.
- ▶ Terminology of pattern, policy, and body.

Operating assumptions:

- ▶ Compute nodes have ~50 complex cores, ~5000 simple cores,
and heterogenous memory.
- ▶ Separate inter-node and intra-node programming models e.g.,
message passing + threading)

Operating assumptions:

- ▶ Compute nodes have ~50 complex cores, ~5000 simple cores, *and* heterogenous memory.
- ▶ Separate inter-node and intra-node programming models e.g., message passing + threading)

Goal: run on multiple architectures.

Solutions:

- ▶ Maintain **separate versions** for each target architecture (Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)

Operating assumptions:

- ▶ Compute nodes have ~50 complex cores, ~5000 simple cores, *and* heterogenous memory.
- ▶ Separate inter-node and intra-node programming models e.g., message passing + threading)

Goal: run on multiple architectures.

Solutions:

- ▶ Maintain **separate versions** for each target architecture (Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)
- ▶ Use a language or a library that runs on multiple architectures (e.g., openmp, openacc, opencl, kokkos)
 - ▶ Note: not all alternatives support heterogenous memory

Operating assumptions:

- ▶ Compute nodes have ~50 complex cores, ~5000 simple cores, *and* heterogenous memory.
- ▶ Separate inter-node and intra-node programming models e.g., message passing + threading)

Goal: run on multiple architectures.

Solutions:

- ▶ ~~Maintain separate versions for each target architecture
(Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)~~
- ▶ Use a language or a library that runs on multiple architectures (e.g., openmp, openacc, opencl, kokkos)
 - ▶ Note: not all alternatives support heterogenous memory

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.

The Beginning: Threaded (intra-node) data parallelism

What you *must have* for DOE's Trinity (ATS-1) platform

Loop bodies are prime candidates for data parallelism.

Test: Same answer if the loop iterates backwards? random order?

Loop bodies are prime candidates for data parallelism.

Test: Same answer if the loop iterates backwards? random order?

Examples:

- ▶ Forces in MD:

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

Loop bodies are prime candidates for data parallelism.

Test: Same answer if the loop iterates backwards? random order?

Examples:

- ▶ Forces in MD:

```
for (atom = 0; atom < numberAtoms; ++atom) {  
    atomForces[atom] = calculateForce(...);  
}
```

- ▶ Thermodynamic quantities at quadrature points in FEA:

```
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;  
}
```

```
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

```
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**Policy**

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 FEA problem?

What if we want to **thread** the FEA problem?

```
#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 FEA problem?

```
#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 a **GPU** too?

⇒ Want portability to DOE’s Trinity (ATS-1) *and* Sierra (ATS-2)

Option 1: OpenMP 4.0

```
#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 4.0

```
#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?

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 an *easy-to-use, portable, performant, shared-memory* programming model.

- ▶ is a C++ **library**, not a new language or language extension.
- ▶ supports **clear, concise, thread-scalable** parallel patterns.
- ▶ lets you write algorithms once and run on **many architectures**
e.g. multi-core CPU, Nvidia GPGPU, 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**

- ▶ Contemporary compute nodes are **heterogenous** in cores *and* memory.
- ▶ MPI-only is no longer possible because not all cores can run MPI processes.
- ▶ We must now leverage **heterogenous-node parallelism** in addition to MPI.

- ▶ Contemporary compute nodes are **heterogenous** in cores *and* memory.
- ▶ MPI-only is no longer possible because not all cores can run MPI processes.
- ▶ We must now leverage **heterogenous-node parallelism** in addition to MPI.

Coming up next:

- ▶ The Kokkos programming model and **library** allow you to efficiently write code for multiple architectures with minimal changes.
- ▶ Kokkos provides *thread-scalable* parallel **patterns** and execution **policies** for flexible algorithm expression.
- ▶ Kokkos **solves the data layout** problem with multi-dimensional arrays tuned for the underlying architecture.

Data parallel patterns

Learning objectives:

- ▶ How computational bodies are passed to the Kokkos runtime.
- ▶ How work is mapped to cores.
- ▶ The difference between `parallel_for` and `parallel_reduce`.
- ▶ Gain experience parallelizing some simple examples.

Data parallel patterns and work

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

Kokkos maps **work** to cores

Data parallel patterns and work

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

Kokkos maps **work** to cores

- ▶ 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 < numberAtoms; ++atomIndex) {  
    atomForces[atomIndex] = calculateForce(...data...);  
}
```

Kokkos maps **work** to cores

- ▶ 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, Kokkos maps iteration indices to cores and then runs the computational body on those cores.

How is a computational body given to Kokkos?

- ▶ Free function, like pthreads?

How is a computational body given to Kokkos?

- ▶ ~~Free function, like pthreads?~~
- ▶ Operator of a functor

What is a *functor*?

How is a computational body given to Kokkos?

- ▶ ~~Free function, like pthreads?~~
- ▶ Operator of a functor

What is a *functor*?

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

How is a computational body given to Kokkos?

- ▶ ~~Free function, like pthreads?~~
- ▶ Operator of a functor

What is a *functor*?

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

Giving a functor to Kokkos:

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

Passing work assignments to functors' operator

Intel Threading Building Blocks' approach:

```
struct Functor {  
    ...  
    void operator()(const Range & range) const {...}  
    ...  
}
```

Passing work assignments to functors' operator

Intel Threading Building Blocks' approach:

```
struct Functor {  
    ...  
    void operator()(const Range & range) const {...}  
    ...  
}
```

Problem: contiguous **ranges** are **bad** for GPUs.

Kokkos uses the **simplest interface** possible:

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

Passing work assignments to functors' operator

Intel Threading Building Blocks' approach:

```
struct Functor {  
    ...  
    void operator()(const Range & range) const {...}  
    ...  
}
```

Problem: contiguous **ranges** are **bad** for GPUs.

Kokkos uses the **simplest interface** possible:

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

Warning: concurrency and order

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

Passing data to functors' body

```
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {  
    atomForces[atomIndex] = calculateForce(...data...);  
}  
  
struct AtomForceFunctor {  
    ...  
    void operator()(const size_t atomIndex) const {  
        atomForces[atomIndex] = calculateForce(...data...);  
    }  
    ...  
}
```

Passing data to functors' body

```
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {  
    atomForces[atomIndex] = calculateForce(...data...);  
}  
  
struct AtomForceFunctor {  
    ...  
    void operator()(const size_t atomIndex) const {  
        atomForces[atomIndex] = calculateForce(...data...);  
    }  
    ...  
}
```

How does the body access the data?

Passing data to functors' body

```
for (atomIndex = 0; atomIndex < numberOfAtoms; ++atomIndex) {  
    atomForces[atomIndex] = calculateForce(...data...);  
}  
  
struct AtomForceFunctor {  
    ...  
    void operator()(const size_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**.

Manual serial execution policy:

Serial version:

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

How would we reproduce serial execution **with a functor?**

Manual serial execution policy:

Serial version:

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

How would we reproduce serial execution **with a functor?**

```
struct AtomForceFunctor {  
    ForceType _atomForces;  
    AtomDataType _atomData;  
    void operator()(const size_t atomIndex) const {  
        _atomForces[atomIndex] = calculateForce(_atomData);  
    }  
}
```

Manual serial execution policy:

Serial version:

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

How would we reproduce serial execution **with a functor?**

```
struct AtomForceFunctor {  
    ForceType _atomForces;  
    AtomDataType _atomData;  
    void operator()(const size_t atomIndex) const {  
        _atomForces[atomIndex] = calculateForce(_atomData);  
    }  
}
```

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

The complete picture (using functors):

Defining the functor (operator+data):

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

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

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

Passing the functor to Kokkos:

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

C++11 Lambda

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

C++11 Lambda

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

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

C++11 Lambda

```
atomForces already exists
data already exists
Kokkos::parallel_for(numberOfAtoms,
    [=] (const size_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 a lambda must capture by value [=]. Don't capture containers (e.g., std::vector) by value because this copies the container's entire contents.

Example: Thermodynamic quantities at quadrature points:

OpenMP

```
double **left = ..., **right = ..., *elementValues = ...;
#pragma omp parallel for
for (element = 0; element < numberOfElements; ++element) {
    double total = 0;
    for (qp = 0; qp < numberOfQuadraturePoints; ++qp) {
        total += dot(left[element][qp], right[element][qp]);
    }
    elementValues[element] = total;
}
```

Kokkos

```
double **left = ..., **right = ..., *elementValues = ...;
parallel_for(numberOfElements, [=] (const size_t element) {
    double total = 0;
    for (qp = 0; qp < numberOfQuadraturePoints; ++qp) {
        total += dot(left[element][qp], right[element][qp]);
    }
    elementValues[element] = total;
});
```

```
double * x = new double[N]; // also y
#pragma omp parallel for
for (size_t i = 0; i < N; ++i) {
    y[i] = a * x[i] + y[i];
}
```

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

```
int main(int argc, char** argv) {  
    ...  
    Kokkos::initialize(argc, argv);  
    ...  
    Kokkos::finalize();  
    return 0;  
}
```

Command-line arguments:

--kokkos-threads=INT	total number of threads (or threads within NUMA region)
--kokkos-numa=INT	number of NUMA regions
--kokkos-device=INT	device (GPU) id to use

shannon.sandia.gov (32 nodes, 2-4 GPUs each, mix of K20s, K40s, K80s)

```
salloc -N 1 -p [pbatch,fatk20x,atlas,stella]
module load gcc/4.8.3 cuda/7.5.7 nvcc-wrapper/gnu
export OMP_NUM_THREADS=8 [16]
export GOMP_CPU_AFFINITY=0-7 [0-15]
export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1
```

white.sandia.gov (9 nodes, 7 nodes have GPUs 2 don't)

```
salloc -N 1 -p [whitep,white\_no\_gpu]
module load gcc/4.9.2 cuda/7.5.7 nvcc-wrapper/gnu
export OMP_NUM_THREADS=40 [160]
export GOMP_CPU_AFFINITY=0-4,20-24,40-44,...,140-144 [0-159]
export CUDA_MANAGED_FORCE_DEVICE_ALLOC=1
```

compton.sandia.gov (40 nodes, 2 Xeon Phi Knights Corner each)

```
salloc -N 1
module load intel/compiler/15.2.164 git
export OMP_NUM_THREADS=8 [16]
export GOMP_CPU_AFFINITY=0-7 [0-15]
```

Getting the Hands-On Material

```
cp /home/kokkos-tutorial.tar ~/  
tar -xf kokkos-tutorial.tar
```

- ▶ Directory: `~/kokkos-tutorial/kokkos` is a direct git clone
- ▶ Directory: `~/kokkos-tutorial/hands-on/[EXAMPLE_NAME]/[VARIANT]` has hands-on examples
- ▶ Typically: create new variant directory; copy contents from Serial variant; start coding

Compiling

```
cd ~/kokkos-tutorial/hands-on/[EXAMPLE_NAME]/[VARIANT] ~/  
# OpenMP (Shannon/Compton)  
make -j 4 [CXX=icpc]  
# OpenMP Xeon Phi (Compton)  
make -j CXX=icpc KOKKOS_ARCH=KNC  
# OpenMP Power8 (White)  
make -j KOKKOS_ARCH=Power8  
# Cuda (Shannon/White)  
make -j 4 KOKKOS_DEVICES=Cuda,OpenMP [KOKKOS_ARCH=Kepler35,Power8]
```

Running

```
# Print example command line options:  
./EXAMPLE_NAME.host -h  
# Run with defaults on CPU  
./EXAMPLE_NAME.host  
# Run with defaults on GPU  
./EXAMPLE_NAME.cuda
```

Running on Xeon Phi (Compton only)

```
ssh 'hostname '-mic0  
source /home/projects/scripts/mic/configmicenv-2015.sh  
cd kokkos-tutorial/hands-on/[EXAMPLE_NAME]/[VARIANT]  
export KMP_AFFINITY=compact  
export OMP_NUM_THREADS=224  
./EXAMPLE_NAME.host
```

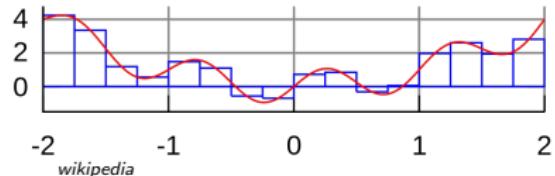
Exercise: Implement saxpy

Details:

- ▶ Make a lambda version and a functor version.
- ▶ For now, this will only use the CPU.
- ▶ Instructions in `Exercises/SAXPY/README`.
- ▶ **New material:** Compiling, running, `parallel_for`.

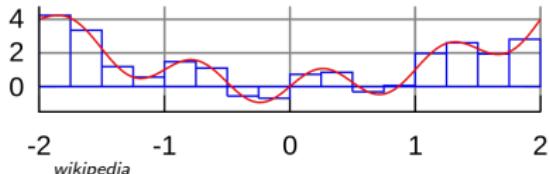
Riemann-sum-style numerical integration:

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



Riemann-sum-style numerical integration:

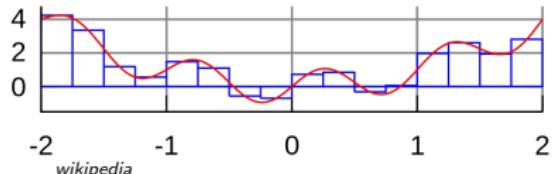
$$y = \int_{\text{lower}}^{\text{upper}} \text{function}(x) dx$$



```
double totalIntegral = 0;
for (size_t i = 0; i < number_of_intervals; ++i) {
    const double x =
        lower + (i / number_of_intervals) * (upper - lower);
    const double this_intervals_contribution = function(x);
    totalIntegral += this_intervals_contribution;
}
totalIntegral *= dx;
```

Riemann-sum-style numerical integration:

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



Pattern

```

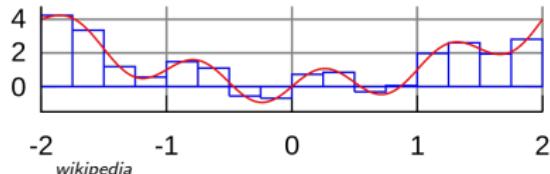
double totalIntegral = 0;                                Policy
for (size_t i = 0; i < number0fIntervals; ++i) {
    const double x =
        lower + (i/number0fIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;

```

Body

Riemann-sum-style numerical integration:

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



Pattern

```

double totalIntegral = 0;
for (size_t i = 0; i < number0fIntervals; ++i) {
    const double x =
        lower + (i/number0fIntervals) * (upper - lower);
    const double thisIntervalsContribution = function(x);
    totalIntegral += thisIntervalsContribution;
}
totalIntegral *= dx;

```

Policy

Body

How would we **parallelize** it?

An (incorrect) attempt:

```
double totalIntegral = 0;
Kokkos::parallel_for(numberOfIntervals,
    [=] (const size_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 size_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 size_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**?

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

OpenMP

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

Kokkos

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

OpenMP

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

Kokkos

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

Exercise: Implement scalar integration.

Details:

- ▶ Use a functor instead of a lambda (in preparation for the next exercise).
- ▶ Instructions in `Exercises/ScalarIntegration/README`.
- ▶ To compile the Cuda version use “`TODO: Christian`”
- ▶ To run the Cuda version use “`TODO: Christian`”
- ▶ **New material:** `parallel_reduce`.

Execution Spaces

Learning objectives:

- ▶ Heterogeneous nodes and the **execution space** abstraction.
- ▶ How to control where parallel bodies are run.
- ▶ The need for Kokkos::initialize and finalize.
- ▶ Where to use Kokkos annotation macros for portability.

Thought experiment: Consider this code:

```
section 1
MPI_Reduce(...);
FILE * file = fopen(...);
runANormalFunction(...data...);

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

section 2

Thought experiment: Consider this code:

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

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

```

section 2

- ▶ Where will **section 1** be run? CPU? GPU?
- ▶ Where will **section 2** be run? CPU? GPU?
- ▶ How do I **control** where code is executed?

Thought experiment: Consider this code:

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

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

```

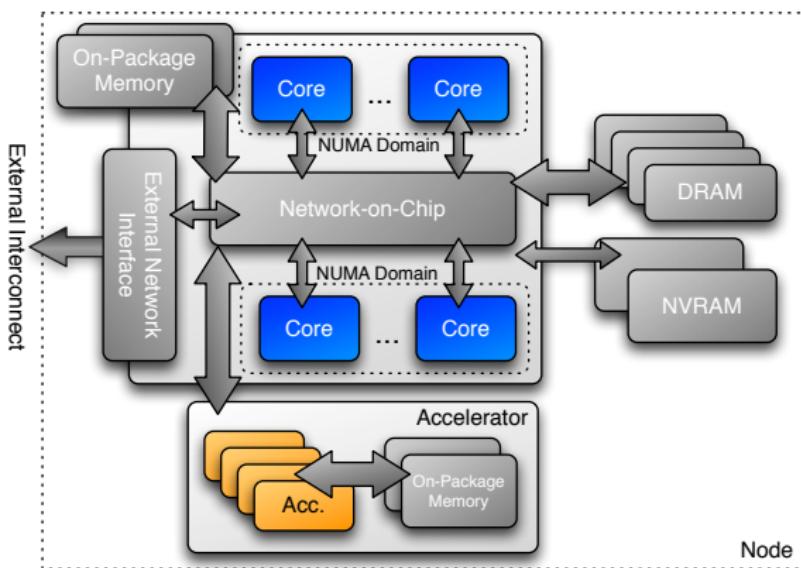
section 2

- ▶ Where will **section 1** be run? CPU? GPU?
- ▶ Where will **section 2** be run? CPU? GPU?
- ▶ How do I **control** where code is executed?

⇒ **Execution spaces**

Execution Space

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



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

Host

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

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

Parallel

Host

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

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

Parallel

- ▶ Where will **Host** code be run? CPU? GPU?
⇒ Always in the **host process**

```
Host MPI_Reduce(...);  
FILE * file = fopen(...);  
runANormalFunction(...data...);  
  
Parallel Kokkos::parallel_for(numberOfSomethings,  
                           [=] (const size_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**

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

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

Changing the parallel execution space:

Custom

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

Default

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

Custom

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

Default

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

Requirements for enabling execution spaces:

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

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```
struct ParallelFunctor {
    KOKKOS_INLINE_FUNCTION
    double helperFunction(const size_t s) const {...}
    KOKKOS_INLINE_FUNCTION
    void operator()(const size_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+GPU */
```

Kokkos function and lambda portability annotation macros:

Function annotation with KOKKOS_INLINE_FUNCTION macro

```
struct ParallelFunctor {
    KOKKOS_INLINE_FUNCTION
    double helperFunction(const size_t s) const {...}
    KOKKOS_INLINE_FUNCTION
    void operator()(const size_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+GPU */
```

Lambda annotation with KOKKOS_LAMBDA macro (CUDA requires v 7.5)

```
Kokkos::parallel_for(numberOfIterations,
    KOKKOS_LAMBDA (const size_t index) {...});

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

Exercise: Implement scalar integration with an explicit ExecutionSpace

Details:

- ▶ The functor needs to be templated on the ExecutionSpace.
- ▶ Instructions in
`Exercises/ScalarIntegration_ExecutionSpaces/README`.
TODO: Christian, what is the path to this example?
- ▶ To run the OpenMP version use “TODO: Christian”
- ▶ To run the Cuda version use “TODO: Christian”
- ▶ **New material:** explicit execution spaces.

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: $\text{Time} = \alpha + \frac{\beta * N}{P}$

- ▶ α = dispatch overhead
- ▶ β = 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.

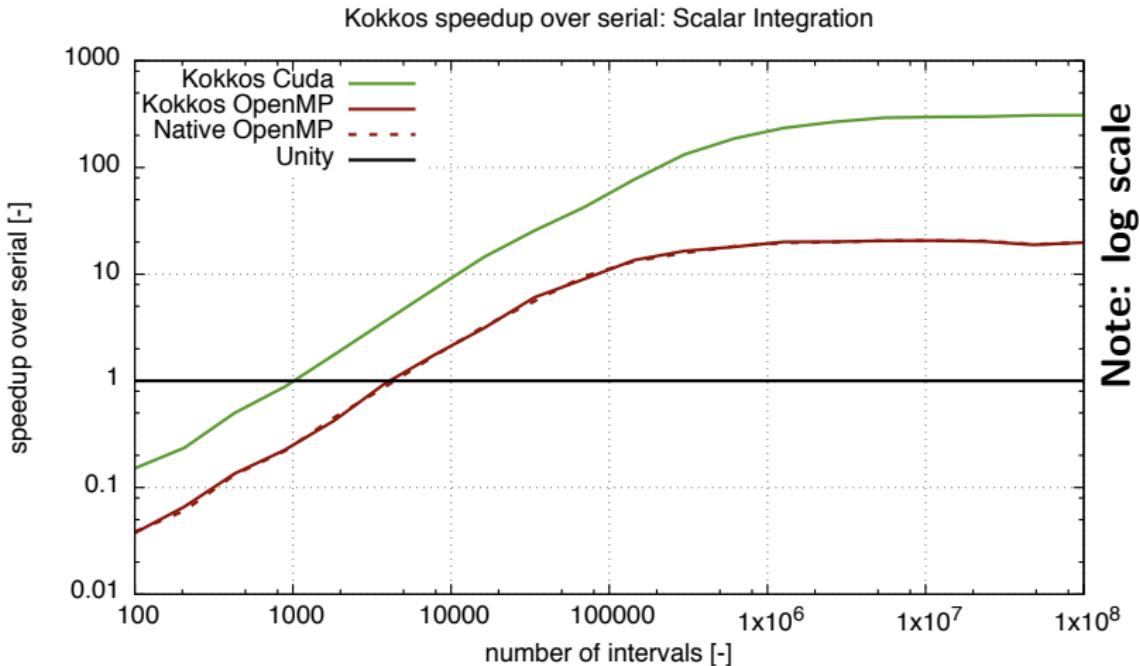
Simplistic data-parallel performance model: Time = $\alpha + \frac{\beta * N}{P}$

- ▶ α = dispatch overhead
- ▶ β = 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 α
- ▶ Find more parallelism to increase N
- ▶ Merge (fuse) parallel operations to increase β

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



Review: Simple parallel reduce using a lambda:

```
ReductionType reducedValue = <irrelevant>;
Kokkos::parallel_reduce(numberOfIterations,
    [=] (const size_t index,
        ReductionType & valueToUpdate) {
    valueToUpdate += // ... contribution for index
},
reducedValue);
```

Review: Simple parallel reduce using a lambda:

```
ReductionType reducedValue = <irrelevant>;
Kokkos::parallel_reduce(numberOfIterations,
    [=] (const size_t index,
        ReductionType & valueToUpdate) {
    valueToUpdate += // ... contribution for index
},
reducedValue);
```

Limitation of using defaults: the reduced value is (re-)initialized to zero and is reduced with operator+=.

Review: Simple parallel reduce using a lambda:

```
ReductionType reducedValue = <irrelevant>;
Kokkos::parallel_reduce(numberOfIterations,
    [=] (const size_t index,
        ReductionType & valueToUpdate) {
    valueToUpdate += // ... contribution for index
},
reducedValue);
```

Limitation of using defaults: the reduced value is (re-)initialized to zero and is reduced with operator+=.

For non-trivial reductions you need to use a **general reduction functor**.

How do you do reductions on **arbitrary types**?

Example: finding centroid of points

```
Point centroid = {{0., 0., 0.}};
for (size_t i = 0; i < numberOfPoints; ++i) {
    centroid += points[i];
}
centroid /= numberOfPoints;
```

OpenMP 3.1: Not supported

TODO: Christian

OpenMP 4.0: Hypothetically supported

```
#pragma omp declare reduction (+ : Point :  
for (int i = 0; i < 3; ++i) { omp_out[i] += omp_in[i];}) \  
[for (int i = 0; i < 3; ++i) { omp_orig[i] = 0;}]  
  
Point centroid = {{0., 0., 0.}};  
#pragma omp parallel for reduction(+:centroid)  
for (size_t i = 0; i < number0fPoints; ++i) {  
    centroid += points[i];  
}  
centroid /= number0fPoints;
```

You don't have to use the reduction clause:

Manual reductions (on a CPU):

```
double partialIntegrals[numberOfThreads]
#pragma omp parallel
{
    const unsigned int threadIndex = omp_get_thread_num();
#pragma omp for
    for (size_t i = 0; i < numberOfIntervals; ++i) {
        partialIntegrals[threadIndex] += ...;
    }
}
double totalIntegral = sum of partialIntegrals;
totalIntegral *= dx;
```

You don't have to use the reduction clause:

Manual reductions (on a CPU):

```
double partialIntegrals[numberOfThreads]
#pragma omp parallel
{
    const unsigned int threadIndex = omp_get_thread_num();
#pragma omp for
    for (size_t i = 0; i < numberOfIntervals; ++i) {
        partialIntegrals[threadIndex] += ...;
    }
}
double totalIntegral = sum of partialIntegrals;
totalIntegral *= dx;
```

Performance problem: thread scalability and false sharing

Correct (CPU) manual reductions:

```
double totalIntegral = 0;
#pragma omp parallel
{
    double localIntegral = 0;
#pragma omp for
    for (size_t i = 0; i < number_of_intervals; ++i) {
        localIntegral += ...;
    }
#pragma omp critical [well, atomic is better here]
    totalIntegral += localIntegral;
}
totalIntegral *= dx;
```

Correct (CPU) manual reductions:

```
double totalIntegral = 0;
#pragma omp parallel
{
    double localIntegral = 0;
#pragma omp for
    for (size_t i = 0; i < number_of_intervals; ++i) {
        localIntegral += ...;
    }
#pragma omp critical [well, atomic is better here]
    totalIntegral += localIntegral;
}
totalIntegral *= dx;
```

We shouldn't be thinking about this.

GPU? Xeon Phi?

Parallel programming models should support **robust**, **arbitrary**,
performant reductions **tuned to the architecture**.

General reductions:

What information must we provide to do a reduction?

- ▶ The **type** of the value to reduce (“`value_type`”)
- ▶ How to combine (“**join**”) two `value_type`s
- ▶ How to **initialize** a `value_type`

```
struct ParallelFunctor {  
    typedef double value_type;  
    void operator()(const size_t index,  
                    value_type & valueToUpdate) const {...}  
  
    void join(volatile value_type & destination,  
              const volatile value_type & source) const {...}  
  
    void init(value_type & initialValue) const {...}  
}
```

Example: finding centroid of points

```
struct ParallelFunctor {
    typedef Point value_type;
    void operator()(const size_t index,
                    value_type & valueToUpdate) const {
        valueToUpdate += _points[index];
    }

    void join(volatile value_type & destination,
              const volatile value_type & source) const {
        destination += source;
    }

    void init(value_type & initialValue) const {
        initialValue = Point::Zero();
    }
};
```

Exercise: Find the index of a point in an array of points that is closest to a search location.

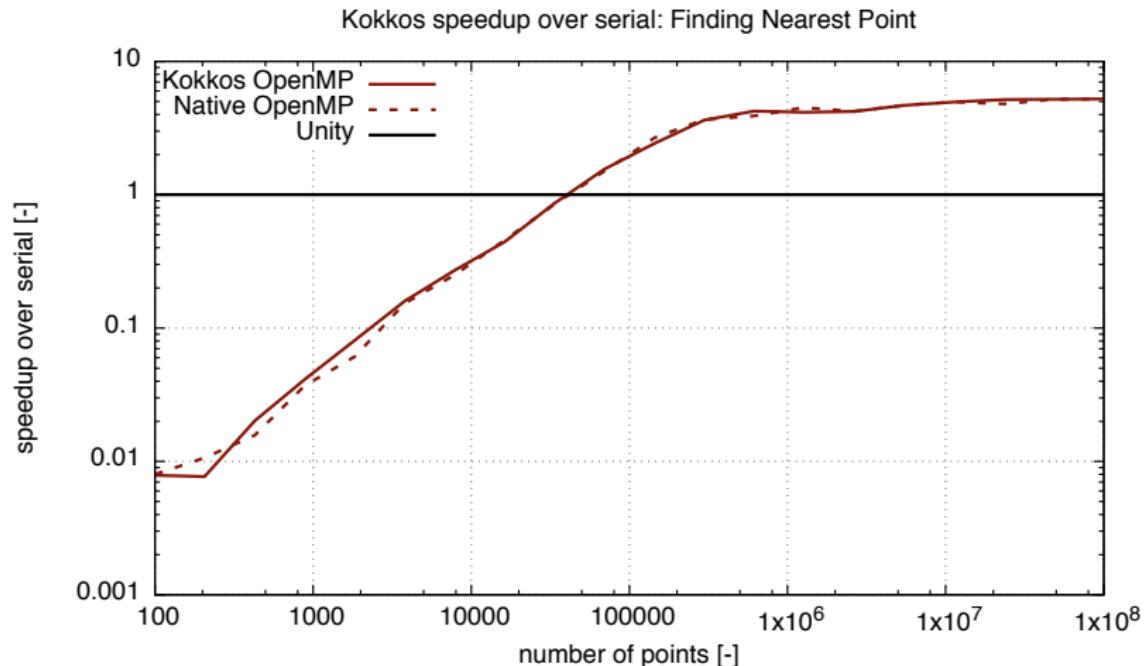
Details:

- ▶ Instructions in `Exercises/NearestPoint/README`.
- ▶ **New material:** general reductions.

Warning

Only runs in a CPU space, for now.

Results:



- ▶ Non-trivial computations have multiple parallel operations that share multiple variables.
- ▶ Natural to group these parallel operations' bodies into a single C++ struct (or class).
- ▶ Problem: only one “operator()” function.

- ▶ Non-trivial computations have multiple parallel operations that share multiple variables.
- ▶ Natural to group these parallel operations' bodies into a single C++ struct (or class).
- ▶ Problem: only one “operator()” function.
- ▶ Solution: more than one “operator()” function via *tag*.

```
struct ParallelFunctor {  
    struct TagA {};  
    struct TagB {};  
    void operator()( const TagA &, int i ) const { ... };  
    void operator()( const TagB &, int i ) const { ... };  
};  
  
ParallelFunctor func(...);  
parallel_for( RangePolicy<Space, TagA>(0,N), func );  
parallel_for( RangePolicy<Space, TagB>(0,N), func );
```

- ▶ Kokkos supports (exclusive and inclusive) **prefix scans**:

Thread value	1	2	3	4	5	6
Exclusive scan	0	1	3	6	10	15
Inclusive scan	1	3	6	10	15	21

Key idea: it's simply a change in the *pattern*.

```
ParallelFunctor functor(...);  
parallel_scan(RangePolicy<Space>(0, N), functor);
```

- ▶ Kokkos supports (exclusive and inclusive) **prefix scans**:

Thread value	1	2	3	4	5	6
Exclusive scan	0	1	3	6	10	15
Inclusive scan	1	3	6	10	15	21

Key idea: it's simply a change in the *pattern*.

```
ParallelFunctor functor(...);  
parallel_scan(RangePolicy<Space>(0, N), functor);
```

- ▶ Hierarchical parallelism through **team policies** (later).

- ▶ Kokkos supports (exclusive and inclusive) **prefix scans**:

Thread value	1	2	3	4	5	6
Exclusive scan	0	1	3	6	10	15
Inclusive scan	1	3	6	10	15	21

Key idea: it's simply a change in the *pattern*.

```
ParallelFunctor functor(...);  
parallel_scan(RangePolicy<Space>(0, N), functor);
```

- ▶ Hierarchical parallelism through **team policies** (later).
- ▶ **Task-DAG** functionality under development.

- ▶ Kokkos supports (exclusive and inclusive) **prefix scans**:

Thread value	1	2	3	4	5	6
Exclusive scan	0	1	3	6	10	15
Inclusive scan	1	3	6	10	15	21

Key idea: it's simply a change in the *pattern*.

```
ParallelFunctor functor(...);  
parallel_scan(RangePolicy<Space>(0, N), functor);
```

- ▶ Hierarchical parallelism through **team policies** (later).
- ▶ **Task-DAG** functionality under development.
- ▶ **Concurrently** executing parallel kernels on host and device.

- ▶ **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, space, and body**.
- ▶ User provides **computational bodies** as functors or lambdas which handle a single work item.
- ▶ Heterogenous nodes have one or more **execution spaces**.
- ▶ You **control where** parallel code is run by a template parameter on the execution policy, or by compile-time selection of the default execution space.



Living in a Heterogeneous Memory Architecture World

Using Kokkos

Living in a Heterogeneous Memory Architecture World

Using Kokkos

- ▶ DOE's Trinity (ATS-1) high bandwidth memory
- ▶ DOE's Sierra (ATS-2) GPU memory

Views and Memory Spaces

Learning objectives:

- ▶ Motivation behind the View abstraction.
- ▶ Key View concepts and template parameters.
- ▶ The View life cycle.

Why couldn't we run NearestPoint on the GPU?

Why couldn't we run NearestPoint on the GPU?

Version_kokkos.h:

```
struct Functor {
    const Point * _points;

    void operator()(const size_t pointIndex,
                    value_type & valueToUpdate) const {
        const Point & thisPoint = _points[pointIndex];
        ...
    }
    ...
};
```

Why couldn't we run NearestPoint on the GPU?

Version `kokkos.h`:

```
struct Functor {
    const Point * _points;

    void operator()(const size_t pointIndex,
                    value_type & valueToUpdate) const {
        const Point & thisPoint = _points[pointIndex];
        ...
    }
    ...
};
```

Problem: `_points` reside in CPU memory.

Why couldn't we run NearestPoint on the GPU?

Version `kokkos.h`:

```
struct Functor {
    const Point * _points;

    void operator()(const size_t pointIndex,
                    value_type & valueToUpdate) const {
        const Point & thisPoint = _points[pointIndex];
        ...
    }
    ...
};
```

Problem: `_points` reside in CPU memory.

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

Why couldn't we run NearestPoint on the GPU?

Version `kokkos.h`:

```
struct Functor {
    const Point * _points;

    void operator()(const size_t pointIndex,
                    value_type & valueToUpdate) const {
        const Point & thisPoint = _points[pointIndex];
        ...
    }
    ...
};
```

Problem: `_points` reside in CPU memory.

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

⇒ **Views**

High-level usage of Views in NearestPoints:

```
struct Functor {
    View<...> _points;

    Body(View<...> points) _points(points) {...}

    void operator()(const size_t pointIndex,
                     value_type & valueToUpdate) const {
        ..._points(pointIndex, coordinate)...
    }
};

View<...> points(...);
...populate points...

Functor functor(points);
parallel_reduce(N, functor);
```

High-level usage of Views in NearestPoints:

```
struct Functor {  
    View<...> _points;  
  
    Body<View<...> points> _points(points) {...}  
  
    void operator()(const size_t pointIndex,  
                    value_type & valueToUpdate) const {  
        ..._points(pointIndex, coordinate)...  
    }  
};  
  
View<...> points(...);  
... populate points...  
  
Functor functor(points);  
parallel_reduce(N, functor);
```

Important point

Views behave **like pointers** (with extra metadata)

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.

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.

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
```

Note: runtime-sized dimensions must come first.

View life cycle:

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

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**.

Example:

```
void assignValueInView(View<double*> data) { data(0) = 3; }

View<double*> a("a", N0), b("b", N0);
a(0) = 1;
b(0) = 2;
a = b;
View<double*> c(b);
assignValueInView(c);
print a(0)
```

What gets printed?

View life cycle:

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

Example:

```
void assignValueInView(View<double*> data) { data(0) = 3; }

View<double*> a("a", N0), b("b", N0);
a(0) = 1;
b(0) = 2;
a = b;
View<double*> c(b);
assignValueInView(c);
print a(0)
```

What gets printed?
3.0

Example: summing an array:

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

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

Example: summing an array:

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

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

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

Example: summing an array:

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

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

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

Example: summing an array:

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

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

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

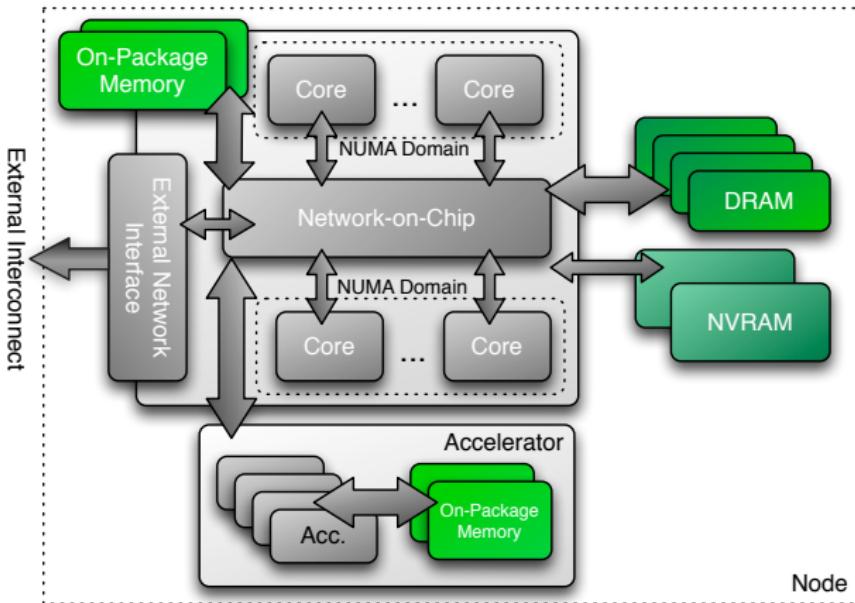
⇒ **Memory Spaces**

Memory spaces

Learning objectives:

- ▶ Node memory heterogeneity and the memory space abstraction.
- ▶ How to control where data is stored via the `MemorySpace` template parameter.
- ▶ How to avoid illegal memory access to views in different memory spaces.
- ▶ Understand motivation behind, design of, and use of mirroring.

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



Important concept: Memory spaces

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

Important concept: Memory spaces

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

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

Important concept: Memory spaces

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

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

Important concept: Memory spaces

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

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

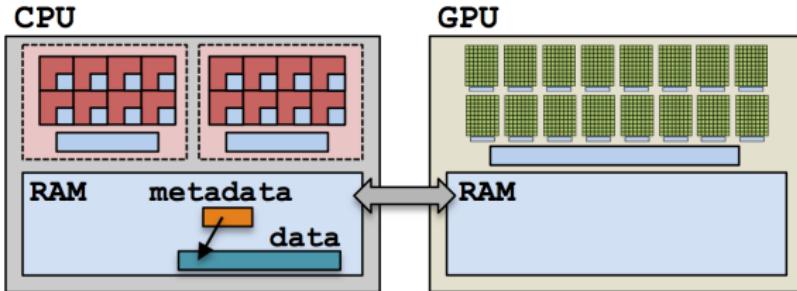
Important concept: Memory spaces

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

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

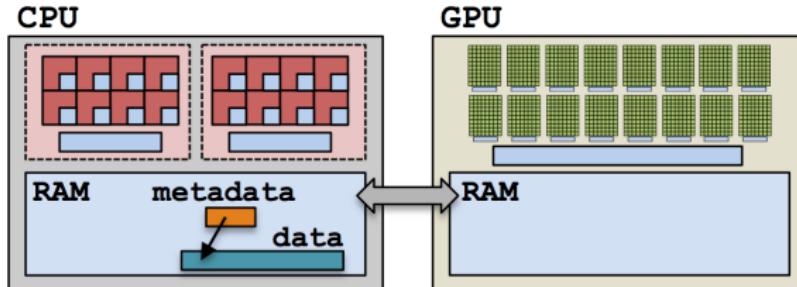
Example: HostSpace

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



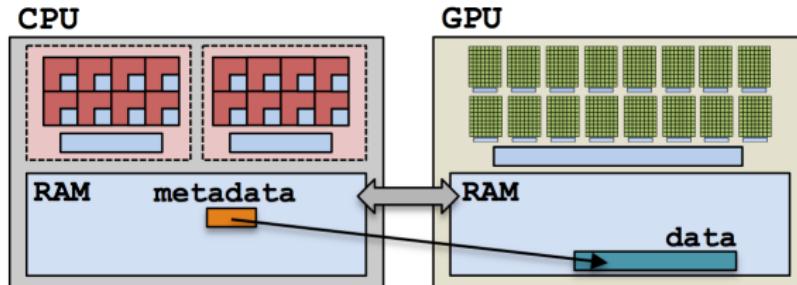
Example: HostSpace

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



Example: CudaSpace

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



Anatomy of a kernel launch:

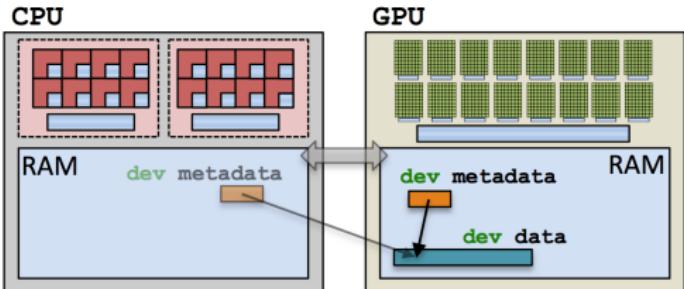
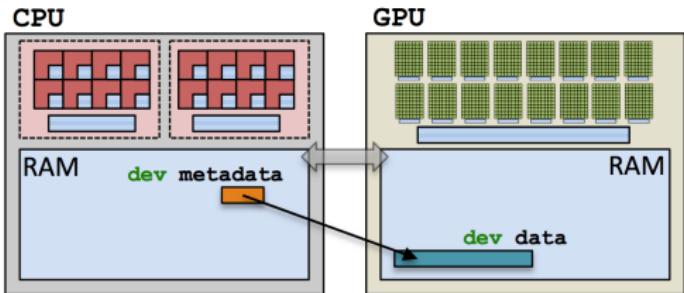
1. User declares views, allocating.
2. User instantiates a functor with views.
3. User launches parallel_***:
 - ▶ Functor is copied to the device.
 - ▶ Kernel is run.
 - ▶ Copy of functor on the device is released.

```
View<int*, Cuda> dev;  
parallel_for(N, f);  
  [=] (int i) {  
    dev(i) = ...;  
});
```

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

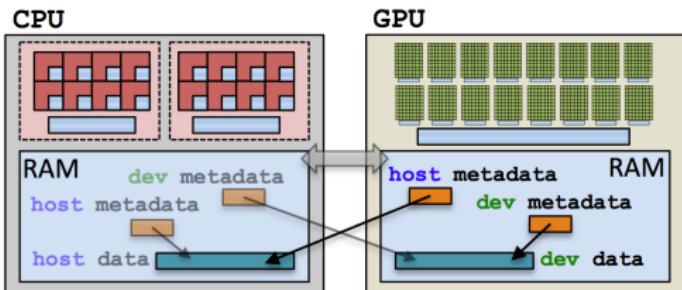
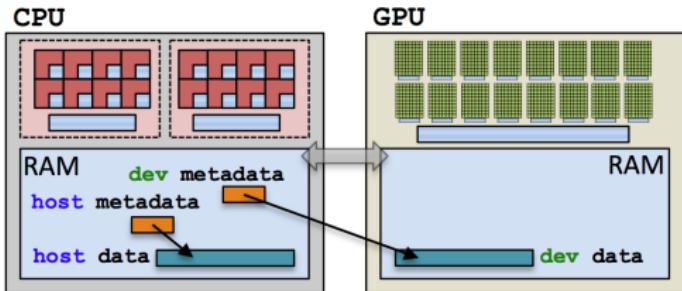
Example: one view

```
View<int*, Cuda> dev;
parallel_for(N,
 [=] (int i) {
    dev(i) = ...;
});
```



Example: two views

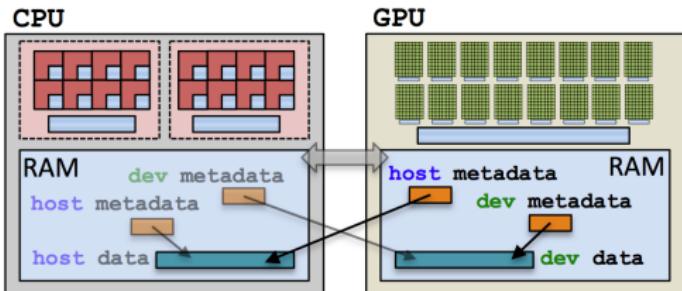
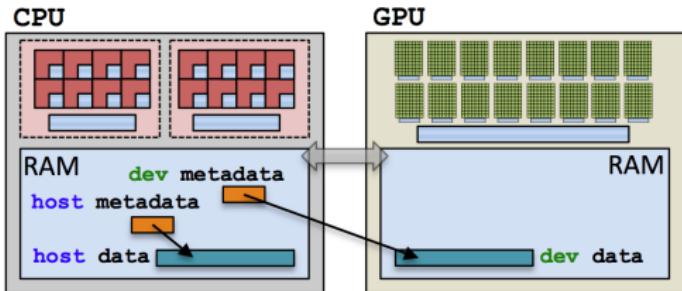
```
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for(N,
 [=] (int i) {
    dev(i) = ...;
    host(i) = ...;
});
```



Example: two views

```
View<int*, Cuda> dev;
View<int*, Host> host;
parallel_for(N,
    [=] (int i) {
        dev(i) = ...;
        host(i) = ...;
    });

```



Example (redux): summing an array with the GPU

(failed) Attempt 1:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 1:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2:

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

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

What's the solution?

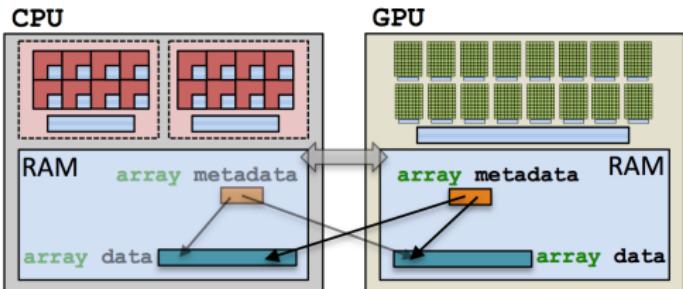
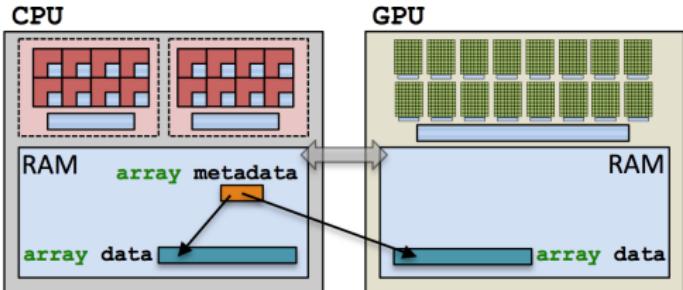
- ▶ CudaUVMSpace
- ▶ CudaHostPinnedSpace
- ▶ Mirroring

CudaUVMSpace

```

View<double*,
    CudaUVMSpace> array
array = ...from file...
double sum = 0;
parallel_reduce(N,
[=] (int i,
     double & d) {
    d += array(i);
},
sum);

```



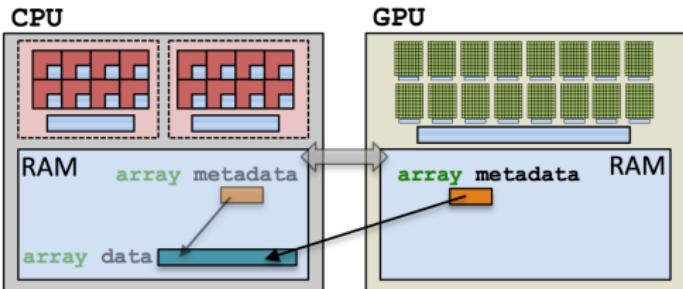
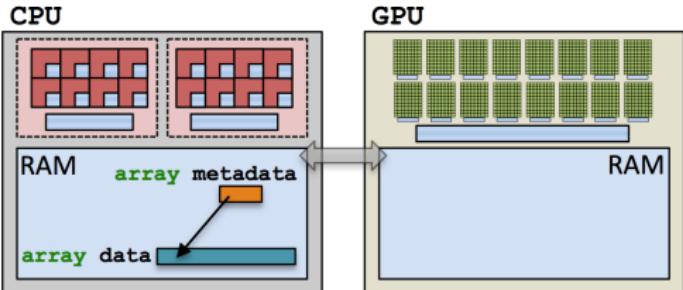
Cuda runtime automatically handles data movement,
at a **performance hit**.

CudaHostPinnedSpace

```

View<double*,
    CudaHost...> array;
array = ...from file...
double sum = 0;
parallel_reduce(N,
    [=] (int i,
        double & d) {
    d += array(i);
},
sum);

```



Cuda runtime allows cuda-code access to CPU memory,
at a **performance hit**.

Important concept: Mirrors

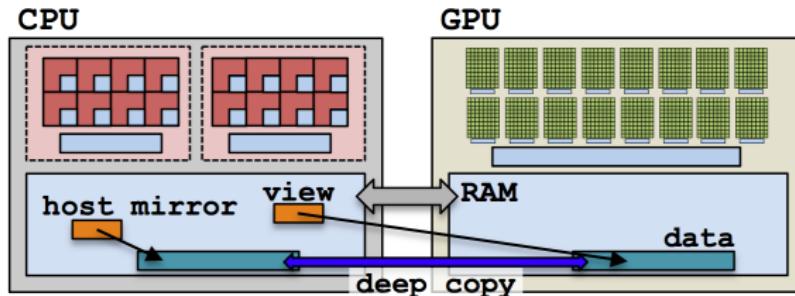
Mirrors are views of equivalent arrays residing in possibly different memory spaces.

Important concept: Mirrors

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

Mirroring schematic

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



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

```
typedef Kokkos::View<double*, Space> ViewType;  
ViewType view(...);
```

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

```
typedef Kokkos::View<double*, Space> ViewType;  
ViewType view(...);
```

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

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

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

```
typedef Kokkos::View<double*, Space> ViewType;  
ViewType view(...);
```

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

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

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

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

```
typedef Kokkos::View<double*, Space> ViewType;  
ViewType view(...);
```

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

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

3. Populate `hostView` on the host (from file, etc.).
4. Deep copy `hostView`'s array to `view`'s array.

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

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

```
typedef Kokkos::View<double*, Space> ViewType;  
ViewType view(...);
```

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

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

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

```
typedef Kokkos::View<double*, Space> ViewType;  
ViewType view(...);
```

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

```
ViewType::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(  
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);
```

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

hostView(0) = 3.14;

Kokkos::parallel_for(
    RangePolicy< Space>(0, 1),
    KOKKOS_LAMBDA (const size_t index) { print view(0) });
```

Experiment: What is printed if Space is Cuda? OpenMP?

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

hostView(0) = 3.14;

Kokkos::parallel_for(
    RangePolicy< Space>(0, 1),
    KOKKOS_LAMBDA (const size_t index) { print view(0) });
```

Experiment: What is printed if Space is Cuda? OpenMP?

Details about mirror views:

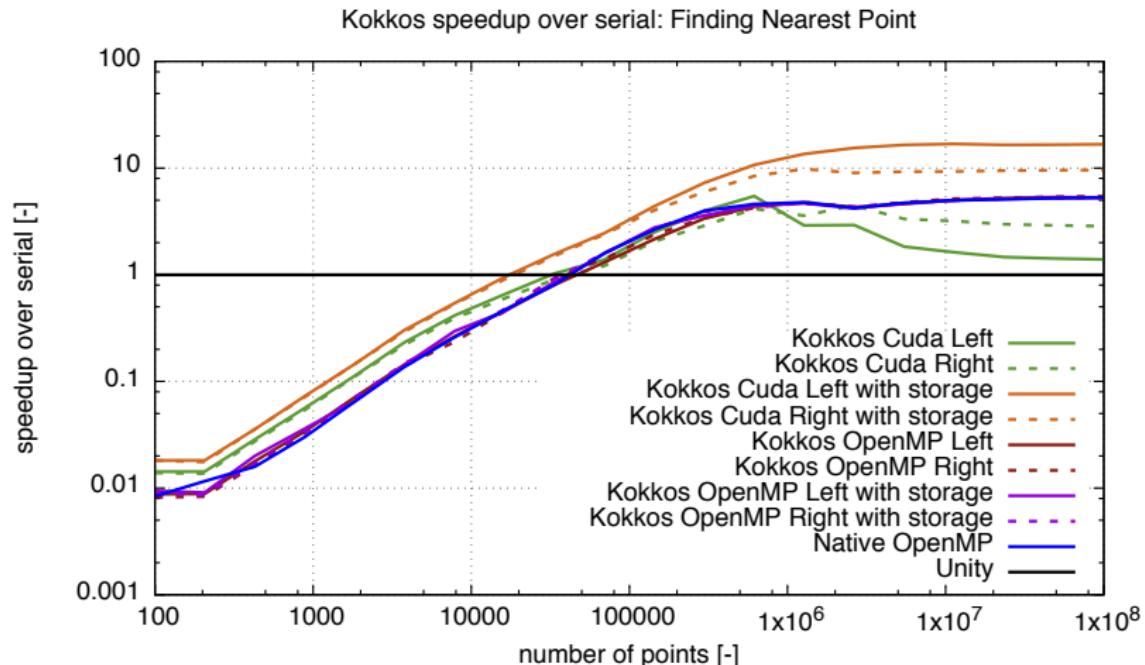
- ▶ `create_mirror_view` allocates data only if the host process cannot access `view`'s data, otherwise `hostView` references the same data.
- ▶ `create_mirror` **always** allocates data.
- ▶ Kokkos *never* performs a **hidden deep copy**.

Task: Revisit NearestPoint and plot speedup using views.

Details:

- ▶ Instructions in Exercises/NearestPoint_VIEWS/README.
TODO: Christian is this a separate example? What's the path?
- ▶ **New material:** Views, mirrors.

Results:



Managing memory access patterns for performance portability

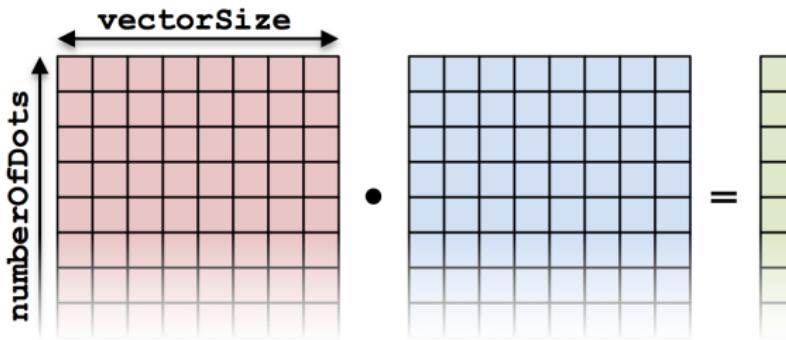
Learning objectives:

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

Let's simplify the thermodynamic quantities at quadrature points in FEA algorithm into an **array of dot products**:

```
Kokkos::parallel_for(
    RangePolicy<ExecutionSpace>(0, number0fDots),
    KOKKOS_LAMBDA (const size_t dotIndex) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(dotIndex, i) * B(dotIndex, i);
        }
        dotProducts(dotIndex) = total; });

```

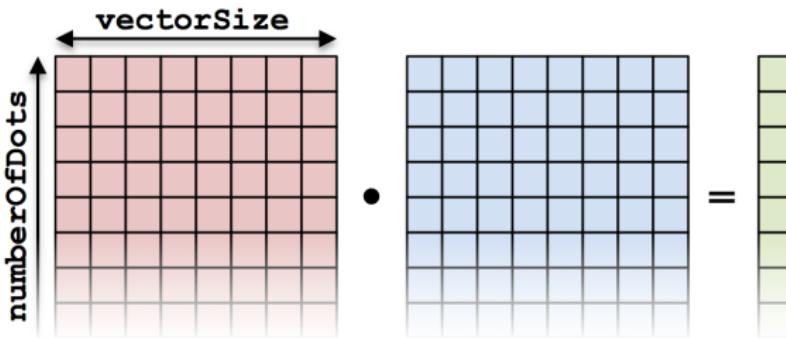


Example: array of dot products (0)

Let's simplify the thermodynamic quantities at quadrature points in FEA algorithm into an **array of dot products**:

```
Kokkos::parallel_for(
    RangePolicy<ExecutionSpace>(0, number0fDots),
    KOKKOS_LAMBDA (const size_t dotIndex) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(dotIndex, i) * B(dotIndex, i);
        }
        dotProducts(dotIndex) = total; });

```

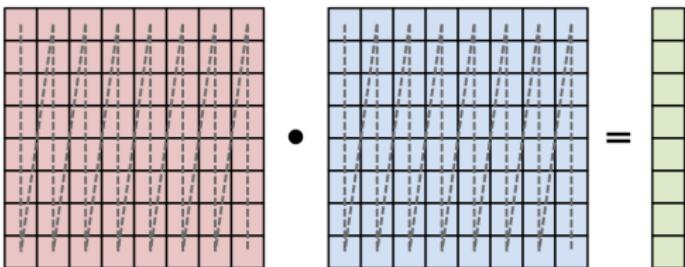


How should **A** and **B** laid out in memory?

Layout is the mapping of multi-index to memory:

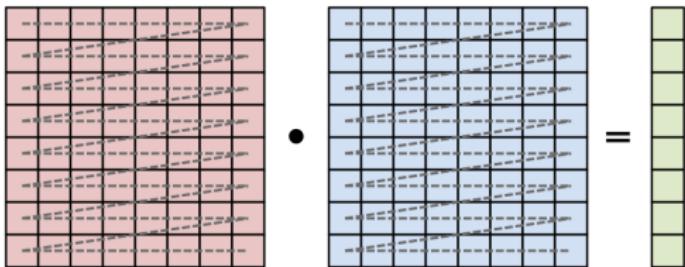
LayoutLeft

in 2D, “column-major”



LayoutRight

in 2D, “row-major”



Important concept: Layouts

Every View has a Layout set at compile-time.

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

Important concept: Layouts

Every View has a Layout set at compile-time.

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

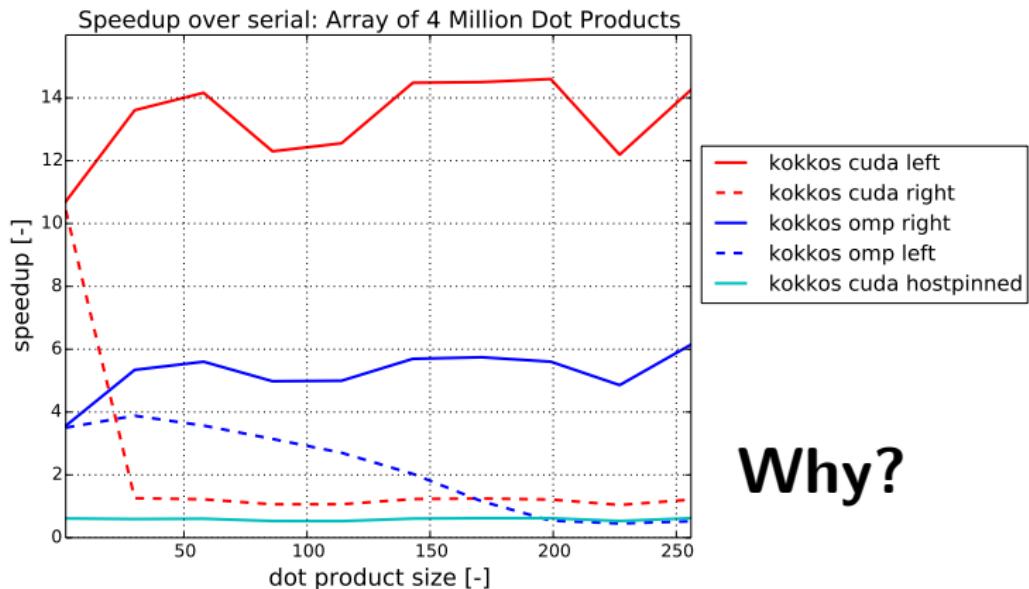
- ▶ Most-common layouts are LayoutLeft and LayoutRight.
 - LayoutLeft: left-most index is stride 1.
 - LayoutRight: right-most index is stride 1.
- ▶ If no layout specified, default for that memory space is used.
 - LayoutLeft for CudaSpace, LayoutRight for HostSpace.
- ▶ Layouts are extensible: ~50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

Task: Implement the array of dot products.

Details:

- ▶ Instructions in `Exercises/ArrayOfDotProducts/README`.
- ▶ **New material:** layouts.

Performance for 4 million dot products:



Thread independence:

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

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

Thread independence:

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

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

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

Thread independence:

```
operator()(const size_t index, double & valueToUpdate) {  
    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 are synchronized in groups (of 32).
i.e., threads in groups must execute instructions together.

Thread independence:

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

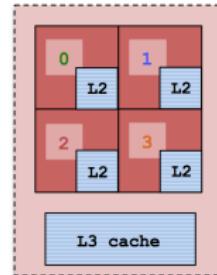
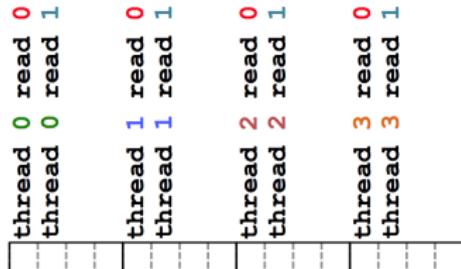
Question: once a thread reads d, does it need to wait?

- ▶ **CPU** threads are independent.
i.e., threads may execute at any rate.
- ▶ **GPU** threads are synchronized in groups (of 32).
i.e., threads in groups must execute instructions together.

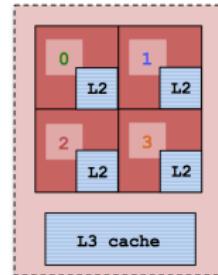
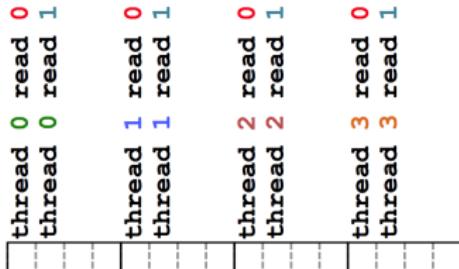
In particular, all threads in a group (*warp*) must finished their loads before *any* thread can move on.

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

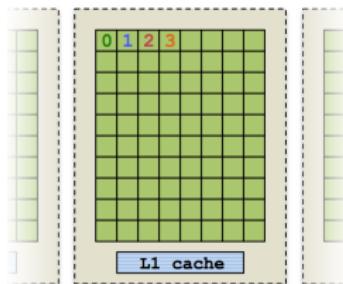
CPUs: few (independent) cores with separate caches:



CPUs: few (independent) cores with separate caches:



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



Important point

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

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

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

Important point

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

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

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

Warning

Uncoalesced access in CudaSpace *greatly* reduces performance
(more than 10X)

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 in CudaSpace *greatly* reduces performance
(more than 10X)

Note: uncoalesced *read-only, random* access in CudaSpace is okay
through Kokkos const RandomAccess views (more later).

Consider the array summation example:

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

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

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

Consider the array summation example:

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

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

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

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

Contiguous:

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

Strided:

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

Consider the array summation example:

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

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

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

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

Contiguous:

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

CPU

Strided:

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

GPU

Why?

Iterating for the execution space:

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

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

Iterating for the execution space:

```
operator()(const size_t index, double & valueToUpdate) {  
    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.

Iterating for the execution space:

```
operator()(const size_t index, double & valueToUpdate) {  
    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.

Important point

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

How do we achieve performant memory access?

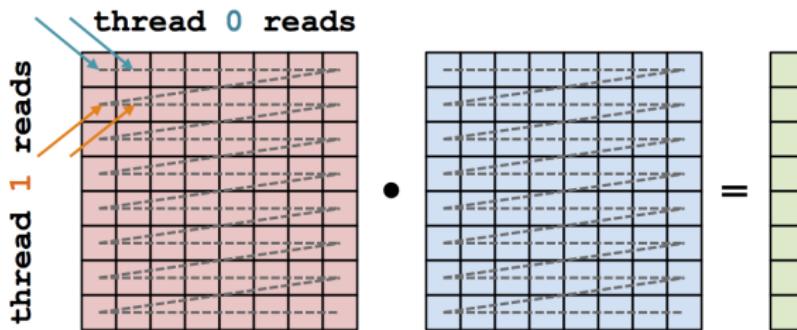
How do we achieve performant memory access?

Kokkos mapping parallel work indices **and** multidimensional array layout *optimally for the architecture.*

How do we achieve performant memory access?

Kokkos mapping parallel work indices **and** multidimensional array layout *optimally for the architecture.*

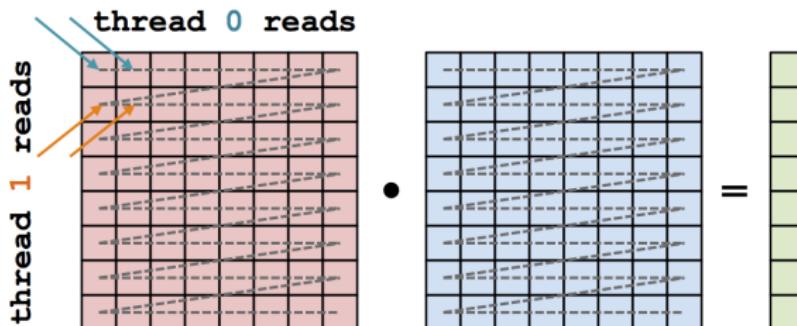
Analysis: row-major (LayoutRight)



How do we achieve performant memory access?

Kokkos mapping parallel work indices **and** multidimensional array layout *optimally for the architecture.*

Analysis: row-major (LayoutRight)

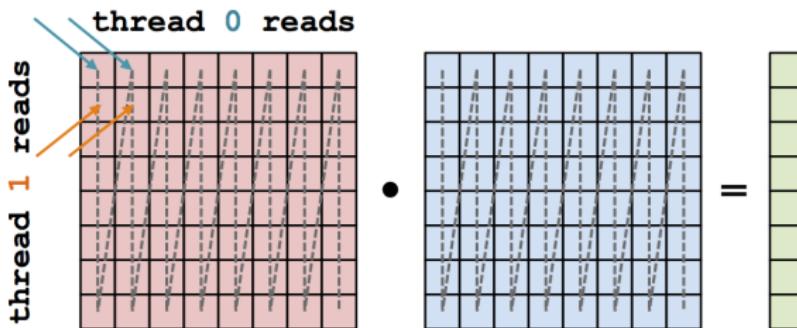


- ▶ **HostSpace:** cached (**good**)
- ▶ **CudaSpace:** uncoalesced (**bad**)

How do we achieve performant memory access?

Kokkos mapping parallel work indices **and** multidimensional array layout *optimally for the architecture.*

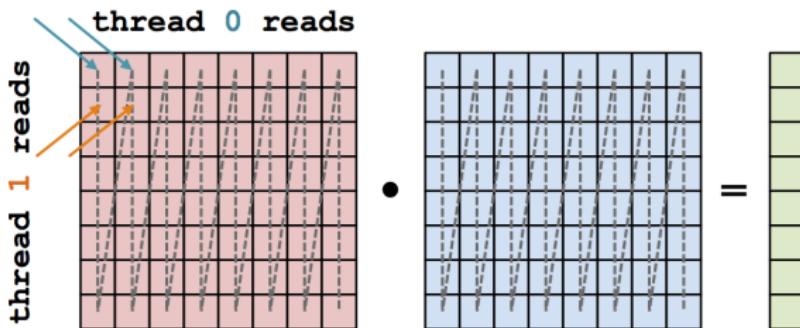
Analysis: column-major (LayoutLeft)



How do we achieve performant memory access?

Kokkos mapping parallel work indices **and** multidimensional array layout *optimally for the architecture.*

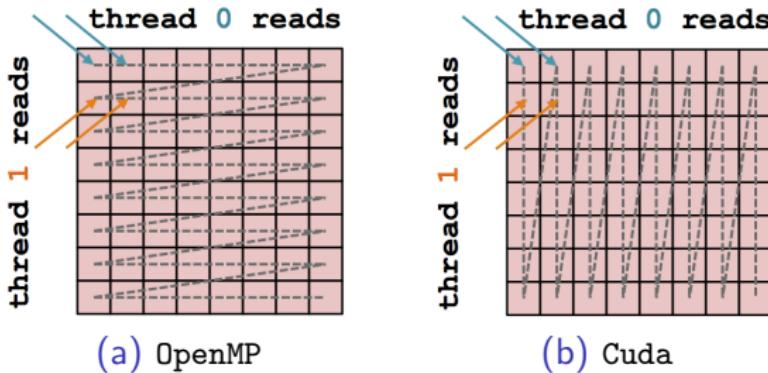
Analysis: column-major (LayoutLeft)



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

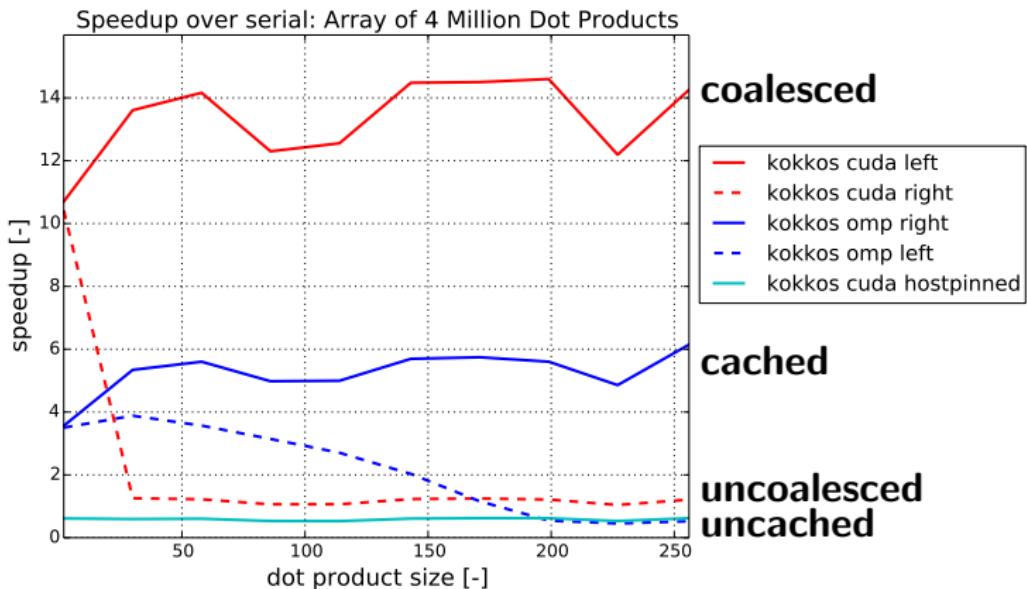
Analysis: Kokkos architecture-dependent

```
View<double**, ExecutionSpace> A(numberOfDots*vectorSize);
View<double**, ExecutionSpace> dotProducts(numberOfDots);
parallel_for(RangePolicy<ExecutionSpace>(0, numberOfDots),
... total += A(dotIndex, i) * B(dotIndex, i);
```

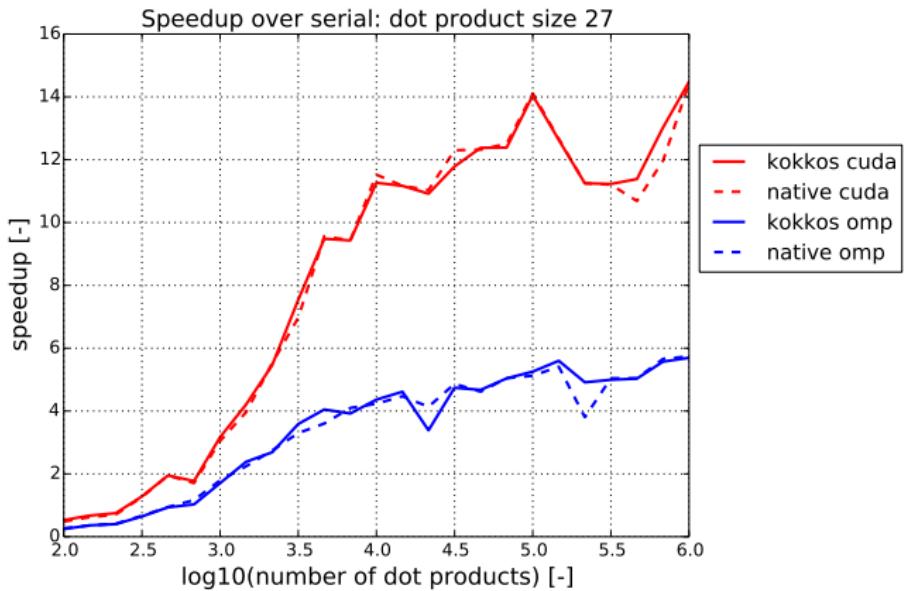


- ▶ **HostSpace:** cached (good)
- ▶ **CudaSpace:** coalesced (good)

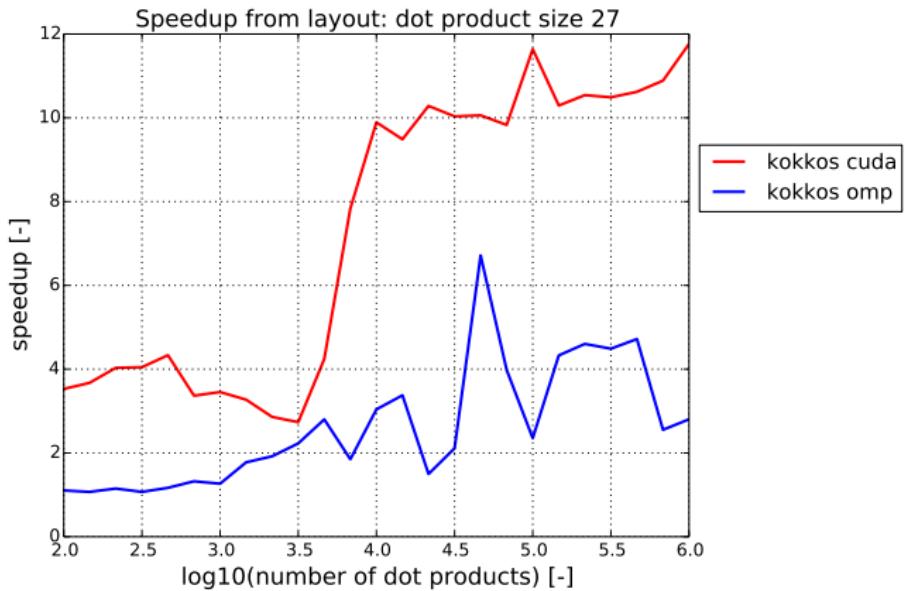
Layout performance, revisited



How is performance a function of data size?



How much does layout matter?



- ▶ Every View has a Layout set at compile-time through a **template parameter**.
- ▶ LayoutRight and LayoutLeft are the **most common** layouts.
- ▶ Views in HostSpace default to LayoutRight and Views in CudaSpace default to LayoutLeft.
- ▶ Layouts are **extensible** and **flexible**.
- ▶ Correct memory access pattern is **essential** for performance.
- ▶ There is **nothing in** OpenMP, OpenACC, or OpenCL to manage layouts.
⇒ You'll need multiple versions of code or pay the performance penalty.

Example: MD force kernel

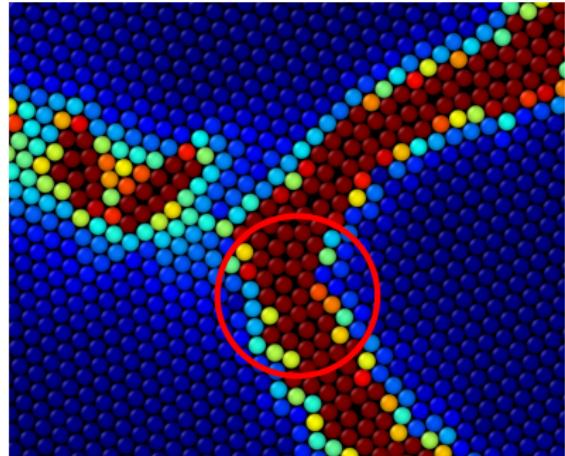
Learning objectives:

- ▶ Practice planning parallelization strategy.
- ▶ Practice using layouts and views.
- ▶ Practice analyzing memory accesses.
- ▶ The RandomAccess memory trait.

High level outline:

```
for every timestep
  for every atom
    force = (0, 0)
    for every neighbor
      forceFromNeighbor = ...
      force += forceFromNeighbor
    atomForces[atom] = force

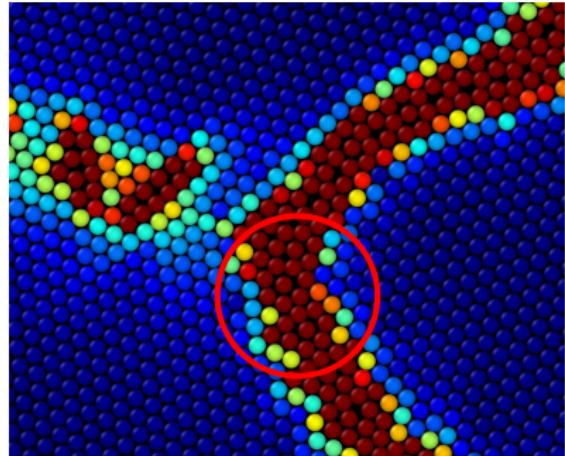
  for every atom
    integrate velocity, position
```



High level outline:

```
for every timestep  
  for every atom  
    force = (0, 0)  
    for every neighbor  
      forceFromNeighbor = ...  
      force += forceFromNeighbor  
    atomForces[atom] = force  
  
  for every atom  
    integrate velocity, position
```

How do we parallelize this?



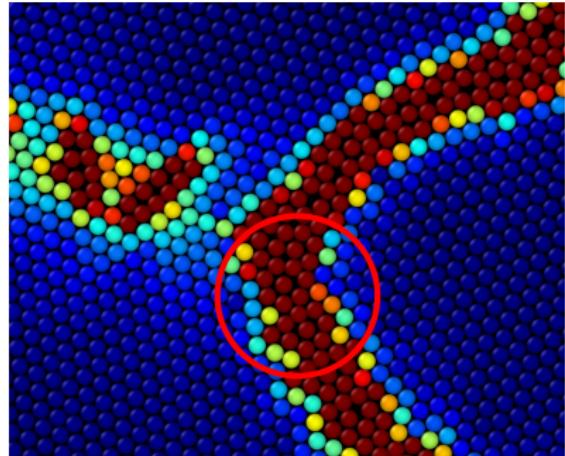
High level outline:

```
for every timestep
  for every atom
    force = (0, 0)
    for every neighbor
      forceFromNeighbor = ...
      force += forceFromNeighbor
    atomForces[atom] = force

  for every atom
    integrate velocity, position
```

How do we parallelize this?

- ▶ Each thread handles a range of atoms

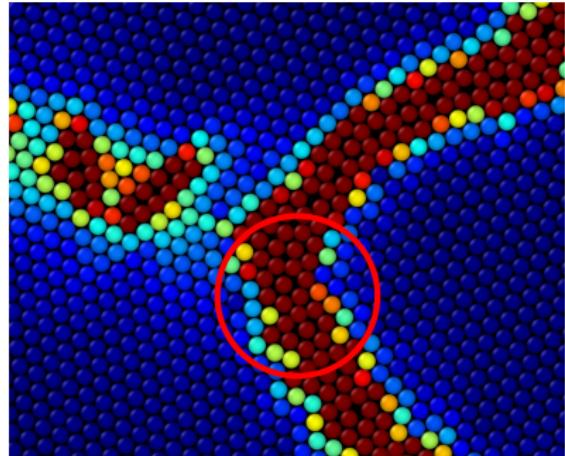


High level outline:

```
for every timestep  
  for every atom  
    force = (0, 0)  
    for every neighbor  
      forceFromNeighbor = ...  
      force += forceFromNeighbor  
    atomForces[atom] = force  
  
  for every atom  
    integrate velocity, position
```

How do we parallelize this?

- ▶ ~~Each thread handles a range of atoms~~
- ▶ Each thread handles an *atom*.
Threads: `numberOfAtoms`

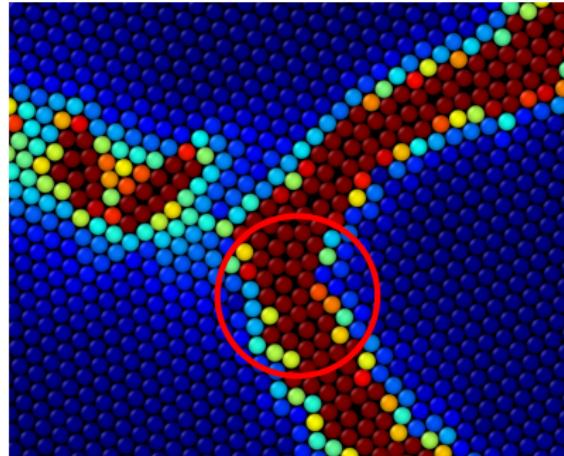


High level outline:

```
for every timestep
    for every atom
        force = (0, 0)
        for every neighbor
            forceFromNeighbor = ...
            force += forceFromNeighbor
        atomForces[atom] = force

    for every atom
        integrate velocity, position
```

How do we parallelize this?



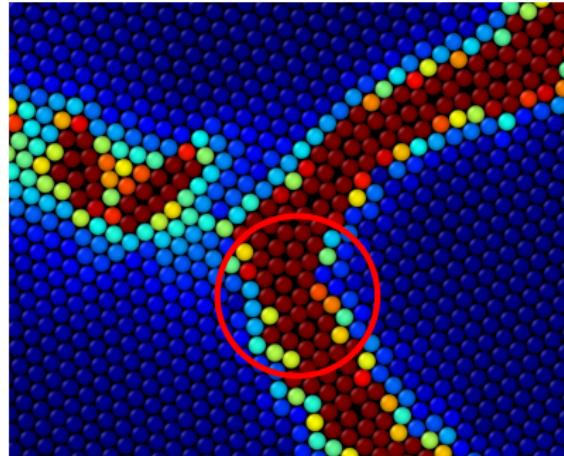
- ▶ ~~Each thread handles a range of atoms~~
- ▶ Each thread handles an *atom*.
Threads: `numberOfAtoms`
- ▶ Each thread handles an *atom-atom pair*.
Threads: `numberOfAtoms * numberOfNeighbors`

High level outline:

```
for every timestep
    for every atom
        force = (0, 0)
        for every neighbor
            forceFromNeighbor = ...
            force += forceFromNeighbor
        atomForces[atom] = force

    for every atom
        integrate velocity, position
```

How do we parallelize this?

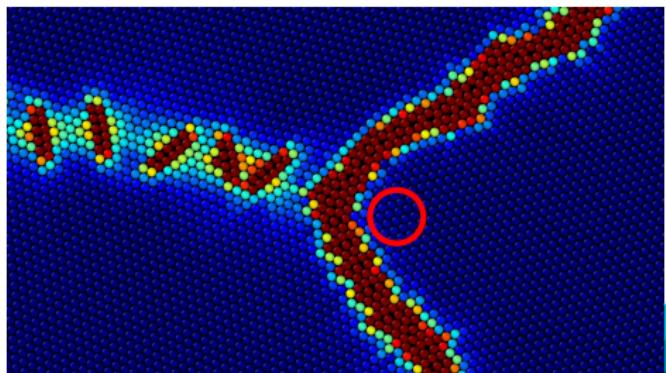


- ▶ ~~Each thread handles a range of atoms~~
- ▶ Each thread handles an *atom*.
Threads: `numberOfAtoms`
- ▶ Each thread handles an *atom-atom pair*.
Threads: `numberOfAtoms * numberOfNeighbors`

Kernel outline:

```
operator()(int atomIndex) {
    atomPosition[0..2] = _positions(atomIndex, 0..2);

    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = _neighborLists(atomIndex, i);
        neighborPosition[0..2] = _positions(neighborIndex, 0..2)
        vecToNeighbor = neighborPosition - atomPosition;
        forceMagnitude = doMaterialModel(vecToNeighbor);
        force += forceMagnitude * vecToNeighbor;
    }
    _forces(atomIndex, 0..2) =
        force[0..2];
}
```



four memory accesses

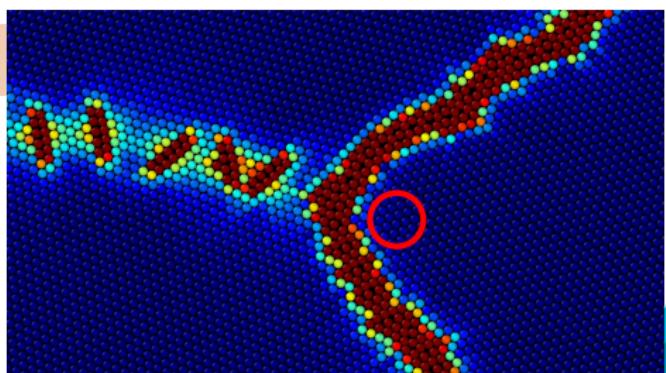
```

operator()(int atomIndex) {
    atomPosition[0..2] = _positions(atomIndex, 0..2);

    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = _neighborLists(atomIndex, i);
        neighborPosition[0..2] = _positions(neighborIndex, 0..2)
        vecToNeighbor = neighborPosition - atomPosition;
        forceMagnitude = doMaterialModel(vecToNeighbor);
        force += forceMagnitude * vecToNeighbor;
    }
    _forces(atomIndex, 0..2) = force[0..2];
}

```

Which of these memory
accesses is the most costly?



four memory accesses

```

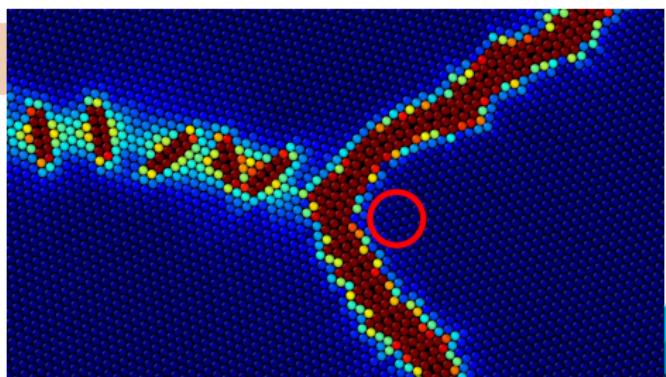
operator()(int atomIndex) {
    atomPosition[0..2] = _positions(atomIndex, 0..2);

    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = _neighborLists(atomIndex, i);
        neighborPosition[0..2] = _positions(neighborIndex, 0..2)
        vecToNeighbor = neighborPosition - atomPosition;
        forceMagnitude = doMaterialModel(vecToNeighbor);
        force += forceMagnitude * vecToNeighbor;
    }
    _forces(atomIndex, 0..2) = force[0..2];
}

```

Which of these memory
accesses is the most costly?

How can we achieve
efficient **random access**?



On **GPUs**, there is a special pathway for fast **read-only, random** access, originally designed for textures.

On **GPUs**, there is a special pathway for fast **read-only, random** access, originally designed for textures.

How to access texture memory via **CUDA**:

```
cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeLinear;
resDesc.res.linear.devPtr = buffer;
resDesc.res.linear.desc.f = cudaChannelFormatKindFloat;
resDesc.res.linear.desc.x = 32; // bits per channel
resDesc.res.linear.sizeInBytes = N*sizeof(float);

cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.readMode = cudaReadModeElementType;

cudaTextureObject_t tex=0;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
```

On **GPUs**, there is a special pathway for fast **read-only, random** access, originally designed for textures.

How to access texture memory via **CUDA**:

```
cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeLinear;
resDesc.res.linear.devPtr = buffer;
resDesc.res.linear.desc.f = cudaChannelFormatKindFloat;
resDesc.res.linear.desc.x = 32; // bits per channel
resDesc.res.linear.sizeInBytes = N*sizeof(float);

cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.readMode = cudaReadModeElementType;

cudaTextureObject_t tex=0;
cudaCreateTextureObject(&tex, &resDesc, &texDesc, NULL);
```

How to access texture memory via **Kokkos**:

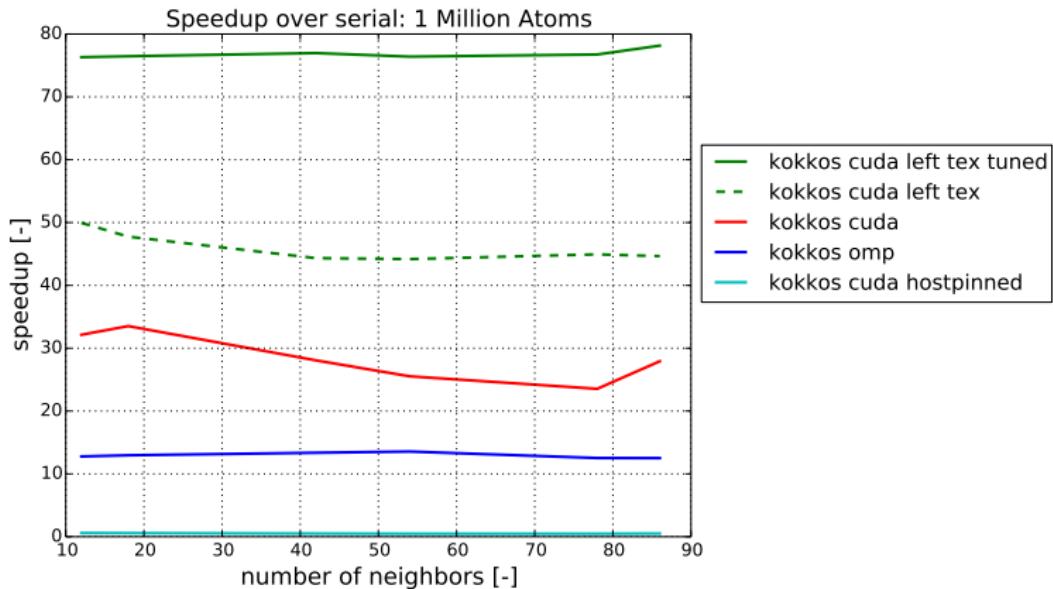
```
View< const double***, Layout, Space,
      MemoryTraits<RandomAccess> > name(...);
```

Task: Implement a Lennard-Jones MD force kernel.

Details:

- ▶ Instructions in `Exercises/MDForceKernel/README`.
- ▶ **New material:** memory trait `RandomAccess`.

Performance for 1 million atoms:



Kokkos supports **subviews** ("slices") of existing Kokkos Views.

- ▶ $B = A(:, 3)$ (FORTRAN notation)

```
View<double**> A("A", N0, N1);
auto B = subview(A, Kokkos::ALL(), 3);
```

- ▶ $B = A(2:4, 3:7)$

```
auto B = subview(A, make_pair(2, 4), make_pair(3, 7));
```

Kokkos supports **subviews** ("slices") of existing Kokkos Views.

- ▶ $B = A(:, 3)$ (FORTRAN notation)

```
View<double**> A("A", N0, N1);
auto B = subview(A, Kokkos::ALL(), 3);
```

- ▶ $B = A(2:4, 3:7)$

```
auto B = subview(A, make_pair(2, 4), make_pair(3, 7));
```

Important point

A subview is a View, possibly with a *strided* layout.

⇒ You use subviews just like ordinary views.

- ▶ Data is stored in Views, **multi-dimensional arrays** residing in **memory spaces** with **architecture-dependent layouts**.
- ▶ **Heterogenous nodes** have one or more memory spaces.
- ▶ Views **abstract away** platform-dependent allocation, (automatic) deallocation, and access.
- ▶ **Mirroring** is used for performant access to views in host and device memory.
- ▶ For performance, memory access patterns must result in **caching** on a CPU and **coalescing** on a GPU.
→ conflicting requirements, unless:
- ▶ Kokkos maps parallel work indices *and* multidimensional array layout for **performance portable memory access patterns**.

Thread safety and atomic operations

Learning objectives:

- ▶ Understand that coordination techniques for low-count CPU threading are not scalable.
- ▶ Understand how atomics can parallelize the **scatter-add** pattern.
- ▶ Gain **performance intuition** for atomics on the CPU and GPU, for different data types and contention rates.
- ▶ Understand a common thread-scalable **array-filling** algorithm.

Kernel outline:

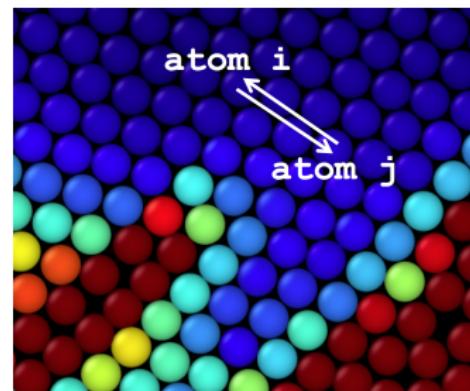
```

operator()(const size_t atomIndex) {
    atomPosition[0..2] = positions(atomIndex, 0..2);
    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = neighborLists(atomIndex, i);
        neighborPosition[0..2] = positions(neighborIndex, 0..2)
        vecToNeighbor = neighborPosition - atomPosition;
        forceMagnitude = doMaterialModel(vecToNeighbor);
        forceToNeighbor[0..2] = forceMagnitude * vecToNeighbor[0..2];
        force[0..2] += forceToNeighbor[0..2];
    }
    forces(atomIndex, 0..2) = force[0..2];
}

```

Duplicated work:

each pair force is calculated twice.



Idea: leverage Newton's third law (action-reaction)

```
operator()(const size_t atomIndex) {
    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = halfNeighborLists(atomIndex, i);
        ...
        forceToNeighbor = forceMagnitude * vectoNeighbor;
        force[0..2]           += forceToNeighbor[0..2];
        forces(neighborIndex, 0..2) -= forceToNeighbor[0..2];
    }
    forces(atomIndex, 0..2) += force[0..2];
}
```

Idea: leverage Newton's third law (action-reaction)

```
operator()(const size_t atomIndex) {
    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = halfNeighborLists(atomIndex, i);
        ...
        forceToNeighbor = forceMagnitude * vectoNeighbor;
        force[0..2]           += forceToNeighbor[0..2];
        forces(neighborIndex, 0..2) -= forceToNeighbor[0..2];
    }
    forces(atomIndex, 0..2) += force[0..2];
}
```

- ▶ This kernel is representative of the general **scatter-add** pattern, where each independent calculation adds its results to many scattered locations.
- ▶ Here we only save half the work; other computations vary (e.g., FEA: save 8 or 27 times the work).

Idea: leverage Newton's third law (action-reaction)

```
operator()(const size_t atomIndex) {  
    force[0..2] = 0.0;  
    for (i = 0; i < maxNumNeighbors; i) {  
        neighborIndex = halfNeighborLists(atomIndex, i);  
        ...  
        forceToNeighbor = forceMagnitude * vectoNeighbor;  
        force[0..2]           += forceToNeighbor[0..2];  
        forces(neighborIndex, 0..2) -= forceToNeighbor[0..2];  
    }  
    forces(atomIndex, 0..2) += force[0..2];  
}
```

Problem:
thread safety

- ▶ This kernel is representative of the general **scatter-add** pattern, where each independent calculation adds its results to many scattered locations.
- ▶ Here we only save half the work; other computations vary (e.g., FEA: save 8 or 27 times the work).

Solutions to thread safety issues: Locks

```
for (i = 0; i < maxNumNeighbors; i) {
    neighborIndex = halfNeighborLists(atomIndex, i);
    ...
    forceToNeighbor = forceMagnitude * vectoNeighbor;
    force[0..2] += forceToNeighbor[0..2];
    // LOCK the lock that protects the neighbor's force
    forces(neighbiorIndex, 0..2) -= forceToNeighbor[0..2];
    // UNLOCK the lock that protects the neighbor's force
}
// LOCK the lock that protects this atom's force
forces(atomIndex, 0..2) += force[0..2];
// UNLOCK the lock that protects this atom's force
```

Solutions to thread safety issues: Locks

```
for (i = 0; i < maxNumNeighbors; i) {  
    neighborIndex = halfNeighborLists(atomIndex, i);  
    ...  
    forceToNeighbor = forceMagnitude * vectoNeighbor;  
    force[0..2] += forceToNeighbor[0..2];  
    // LOCK the lock that protects the neighbor's force  
    forces(neighbiorIndex, 0..2) -= forceToNeighbor[0..2];  
    // UNLOCK the lock that protects the neighbor's force  
}  
// LOCK the lock that protects this atom's force  
forces(atomIndex, 0..2) += force[0..2];  
// UNLOCK the lock that protects this atom's force
```

Problem: contention is too high at O(10,000) threads.

Solutions to thread safety issues: Thread-private copies

```
#pragma omp parallel shared(forces)
{
    ForcesType thisThreadsForces(forces.size())
#pragma omp for nowait
    for each atom {
        for each neighbor {
            ...
            forceToNeighbor = forceMagnitude * vectoNeighbor;
            force[0..2] += forceToNeighbor[0..2];
            thisThreadsForces(neighborIndex, 0..2) -=
                forceToNeighbor[0..2];
        }
    }
#pragma omp critical
    for each atom {
        forces[atomIndex] += thisThreadsForces[atomIndex];
    }
}
```

Solutions to thread safety issues: Thread-private copies

```
#pragma omp parallel shared(forces)
{
    ForcesType thisThreadsForces(forces.size())
#pragma omp for nowait
    for each atom {
        for each neighbor {
            ...
            forceToNeighbor = forceMagnitude * vectoNeighbor;
            force[0..2] += forceToNeighbor[0..2];
            thisThreadsForces(neighborIndex, 0..2) -=
                forceToNeighbor[0..2];
        }
    }
#pragma omp critical
    for each atom {
        forces[atomIndex] += thisThreadsForces[atomIndex];
    }
}
```

Problems: insufficient memory for `thisThreadsForces`
ratio of parallel/serial work too low.

Solutions to thread safety issues: Atomics

```
operator()(const size_t atomIndex) {
    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = halfNeighborLists(atomIndex, i);
        ...
        forceToNeighbor = forceMagnitude * vecToNeighbor;
        force[0..2] += forceToNeighbor[0..2];
        Kokkos::atomic_sub(&forces(neighborIndex, 0..2),
                           forceToNeighbor[0..2]);
    }
    Kokkos::atomic_add(&forces(atomIndex, 0..2), force[0..2]);
}
```

Solutions to thread safety issues: Atomics

```
operator()(const size_t atomIndex) {
    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = halfNeighborLists(atomIndex, i);
        ...
        forceToNeighbor = forceMagnitude * vecToNeighbor;
        force[0..2] += forceToNeighbor[0..2];
        Kokkos::atomic_sub(&forces(neighborIndex, 0..2),
                           forceToNeighbor[0..2]);
    }
    Kokkos::atomic_add(&forces(atomIndex, 0..2), force[0..2]);
}
```

- ▶ Atomics are the **only scalable** solution to thread safety.

Solutions to thread safety issues: Atomics

```
operator()(const size_t atomIndex) {
    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = halfNeighborLists(atomIndex, i);
        ...
        forceToNeighbor = forceMagnitude * vecToNeighbor;
        force[0..2] += forceToNeighbor[0..2];
        Kokkos::atomic_sub(&forces(neighborIndex, 0..2),
                           forceToNeighbor[0..2]);
    }
    Kokkos::atomic_add(&forces(atomIndex, 0..2), force[0..2]);
}
```

- ▶ Atomics are the **only scalable** solution to thread safety.
- ▶ Locks or data replication are **strongly discouraged**.

Solutions to thread safety issues: Atomics

```
operator()(const size_t atomIndex) {
    force[0..2] = 0.0;
    for (i = 0; i < maxNumNeighbors; i) {
        neighborIndex = halfNeighborLists(atomIndex, i);
        ...
        forceToNeighbor = forceMagnitude * vecToNeighbor;
        force[0..2] += forceToNeighbor[0..2];
        Kokkos::atomic_sub(&forces(neighborIndex, 0..2),
                           forceToNeighbor[0..2]);
    }
    Kokkos::atomic_add(&forces(atomIndex, 0..2), force[0..2]);
}
```

- ▶ Atomics are the **only scalable** solution to thread safety.
- ▶ Locks or data replication are **strongly discouraged**.
- ▶ If all accesses to a View will be atomic, use the **MemoryTrait**:

```
View<double**, Layout, Space,
      MemoryTraits<Atomic> > forces(...);
```

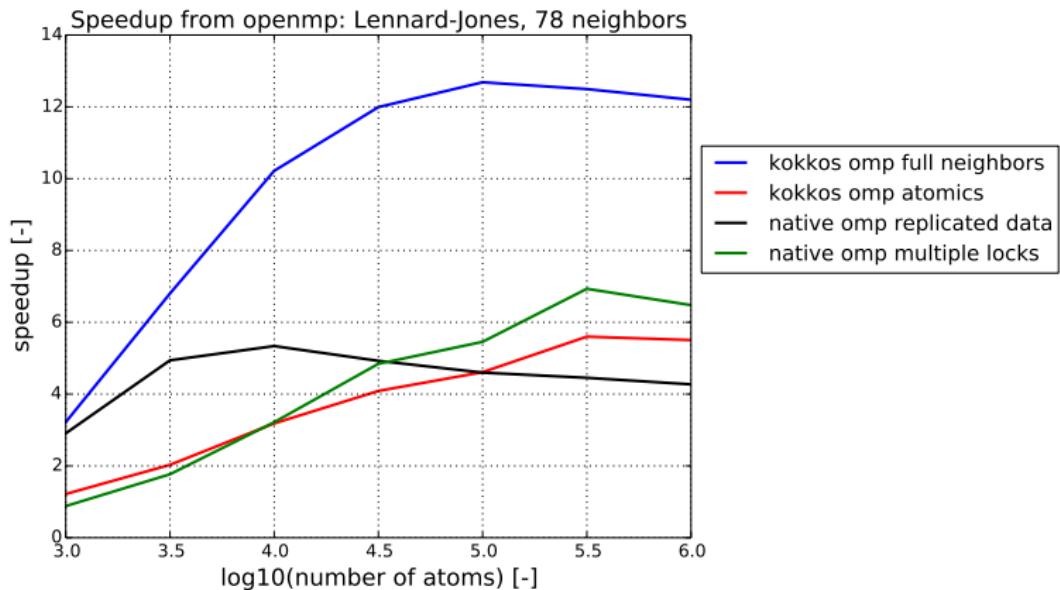
Exercise: Modify the Lennard-Jones MD force kernel to use atomics.

TODO: Christian, are they modifying or making a new one? If making a new one, then we can use half neighbor lists.

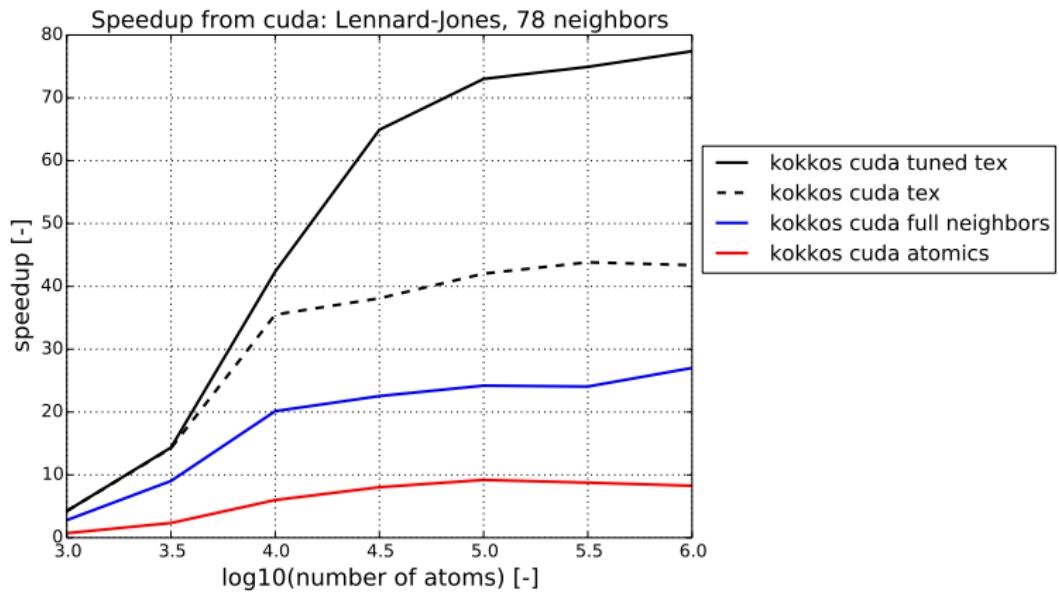
Details:

- ▶ You'll need to exclude neighbors from the full neighbor list.
- ▶ Instructions in Exercises/TODO: Christian/README.
- ▶ **New material:** atomics.

Atomics performance: OpenMP



Atomics performance: Cuda



How expensive are atomics?

Thought experiment: scalar integration

```
operator()(const unsigned int intervalIndex,
           double & valueToUpdate) const {
    double contribution = function(...);
    valueToUpdate += contribution;
}
```

How expensive are atomics?

Thought experiment: scalar integration

```
operator()(const unsigned int intervalIndex,
           double & valueToUpdate) const {
    double contribution = function(...);
    valueToUpdate += contribution;
}
```

Idea: what if we instead do this with parallel_for and atomics?

```
operator()(const unsigned int intervalIndex) const {
    const double contribution = function(...);
    Kokkos::atomic_add(&globalSum, contribution);
}
```

How much of a performance penalty is incurred?

Two costs: (independent) work and coordination.

```
parallel_reduce(numberOfIntervals,
    KOKKOS_LAMBDA (const unsigned int intervalIndex,
                    double & valueToUpdate) {
    valueToUpdate += function(...);
}, totalIntegral);
```

Two costs: (independent) work and coordination.

```
parallel_reduce(numberOfIntervals,
    KOKKOS_LAMBDA (const unsigned int intervalIndex,
                    double & valueToUpdate) {
        valueToUpdate += function(...);
    }, totalIntegral);
```

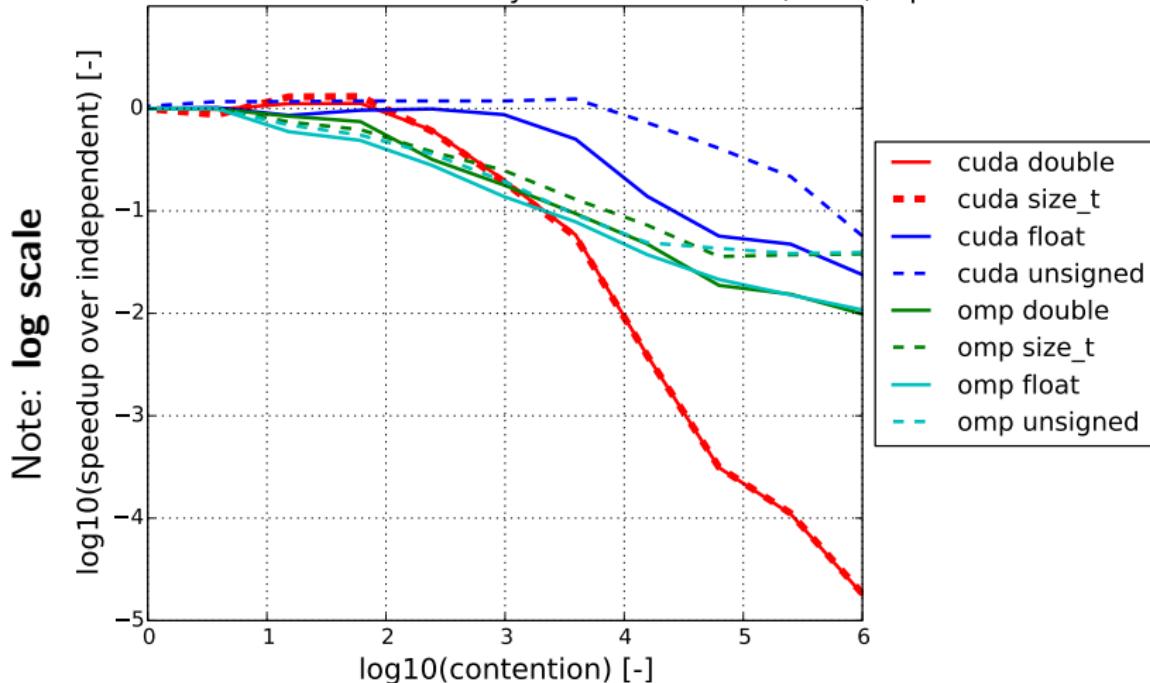
Experimental setup

```
operator()(const unsigned int index) const {
    Kokkos::atomic_add(&globalSums[index % atomicStride], 1);
}
```

- ▶ This is the most extreme case: all coordination and no work.
- ▶ Contention is captured by the `atomicStride`.
 - $\text{atomicStride} \rightarrow 1 \Rightarrow \text{Scalar integration}$
 - $\text{atomicStride} \rightarrow \text{large} \Rightarrow \text{Independent}$

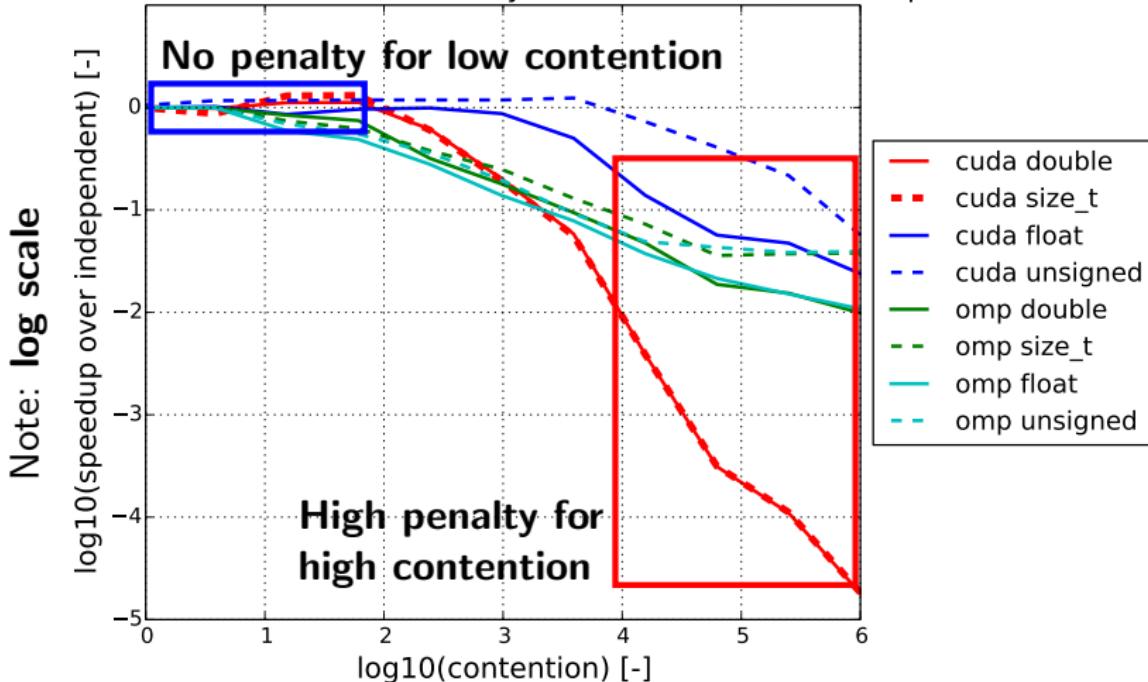
Atomics performance: largest number of adds

Slowdown from atomics: Summary for 1 million adds, mod, 0 pows

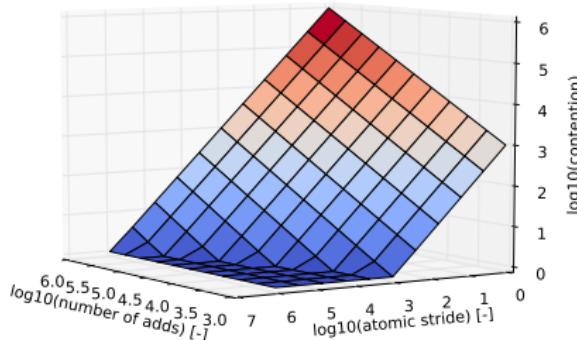


Atomics performance: largest number of adds

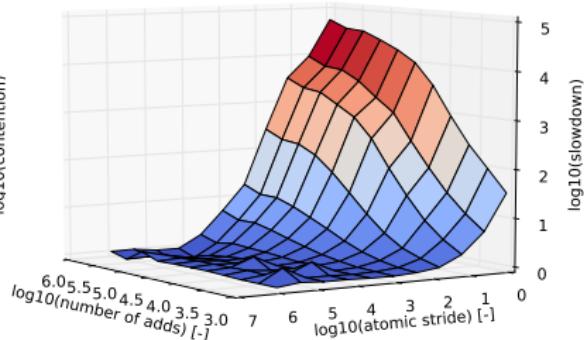
Slowdown from atomics: Summary for 1 million adds, mod, 0 pows



Cuda: How is performance a function of data size?

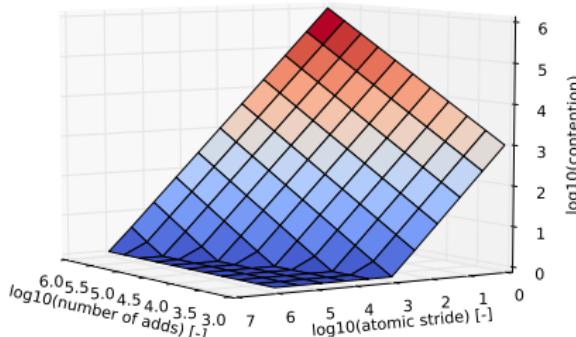


(a) Contention

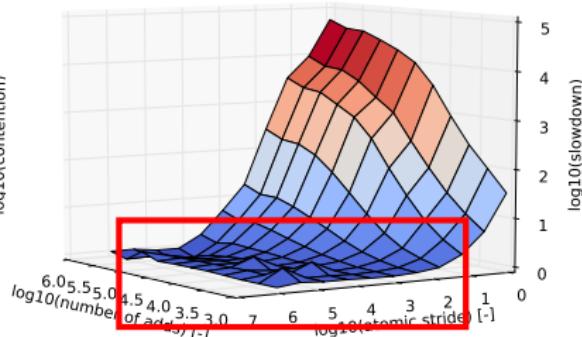


(b) Cuda double

Cuda: How is performance a function of data size?



(c) Contention

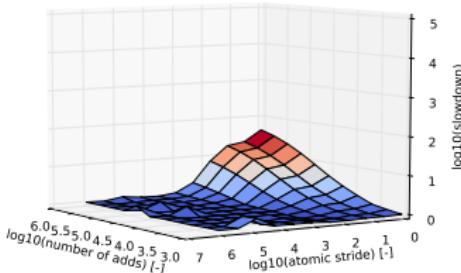


(d) Cuda double

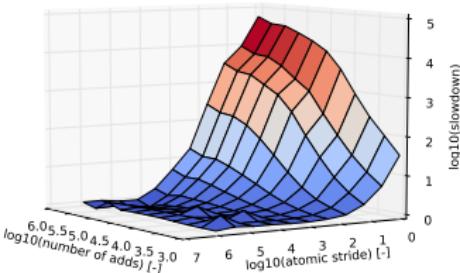
Important point

Cuda atomics can be free at moderate contention and scattered access; e.g., *scatter-add* pattern.

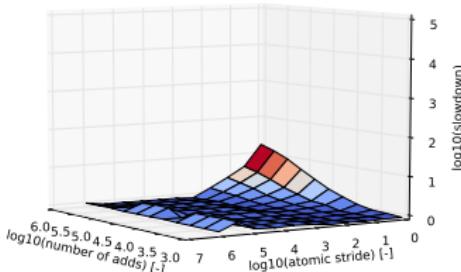
Cuda: How is performance a function of data type?



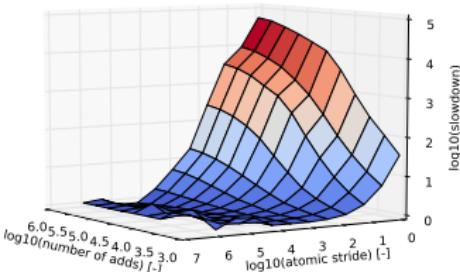
(a) float



(b) double

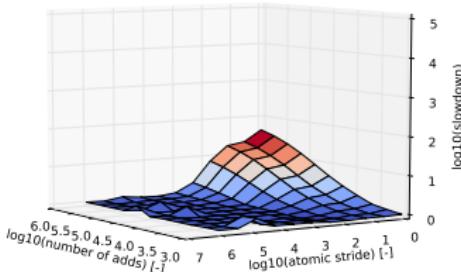


(c) unsigned

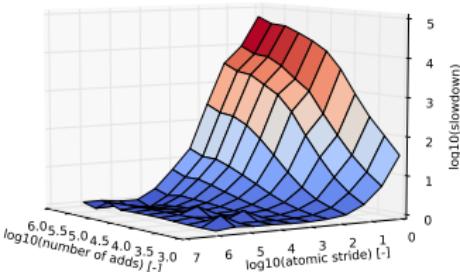


(d) size_t

Cuda: How is performance a function of data type?

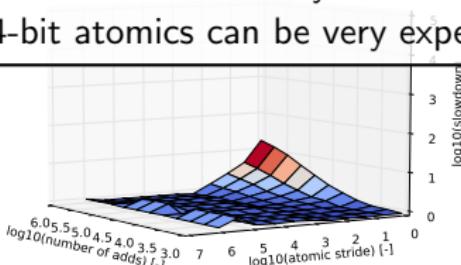


(e) float

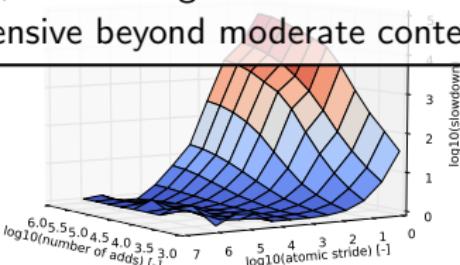


(f) double

- ▶ 32-bit atomics are very efficient, even at high contention.
- ▶ 64-bit atomics can be very expensive beyond moderate contention.

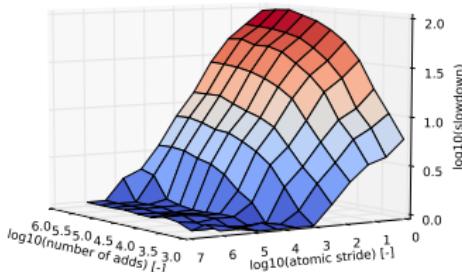


(g) unsigned

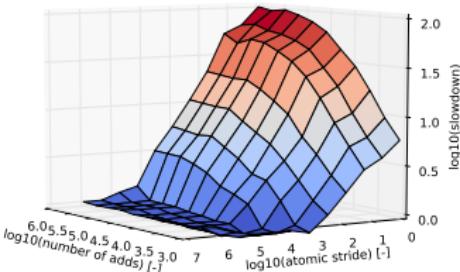


(h) size_t

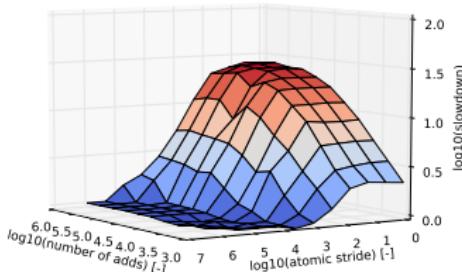
OpenMP: How is performance a function of type and size?



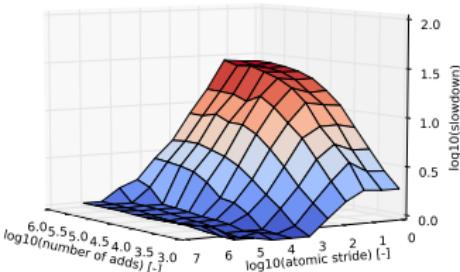
(a) float



(b) double

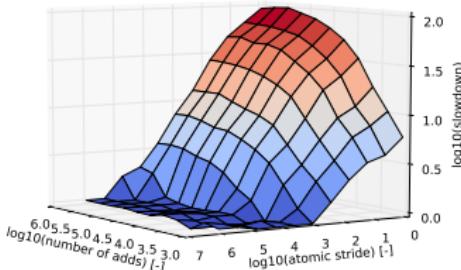


(c) unsigned

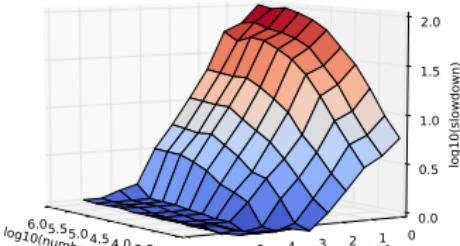


(d) size_t

OpenMP: How is performance a function of type and size?

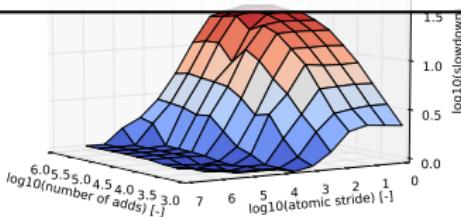


(e) float

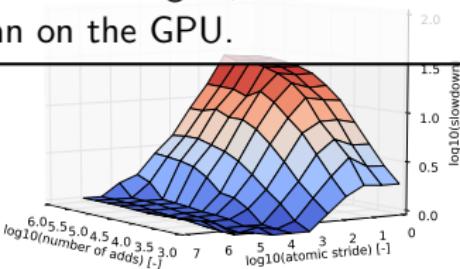


(f) double

- ▶ Floating-point is more expensive than integers, size is irrelevant.
- ▶ Atomics are less “forgiving” than on the GPU.



(g) unsigned



(h) size_t

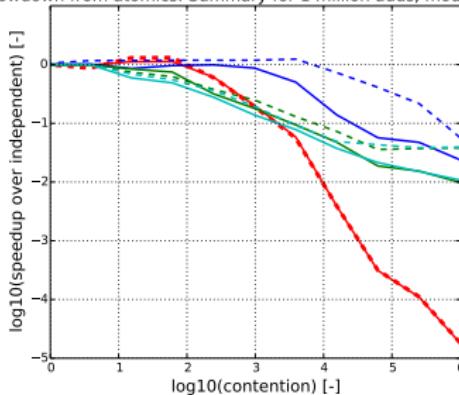
Contiguous instead of scattered: largest number of adds

```
Kokkos::atomic_add(&globalSums[index / atomicStride], 1);
```

Contiguous instead of scattered: largest number of adds

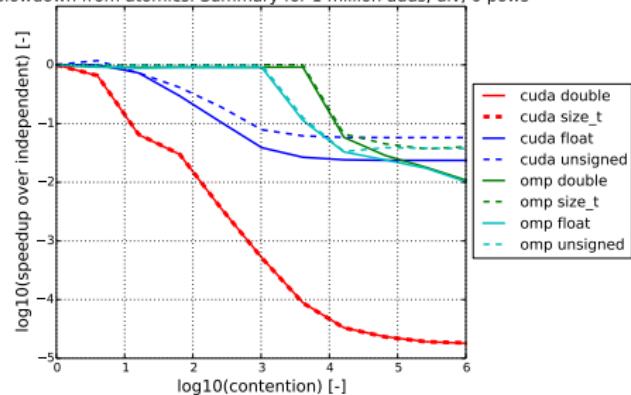
```
Kokkos::atomic_add(&globalSums[index / atomicStride], 1);
```

Slowdown from atomics: Summary for 1 million adds, mod



(c) Scattered (mod)

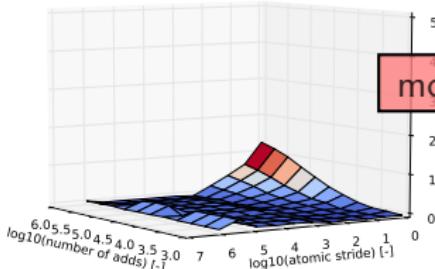
Slowdown from atomics: Summary for 1 million adds, div, 0 pows



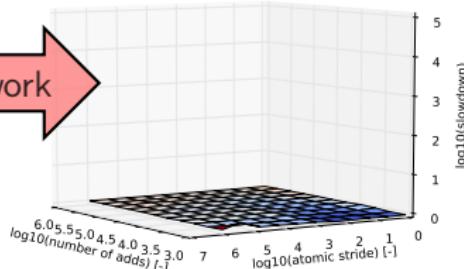
(d) Contiguous (div)

OpenMP performance improves and Cuda performance degrades significantly.

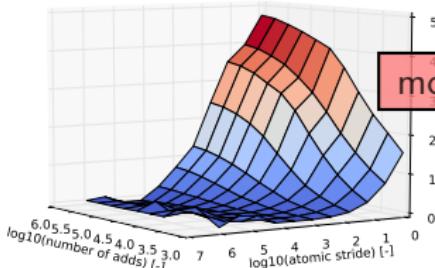
Kernel with work: 10 calls to pow per add, Cuda



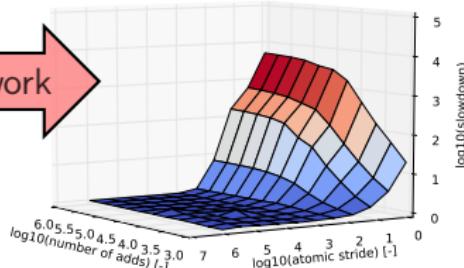
(a) unsigned, 0 pows



(b) unsigned, 10 pows

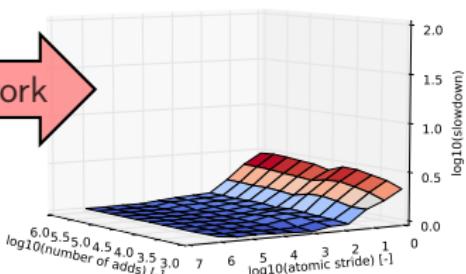
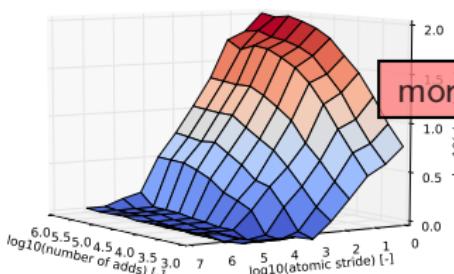
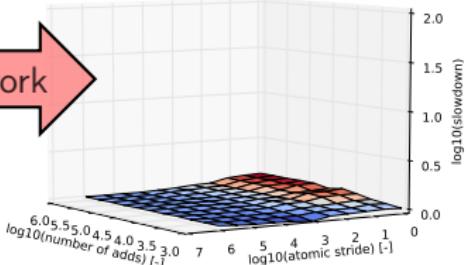
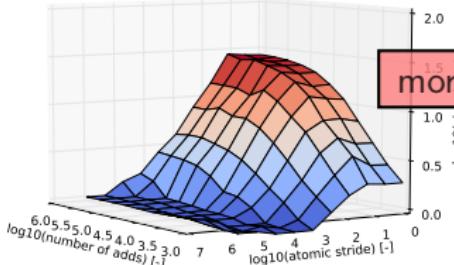


(c) size_t, 0 pows



(d) size_t, 10 pows

Kernel with work: 10 calls to pow per add, OpenMP



Summary:

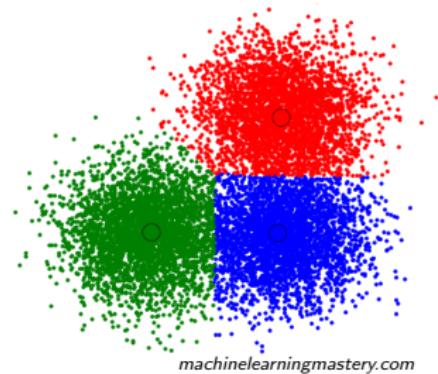
- ▶ Atomic performance **depends on ratio** of independent work and atomic operations.
 - ▶ With more work, there is a lower performance penalty, because of increased opportunity to interleave work and atomic.
- ▶ **Contiguous** atomics work better on the **CPU**,
Scattered atomics work better on the **GPU**.
- ▶ Scatter-atomic-add *can have negligible cost* for low contention ($<\sim 100$) and non-trivial work.
- ▶ CPU: integers perform better than floating-point.
- ▶ GPU: 32-bit performs *significantly* better than 64-bit.
 - ▶ Note: 64-bit atomics are planned for future architectures.

Exercise: k-means clustering

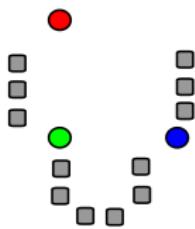
Learning objectives:

- ▶ Practice atomics in a machine-learning technique.
- ▶ Observe performance in a kernel with realistic work.

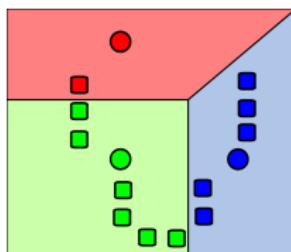
k-means clustering is a technique to classify noisy data from a discrete number of sources.



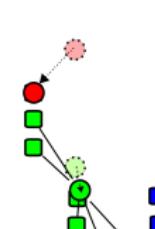
machinelearningmastery.com



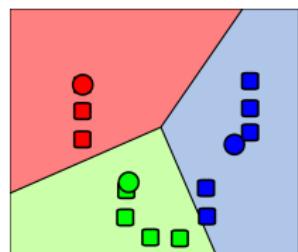
(a) Step 1



(b) Step 2



(c) Step 3



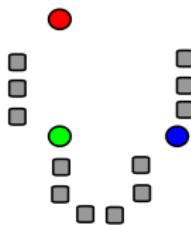
(d) Step 4

[wikipedia](https://en.wikipedia.org)

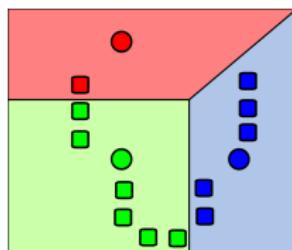
Exercise: Implement the k-means clustering example.

Details:

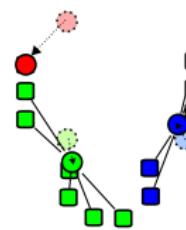
- ▶ Instructions in Exercises/KMeansClustering/README.



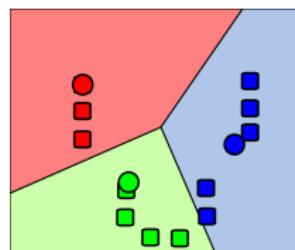
(e) Step 1



(f) Step 2



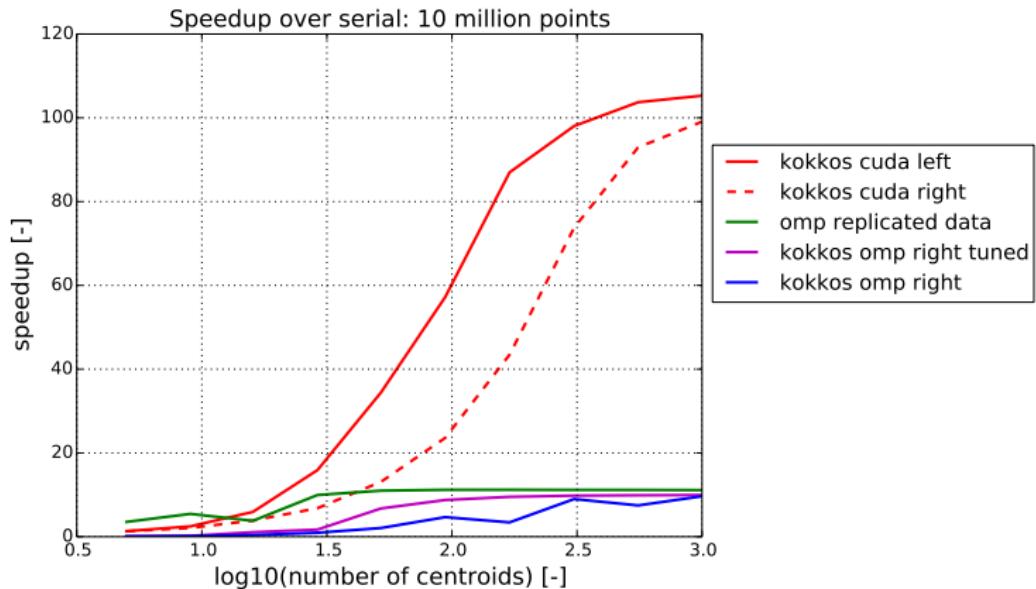
(g) Step 3



(h) Step 4

wikipedia

Performance summary: 10 million points



Example: array-fill

Learning objectives:

- ▶ Understand a common pattern used to fill communication buffers and grow arrays.
- ▶ See motivation for thread teams.

Scenario: filling an array in parallel

```
for each atom {  
    atomPosition += integrateVelocity(...);  
    if (atomPosition is near subdomain boundary) {  
        positionsToSend.push_back(atomPosition);  
    }  
}
```

Scenario: filling an array in parallel

```
for each atom {  
    atomPosition += integrateVelocity(...);  
    if (atomPosition is near subdomain boundary) {  
        positionsToSend.push_back(atomPosition);  
    }  
}
```

(unscalable) Parallelization technique: **Locks**

```
for each atom {  
    atomPosition += integrateVelocity(...);  
    if (atomPosition is near subdomain boundary) {  
        // LOCK the lock  
        positionsToSend.push_back(atomPosition);  
        // UNLOCK the lock  
    }  
}
```

(unscalable) Parallelization technique: **Thread-private copies**

```
#pragma omp parallel
{
    PositionsType thisThreadsPositionsToSend(forces.size())
#pragma omp for nowait
    for each atom {
        atomPosition += integrateVelocity(...);
        if (atomPosition is near subdomain boundary) {
            thisThreadsPositionsToSend.push_back(atomPosition);
        }
    }
#pragma omp critical
    positionsToSend.push_back(thisThreadsPositionsToSend);
}
```

Parallelization technique: **atomic claims on a location**

```
operator()(const unsigned int atomIndex) const {
    atomPosition += integrateVelocity(...);
    if (atomPosition is near subdomain boundary) {
        const unsigned int claimedIndex =
            Kokkos::atomic_fetch_add(&globalCount, 1);
        positionsToSend[claimedIndex] = atomPosition;
    }
}
```

Parallelization technique: **atomic claims on a location**

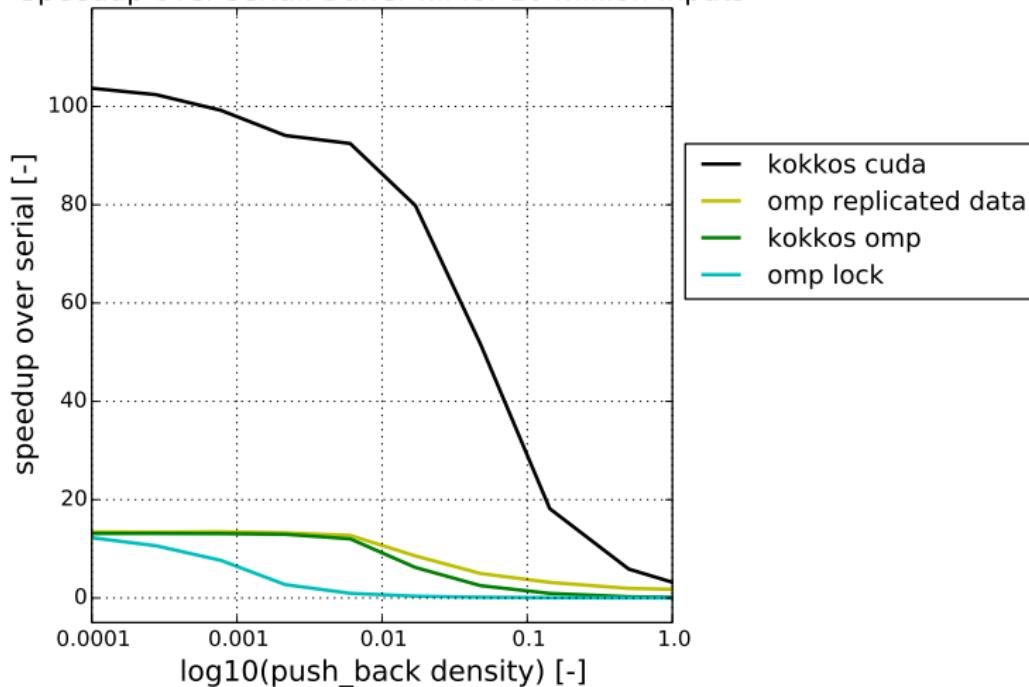
```
operator()(const unsigned int atomIndex) const {
    atomPosition += integrateVelocity(...);
    if (atomPosition is near subdomain boundary) {
        const unsigned int claimedIndex =
            Kokkos::atomic_fetch_add(&globalCount, 1);
        positionsToSend[claimedIndex] = atomPosition;
    }
}
```

Performance test:

```
operator()(const unsigned int atomIndex) const {
    if (atomIndex % atomicStride == 0) {
        const unsigned int claimedIndex =
            Kokkos::atomic_fetch_add(&globalCount, 1);
        positionsToSend[claimedIndex] = atomPosition;
    }
}
```

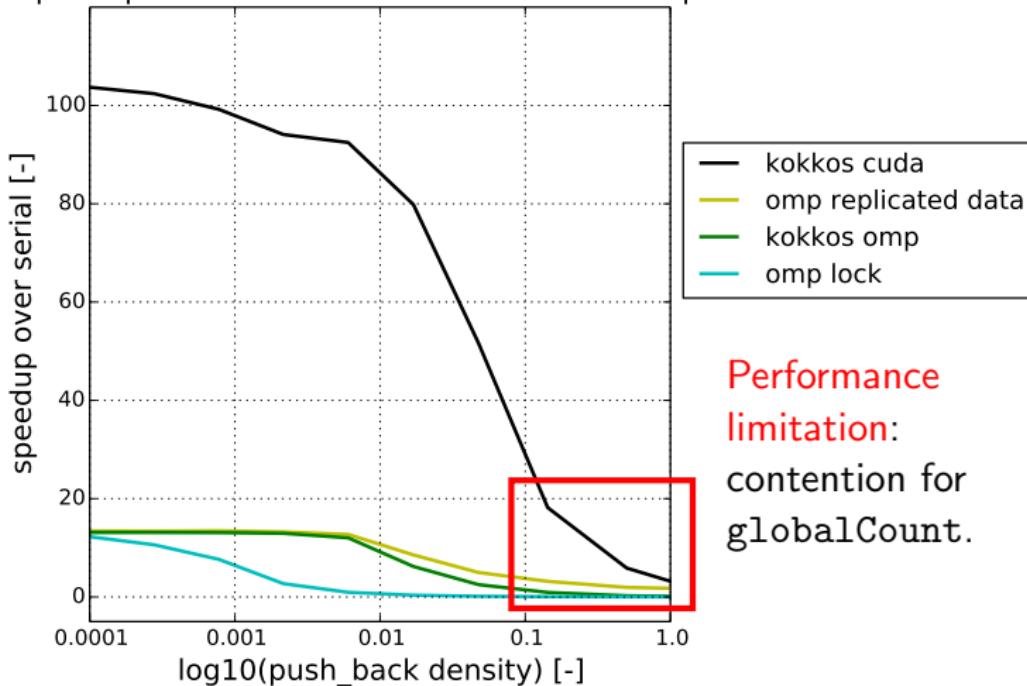
ArrayFill performance:

Speedup over serial: Buffer fill for 10 million inputs



ArrayFill performance:

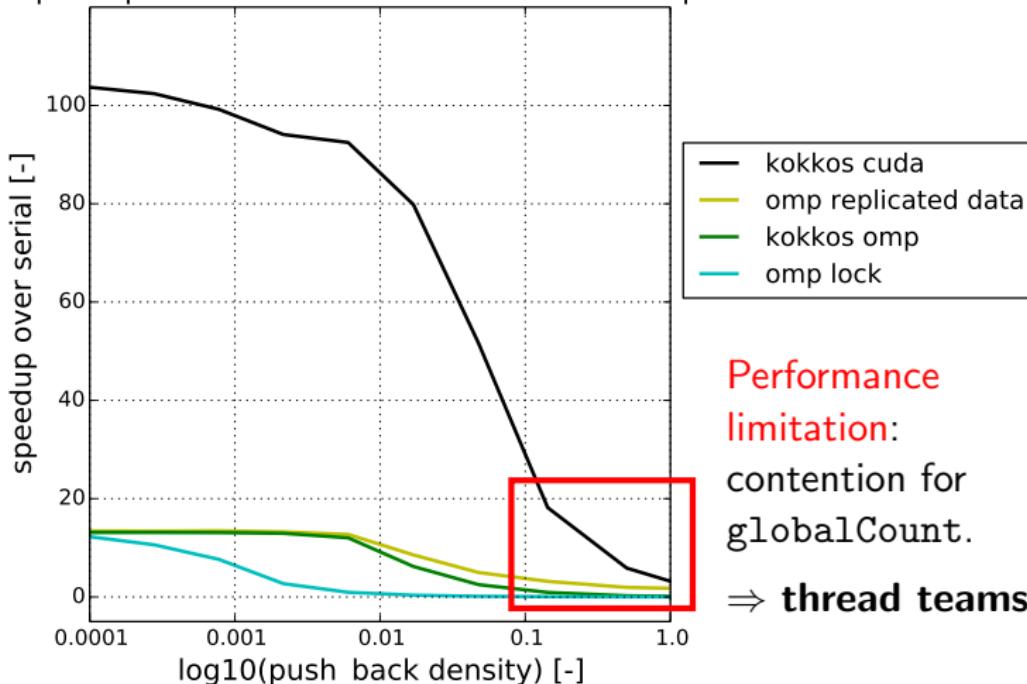
Speedup over serial: Buffer fill for 10 million inputs



Performance
limitation:
contention for
globalCount.

ArrayFill performance:

Speedup over serial: Buffer fill for 10 million inputs



Atomics on arbitrary types:

- ▶ Atomic add works on any word-aligned data type with “`+ =`”.
- ▶ Atomic exchange works on any word-aligned data type.

```
// Assign *dest to val, return former value of *dest
template< typename T >
T atomic_exchange( T * dest , T val );
// If *dest == comp then assign *dest to val
// Return true if succeeds.
template< typename T >
bool atomic_compare_exchange_strong( T * dest , T comp , T val );
```

Atomics on arbitrary types:

- ▶ Atomic add works on any word-aligned data type with “`+ =`”.
- ▶ Atomic exchange works on any word-aligned data type.

```
// Assign *dest to val, return former value of *dest
template< typename T >
T atomic_exchange( T * dest , T val );
// If *dest == comp then assign *dest to val
// Return true if succeeds.
template< typename T >
bool atomic_compare_exchange_strong( T * dest , T comp , T val );
```

A note on C++11 std::atomic<T>:

- ▶ Does not allow non-atomic access ⇒ performance penalty
- ▶ Cannot cast between T and atomic<T>
- ▶ Fix is proposed for C++17

- ▶ Atomics are the only thread-scalable solution to thread safety.
Locks or data replication are **strongly discouraged**
- ▶ Many programs with the **scatter-add** pattern can be parallelized using atomics without much modification.
- ▶ The Atomic **memory trait** can be used to make all accesses to a view atomic.
- ▶ The cost of atomics can be negligible:
 - CPU** ideal: contiguous access, integer types
 - GPU** ideal: scattered access, 32-bit types
 - Given **non-trivial work**, atomics are very inexpensive

- ▶ Atomics are the only thread-scalable solution to thread safety.
Locks or data replication are **strongly discouraged**
- ▶ Many programs with the **scatter-add** pattern can be parallelized using atomics without much modification.
- ▶ The Atomic **memory trait** can be used to make all accesses to a view atomic.
- ▶ The cost of atomics can be negligible:
 - CPU** ideal: contiguous access, integer types
 - GPU** ideal: scattered access, 32-bit types
 - Given **non-trivial work**, atomics are very inexpensive
- ▶ *Might* be able to get by on Trinity (ATS-1) without atomics
- ▶ *Cannot* get by on Sierra (ATS-2) without atomics

Hierarchical parallelism

Finding and exploiting more parallelism in your computations.

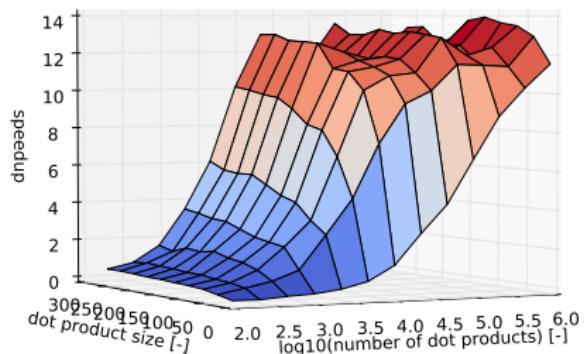
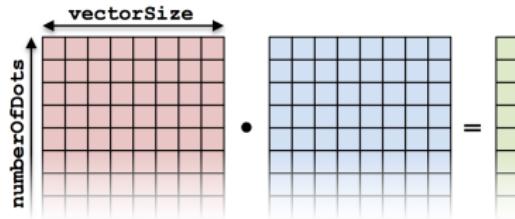
Learning objectives:

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

Kernel: (note the inner loop, make it parallel!)

```
operator()(const size_t dotIndex) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
        total += A(dotIndex, i) * B(dotIndex, i);
    }
    dotProducts(dotIndex) = total; });

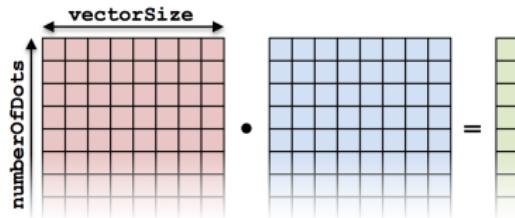
```



Kernel: (note the inner loop, make it parallel!)

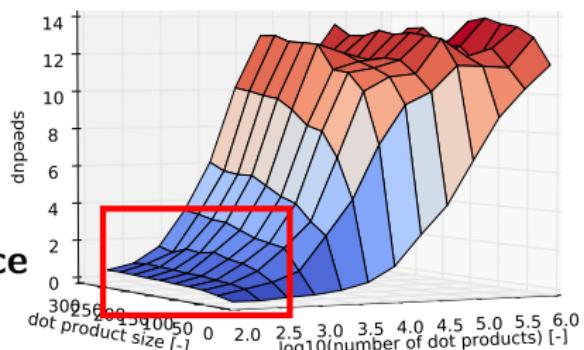
```
operator()(const size_t dotIndex) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
        total += A(dotIndex, i) * B(dotIndex, i);
    }
    dotProducts(dotIndex) = total; });

```



Poor performance

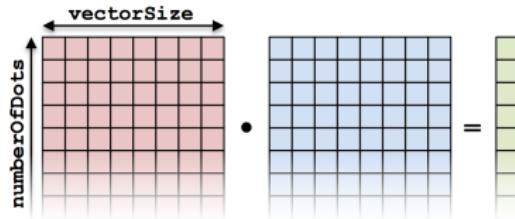
Solutions?



Kernel: (note the inner loop, make it parallel!)

```
operator()(const size_t dotIndex) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
        total += A(dotIndex, i) * B(dotIndex, i);
    }
    dotProducts(dotIndex) = total; });

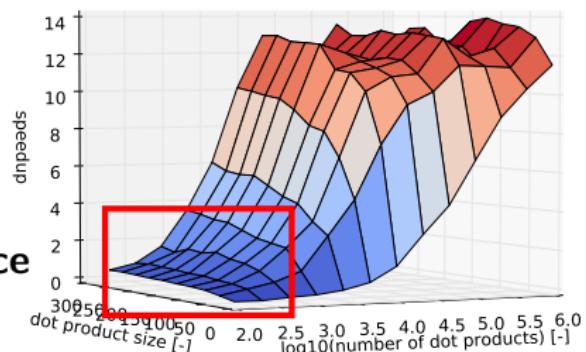
```



Poor performance

Solutions?

- ▶ Atomics
- ▶ Thread teams



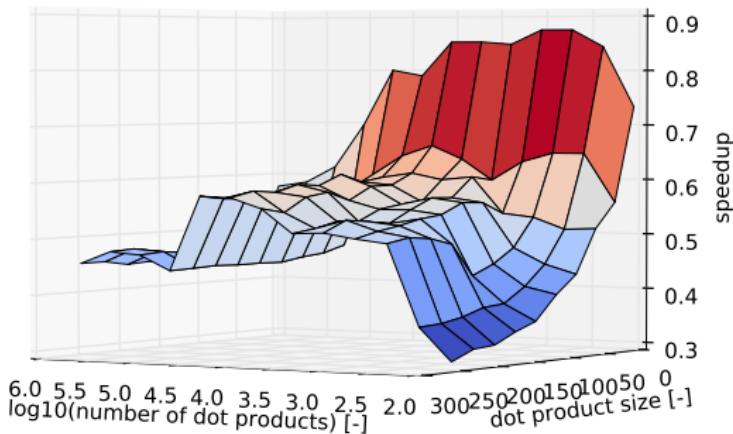
Atomics kernel: (prefer a 2D range policy, *is tbd*)

```
operator()(const size_t index) {
    int dotIndex = extractDotIndex(index);
    int entryIndex = extractEntryIndex(index);
    atomic_add(&_dotProducts(dotIndex),
               A(dotIndex, entryIndex) * B(dotIndex, entryIndex));
}
```

Atomics kernel: (prefer a 2D range policy, *is tbd*)

```
operator()(const size_t index) {
    int dotIndex = extractDotIndex(index);
    int entryIndex = extractEntryIndex(index);
    atomic_add(&_dotProducts(dotIndex),
               A(dotIndex, entryIndex) * B(dotIndex, entryIndex));
}
```

Speedup from using atomics on cuda:



Doing each individual dot product with atomics is like doing scalar integration with atomics.

Instead, you could envision doing a large number of parallel_reduce kernels.

```
for each dot product
    DotProductFunctor functor(dotProductIndex, ...);
    parallel_reduce(
        RangePolicy<ExecutionSpace>(0, dotProductSize), functor);
}
```

Doing each individual dot product with atomics is like doing scalar integration with atomics.

Instead, you could envision doing a large number of parallel_reduce kernels.

```
for each dot product
    DotProductFunctor functor(dotProductIndex, ...);
    parallel_reduce(
        RangePolicy<ExecutionSpace>(0, dotProductSize), functor);
}
```

This is an example of *hierarchical work*.

Important concept: Hierarchical parallelism

Algorithms that exhibit hierarchical structure can exploit hierarchical parallelism with **thread teams**.

Important concept: Thread team

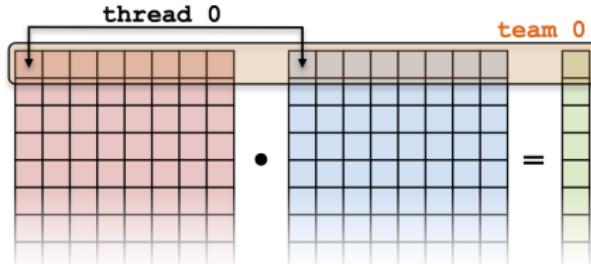
A collection of threads which are guaranteed to be executing **concurrently** and **can synchronize**.

Important concept: Thread team

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

High-level strategy:

1. Do **one parallel launch** of `numberOfDotProducts` teams of `dotProductSize` threads.
2. Each thread performs **one multiplication** of an entry in the dot product.
3. The threads within **teams perform a reduction**.



The final hierarchical parallel kernel:

```
parallel_for(
    TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize),
    KOKKOS_LAMBDA (member_type teamMember) {
        int dotIndex = teamMember.league_rank();
        double dotProduct = 0 ;
        parallel_reduce(TeamThreadRange(teamMember, dotProductSize),
            [=] (int i, double & valueToUpdate) {
                valueToUpdate += data_A(dotIndex,i) * data_B(dotIndex,i);
            }, dotProduct);
        if (teamMember.team_rank() == 0) {
            _dotProducts[dotIndex] = dotProduct;
        }
    }
}
```

The **performance** and **flexibility** of teams is *naturally* and *concisely* expressed under the Kokkos model.

Let's walk through how we got to this *final* answer.

Important point

Using teams is changing the execution *policy*.

“Flat parallelism” uses RangePolicy:

We specify a *total amount of work*.

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

Important point

Using teams is changing the execution *policy*.

“**Flat** parallelism” uses RangePolicy:

We specify a *total amount of work*.

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

“**Hierarchical** parallelism” uses TeamPolicy:

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

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

Important point

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

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

void operator()(const member_type & teamMember) {
    // Which team am I on?
    const unsigned int leagueRank = teamMember.league_rank();
    // Which thread am I on this team?
    const unsigned int teamRank = teamMember.team_rank();
}
```

Important point

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

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

void operator()(const member_type & teamMember) {
    // Which team am I on?
    const unsigned int leagueRank = teamMember.league_rank();
    // Which thread am I on this team?
    const unsigned int teamRank = teamMember.team_rank();
}
```

Warning

There may be more (or fewer) team members than pieces of work per team

First attempt at array of dot products:

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex      = teamMember.league_rank();
    const unsigned int entryIndex   = teamMember.team_rank();
    atomicDotProducts[dotIndex] +=  
        data_A(dotIndex, entryIndex) * data_B(dotIndex, entryIndex);
}
```

First attempt at array of dot products:

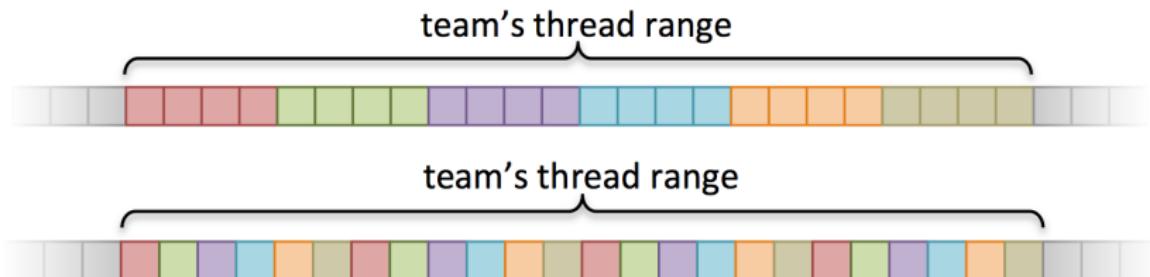
```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex      = teamMember.league_rank();
    const unsigned int entryIndex   = teamMember.team_rank();
    atomicDotProducts[dotIndex] +=  
        data_A(dotIndex, entryIndex) * data_B(dotIndex, entryIndex);
}
```

What if teamSize \neq dotProductSize?

First attempt at array of dot products:

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex      = teamMember.league_rank();
    const unsigned int entryIndex   = teamMember.team_rank();
    atomicDotProducts[dotIndex] +=
        data_A(dotIndex, entryIndex) * data_B(dotIndex, entryIndex);
}
```

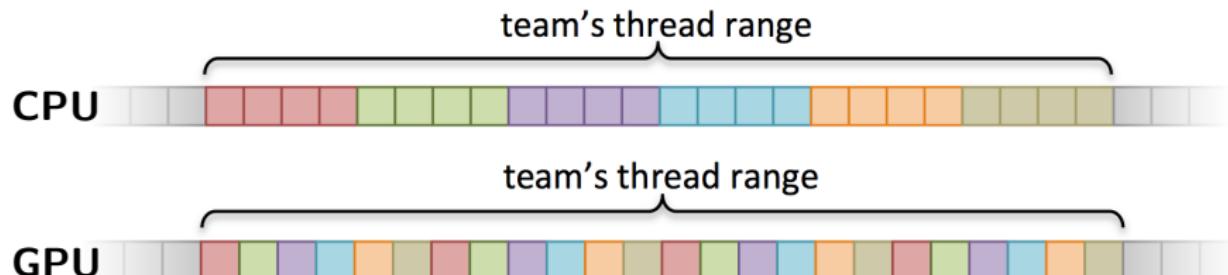
What if teamSize \neq dotProductSize?



First attempt at array of dot products:

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex      = teamMember.league_rank();
    const unsigned int entryIndex   = teamMember.team_rank();
    atomicDotProducts[dotIndex] +=
        data_A(dotIndex, entryIndex) * data_B(dotIndex, entryIndex);
}
```

What if teamSize \neq dotProductSize?



We shouldn't be hard-coding an access pattern...

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex = teamMember.league_rank();
    double dotProduct;
    “do a reduction”(“over dotProductSize indices”,
        [=] (const unsigned int i) {
            dotProduct += data_A(dotIndex,i)*data_B(dotIndex,i);
        });
    if (teamMember.team_rank() == 0) {
        _dotProducts[dotIndex] = dotProduct;
    }
}
```

We shouldn't be hard-coding an access pattern...

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex = teamMember.league_rank();
    double dotProduct;
    “do a reduction”(“over dotProductSize indices”,
        [=] (const unsigned int i) {
            dotProduct += data_A(dotIndex,i)*data_B(dotIndex,i);
        });
    if (teamMember.team_rank() == 0) {
        _dotProducts[dotIndex] = dotProduct;
    }
}
```

If this were a parallel execution,
we'd use Kokkos::parallel_reduce.

We shouldn't be hard-coding an access pattern...

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex = teamMember.league_rank();
    double dotProduct;
    “do a reduction”(“over dotProductSize indices”,
        [=] (const unsigned int i) {
            dotProduct += data_A(dotIndex,i)*data_B(dotIndex,i);
        });
    if (teamMember.team_rank() == 0) {
        _dotProducts[dotIndex] = dotProduct;
    }
}
```

If this were a parallel execution,
we'd use Kokkos::parallel_reduce.

Key idea: this *is* a parallel execution.

We shouldn't be hard-coding an access pattern...

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex = teamMember.league_rank();
    double dotProduct;
    “do a reduction”(“over dotProductSize indices”,
        [=] (const unsigned int i) {
            dotProduct += data_A(dotIndex,i)*data_B(dotIndex,i);
        });
    if (teamMember.team_rank() == 0) {
        _dotProducts[dotIndex] = dotProduct;
    }
}
```

If this were a parallel execution,
we'd use Kokkos::parallel_reduce.

Key idea: this *is* a parallel execution.

⇒ **Nested parallel patterns**

TeamThreadRange:

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex = teamMember.league_rank();
    double dotProduct;
    parallel_reduce(TeamThreadRange(teamMember, dotProductSize),
        [=] (const unsigned int i, double & valueToUpdate) {
            valueToUpdate += data_A(dotIndex,i)*data_B(dotIndex,i)
        }, dotProduct);
    if (teamMember.team_rank() == 0) {
        _dotProducts[dotIndex] = dotProduct;
    }
}
```

TeamThreadRange:

```
operator() (const member_type & teamMember) {
    const unsigned int dotIndex = teamMember.league_rank();
    double dotProduct;
    parallel_reduce(TeamThreadRange(teamMember, dotProductSize),
        [=] (const unsigned int i, double & valueToUpdate) {
            valueToUpdate += data_A(dotIndex,i)*data_B(dotIndex,i)
        }, dotProduct);
    if (teamMember.team_rank() == 0) {
        _dotProducts[dotIndex] = dotProduct;
    }
}
```

- ▶ The mapping of work indices to threads is architecture-dependent.
- ▶ The amount of work given to the TeamThreadRange need not be a multiple of the team_size.

Anatomy of nested parallelism:

```
parallel_outer(
    TeamPolicy<ExecutionSpace>(numberOfTeams, teamSize),
    KOKKOS_LAMBDA (const member_type & teamMember[, ...]) {
        /* beginning of outer body */
        parallel_inner(
            TeamThreadRange(teamMember, thisTeamsRangeSize),
            [=] (const unsigned int indexWithinBatch[, ...]) {
                /* inner body */
                }[, ...]);
        /* end of outer body */
    }[, ...]);
}
```

- ▶ `parallel_outer` and `parallel_inner` may be any combination of `for`, `reduce`, or `scan`.
- ▶ The inner lambda may capture by reference, but capture-by-value is recommended.
- ▶ The policy of the inner lambda is always a `TeamThreadRange`.
- ▶ `TeamThreadRange` cannot be nested.

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

```
ParallelFunctor functor(...);  
const unsigned int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_recommended(functor);  
const unsigned int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_max(functor);
```

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

```
ParallelFunctor functor(...);  
const unsigned int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_recommended(functor);  
const unsigned int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_max(functor);
```

NVIDIA GPU:

- ▶ Special hardware available for coordination within a team.
- ▶ Within a team 32 threads (*warp*) execute “lock step.”
- ▶ Maximum team size: **1024**; Recommended team size: **256**

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

```
ParallelFunctor functor(...);  
const unsigned int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_recommended(functor);  
const unsigned int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_max(functor);
```

NVIDIA GPU:

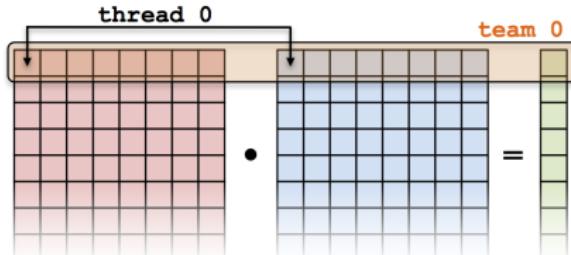
- ▶ Special hardware available for coordination within a team.
- ▶ Within a team 32 threads (*warp*) execute “lock step.”
- ▶ Maximum team size: **1024**; Recommended team size: **256**

Intel Xeon Phi:

- ▶ Recommended team size: # hyperthreads per core
- ▶ Hyperthreads share entire cache hierarchy
 - a well-coordinated team avoids cache-thrashing

Task: Modify the array of dot products implementation to use teams.

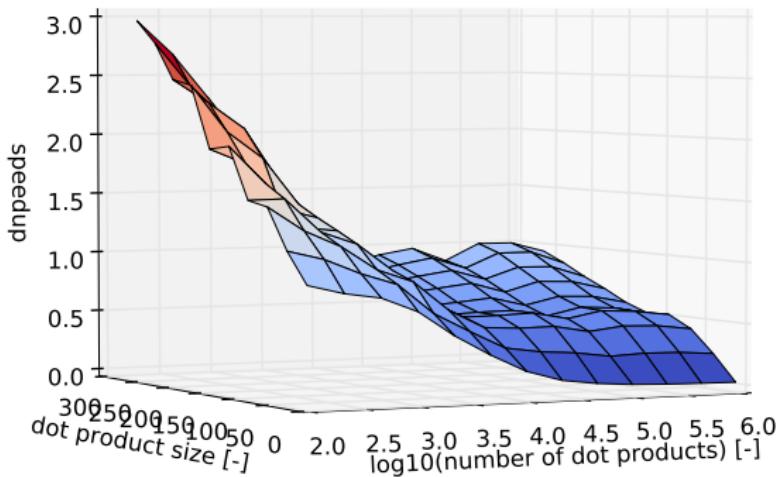
TODO: Christian, are they modifying or making a new one?



Details:

- ▶ What layout should the data have?
- ▶ Instructions in Exercises/TODO: Christian/README.
- ▶ **New material:** TeamPolicy, member_type, TeamThreadRange.

Team performance: team speedup over flat



Advanced examples leveraging hierarchical parallelism and team-shared memory.

Advanced examples leveraging hierarchical parallelism and team-shared memory.

- ▶ Sparse matrix-vector product
- ▶ Array fill via “push back” - resolving performance problem
- ▶ Tensor contraction with team-shared memory
- ▶ Finite difference stencil

Sparse matrix **representation**:

colIndices	17	45	73	7	...
colValues	3.1	4.1	5.9	2.6	...
irow	0	3	9	...	

Serial algorithm:

```
for (row = 0; row < matrixSize; ++row) {  
    double total = 0;  
    for (int i = irow[row]; i < irow[row+1]; ++i) {  
        total += colValues[i] * vec[colIndices[i]];  
    }  
    product[row] = total;  
}
```

Sparse matrix **representation**:

colIndices	17	45	73	7	...
colValues	3.1	4.1	5.9	2.6	...
irow	0	3	9	...	

Serial algorithm:

```
for (row = 0; row < matrixSize; ++row) {  
    double total = 0;  
    for (int i = irow[row]; i < irow[row+1]; ++i) {  
        total += colValues[i] * vec[colIndices[i]];  
    }  
    product[row] = total;  
}
```

How many ways can this be parallelized?

TODO: have a picture here

Flat over rows:

```
operator()(int row) const {
    double total = 0;
    for (int i = _irow(row); i < _irow(row+1); ++i) {
        total += _colValues(i) * _vec(_colIndices(i));
    }
    _product[row] = total;
}
```

- ▶ One thread **per row**
- ▶ `_colValues` and `_colIndices`: **cached/uncoalesced**
- ▶ `_vec`: **uncached/uncoalesced**
- ▶ `_product`: **cached/coalesced**

Flat over non-zeros with atomics:

```
operator()(int i) const {
    atomic_add(&_product(_rowIndices(i)),
               colValues(i) * _vec(_colIndices(i)));
}
```

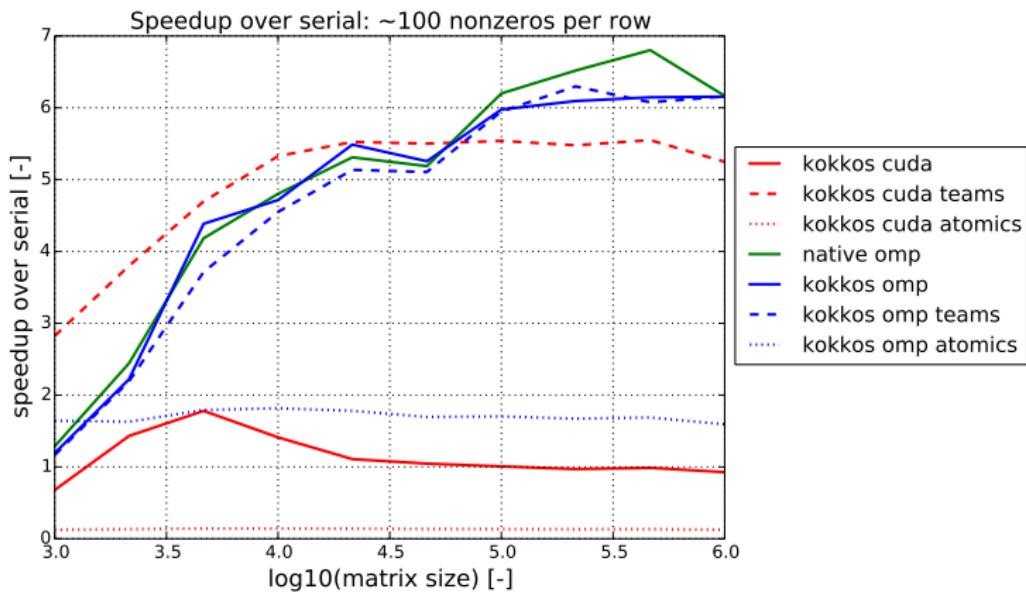
- ▶ One thread **per nonzero entry**
- ▶ `_colValues`, `_colIndices`, `_rowIndices`: **cached/coalesced**
- ▶ `_vec`: **uncached/uncoalesced**
- ▶ `_product`: **good/bad**

Hierarchical **thread teams**:

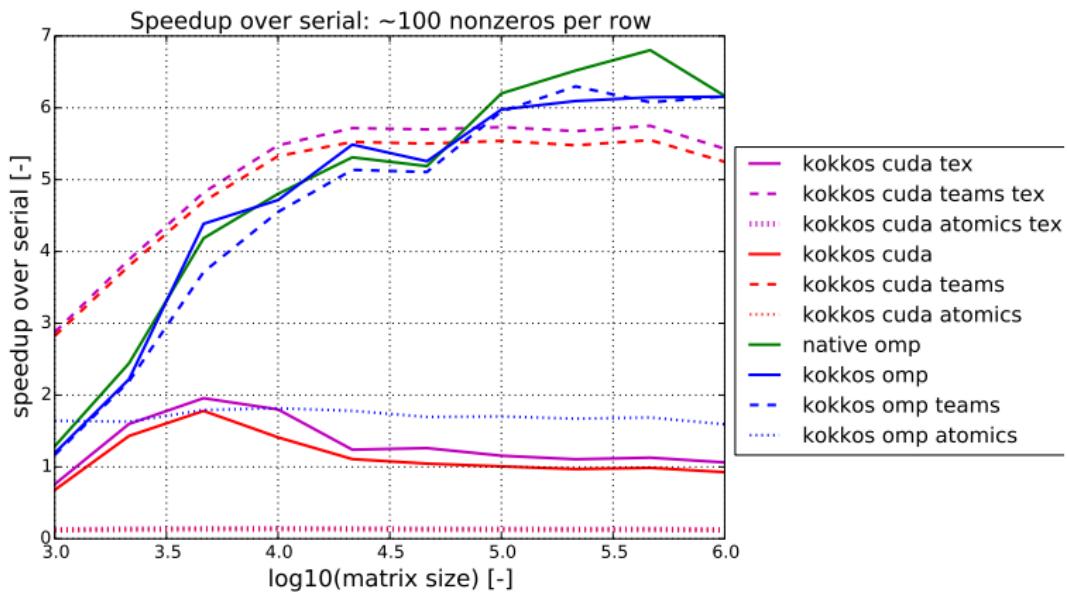
```
operator()(member_type teamMember) const {
    int row = teamMember.league_rank();
    double total;
    Kokkos::parallel_reduce
        (Kokkos::TeamThreadRange(teamMember, numNonzeros),
         [=] (const unsigned int j, double & valueToUpdate) {
            const unsigned int i = irow(row) + j;
            valueToUpdate += _colValues(i) * _vec(_colIndices(i));
        }, total);
    if (teamMember.team_rank() == 0) {
        _product(row) = total;
    }
}
```

- ▶ One thread **per nonzero entry**
- ▶ `_colValues`, `_colIndices`, `_rowIndices`: **cached/coalesced**
- ▶ `_vec`: **uncached/uncoalesced**
- ▶ `_product`: **cached/single write**

Performance:

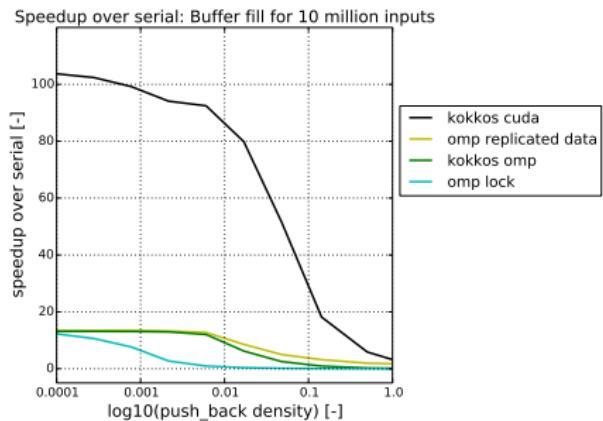


Performance (including texture versions):



Recall: Filling an array by “pushing back” entries with an atomic:

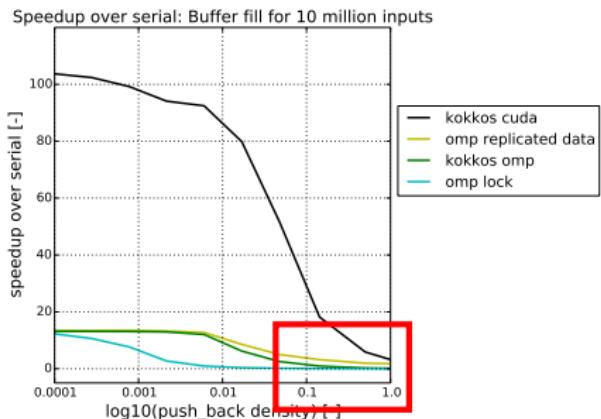
```
operator()(const unsigned int atomIndex) const {
    if (atomIndex % atomicStride == 0) {
        const unsigned int claimedIndex =
            Kokkos::atomic_fetch_add(&globalCount, 1);
        positionsToSend[claimedIndex] = atomPosition;
    }
}
```



Recall: Filling an array by “pushing back” entries with an atomic:

```
operator()(const unsigned int atomIndex) const {
    if (atomIndex % atomicStride == 0) {
        const unsigned int claimedIndex =
            Kokkos::atomic_fetch_add(&globalCount, 1);
        positionsToSend[claimedIndex] = atomPosition;
    }
}
```

Performance limitation:
high contention for
globalCount.

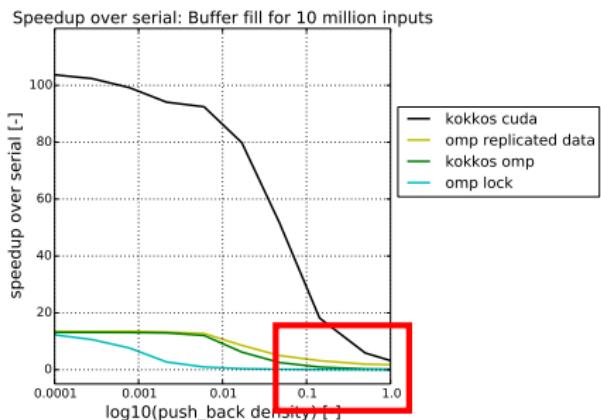


Recall: Filling an array by “pushing back” entries with an atomic:

```
operator()(const unsigned int atomIndex) const {
    if (atomIndex % atomicStride == 0) {
        const unsigned int claimedIndex =
            Kokkos::atomic_fetch_add(&globalCount, 1);
        positionsToSend[claimedIndex] = atomPosition;
    }
}
```

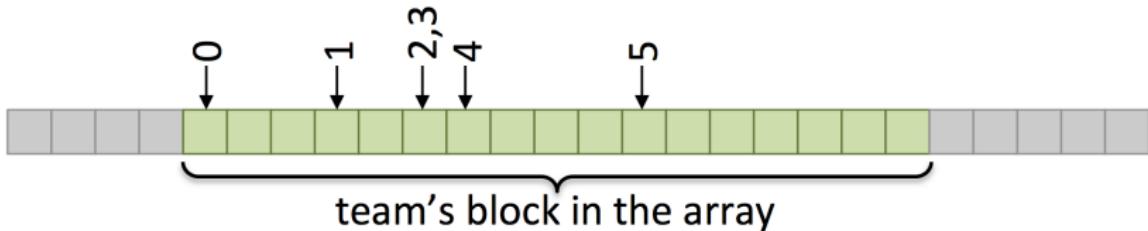
Performance limitation:
high contention for
`globalCount`.

⇒ Reduce contention with
Thread teams



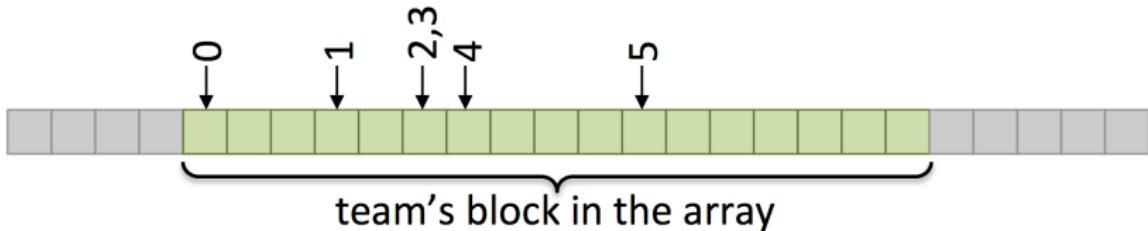
(conceptual) Steps:

1. Each team member determines how many entries to add
2. The team counts the total entries to add
3. One team member claims room for this team's block
4. The block location is shared
5. Each team member determines its offset into the block
6. Each team member inserts its entries into the block

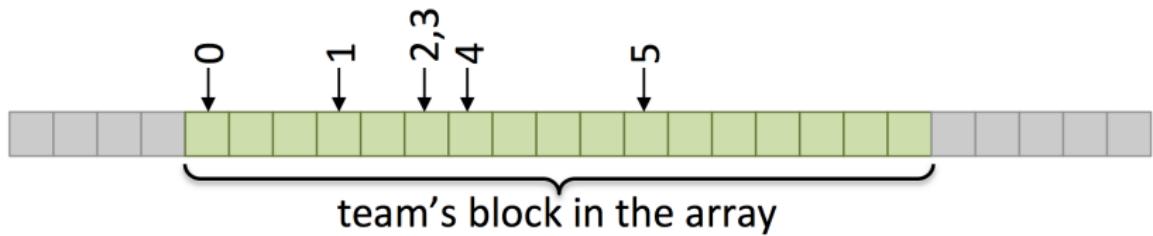


(conceptual) Steps:

1. Each team member determines how many entries to add
2. The team counts the total entries to add
3. One team member claims room for this team's block
4. The block location is shared
5. Each team member determines its offset into the block
6. Each team member inserts its entries into the block

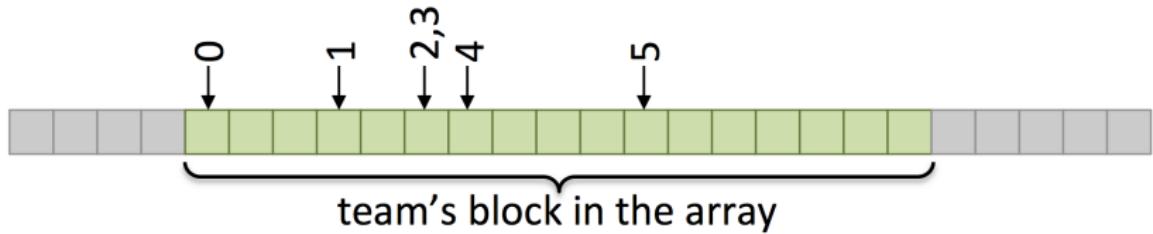


Let's focus on: how will threads know their offsets into the block?



Desired functionality:

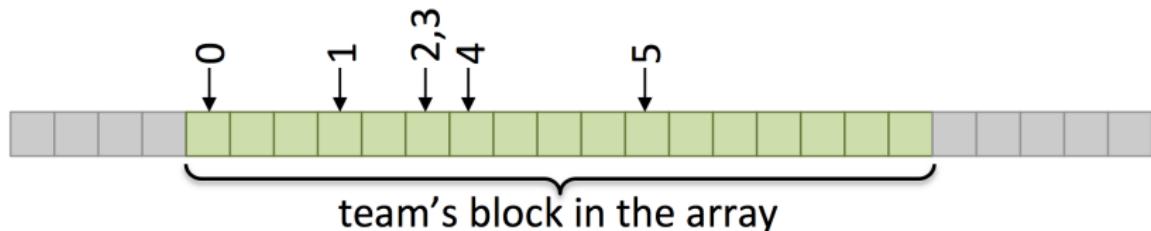
team_rank	input	offset
0	3	0
1	2	3
2	0	5
3	1	5
4	4	6
5	7	10



Desired functionality:

team_rank	input	offset
0	3	0
1	2	3
2	0	5
3	1	5
4	4	6
5	7	10

So, this is like `parallel_scan`, but *within* a team.

Desired functionality:

team_rank	input	offset
0	3	0
1	2	3
2	0	5
3	1	5
4	4	6
5	7	10

So, this is like `parallel_scan`, but *within* a team.

```
thisTeamMembersLocalOffset = teamMember.team_scan(input);
```

(conceptual) Steps:

1. Each team member determines how many entries to add
2. The team counts the total entries to add
3. One team member claims room for this team's block
4. The block location is shared
5. Each team member determines its offset into the block
6. Each team member inserts its entries into the block

array-fill pattern

(conceptual) Steps:

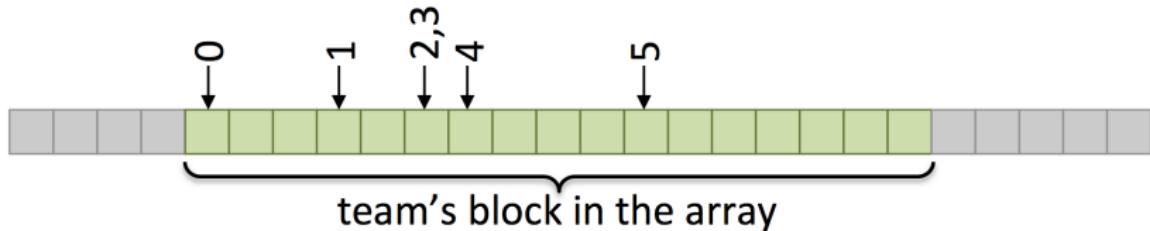
1. Each team member determines how many entries to add
2. The team counts the total entries to add
3. One team member claims room for this team's block
4. The block location is shared
5. Each team member determines its offset into the block
6. Each team member inserts its entries into the block

While these steps *are* possible, this **array-fill** pattern is common enough that an optimized variant of `team_scan` is provided:

```
thisTeamMembersGlobalOffset =  
    teamMember.team_scan(input, &globalCount());
```

Array-fill pseudocode:

```
indices = calculate indices this team member needs to add  
  
thisTeamMembersGlobalOffset =  
    teamMember.team_scan(indices.size(), &globalCount());  
  
for (i = 0; i < indices.size(); ++i) {  
    _buffer(thisTeamMembersGlobalOffset + i) = indices[i];  
}
```



Team size:

```
int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_recommended(functor);
```

League size:

Except in very rare cases, the league size is the smallest number that provides one team member per piece of work:

```
int numberOfTeams = ceil(amountOfWork / float(teamSize));  
                    = (amountOfWork - 1) / teamSize + 1;
```

Team size:

```
int teamSize =  
    TeamPolicy<ExecutionSpace>::team_size_recommended(functor);
```

League size:

Except in very rare cases, the league size is the smallest number that provides one team member per piece of work:

```
int numberOfTeams = ceil(amountOfWork / float(teamSize));  
                    = (amountOfWork - 1) / teamSize + 1;
```

Warning

`league_size() * team_size() ≥ amountOfWork.`

```
operator()(const member_type & teamMember) {  
    int index =  
        teamMember.league_rank() * teamMember.team_size() +  
        teamMember.team_rank();  
    double d = _data(index);
```

Exercise: Implement a team_scan version of the ArrayFill exercise.

Details:

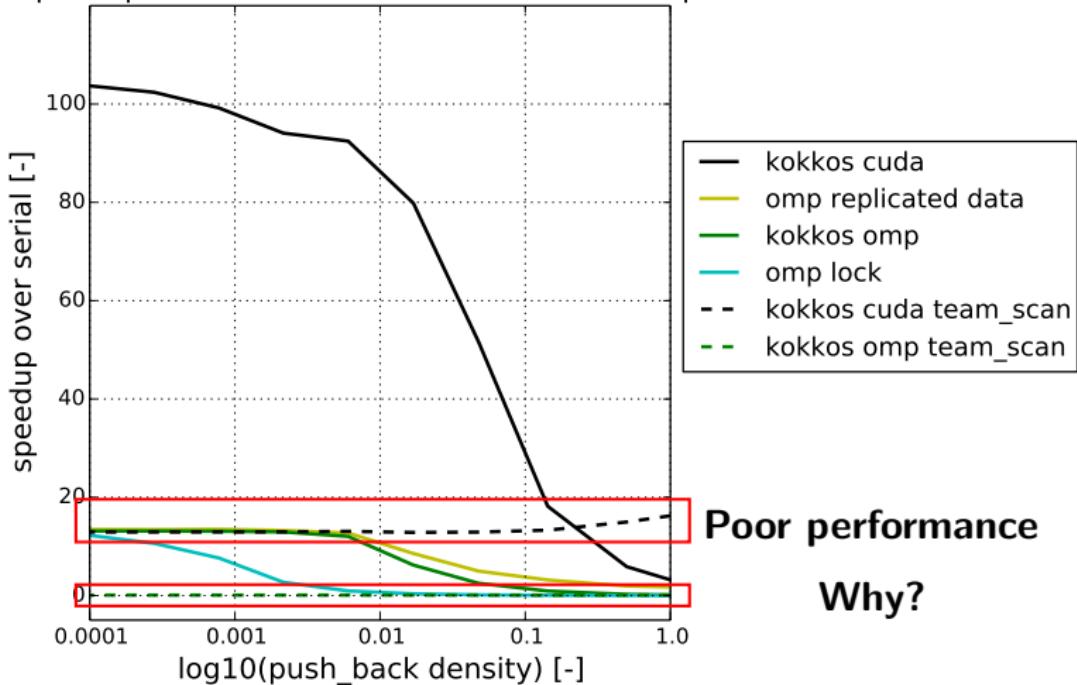
- ▶ Body logic is:

```
operator()(const member_type & teamMember) {  
    determine this team members index  
    determine if this team members index should be added  
    use team_scan to determine this team members offset  
    if (this team members index should be added) {  
        insert index into array  
    }  
}
```

- ▶ Instructions in Exercises/ArrayFill/README.
- ▶ **New material:** team_scan.

team_scan performance:

Speedup over serial: Buffer fill for 10 million inputs



Poor performance

Why?

High level view of kernel:

```
operator()(const member_type & teamMember) {  
    check one number  
    do a team_scan  
    if (...) {  
        insert index into array  
    }  
}
```

Performance limitation: too little work per team_scan

High level view of kernel:

```
operator()(const member_type & teamMember) {  
    check one number  
    do a team_scan  
    if (...) {  
        insert index into array  
    }  
}
```

Performance limitation: too little work per team_scan

Solution: local buffers

```
operator()(const member_type & teamMember) {  
    unsigned int localBuffer[LocalBufferSize];  
    check LocalBufferSize numbers  
    do a team_scan  
    if (...) {  
        add indices to buffer  
    }  
}
```

High level view of kernel:

```
operator()(const member_type & teamMember) {  
    check one number  
    do a team_scan  
    if (...) {  
        insert index into array  
    }  
}
```

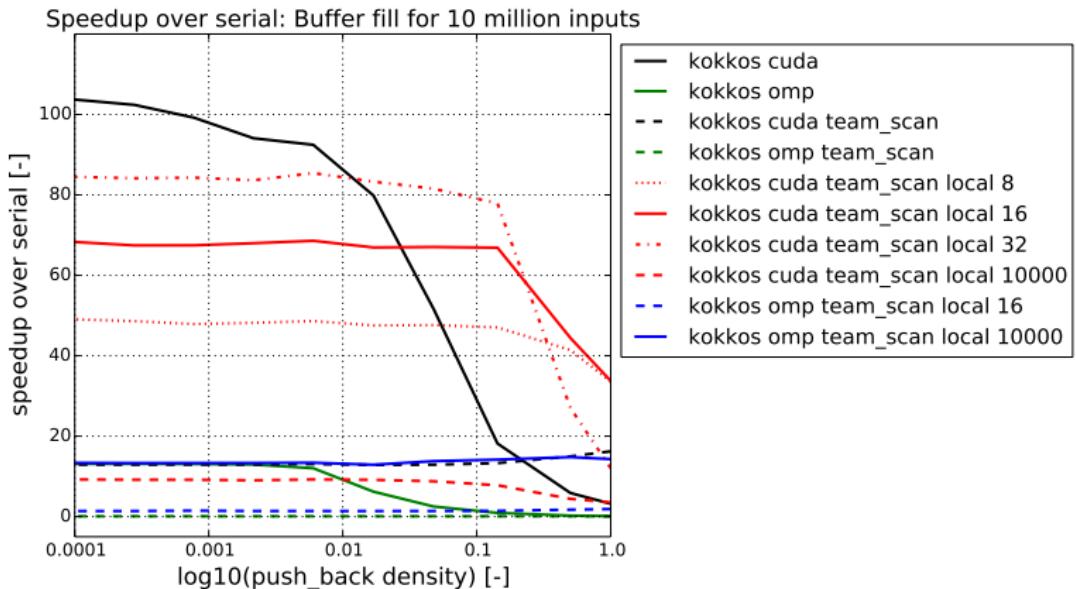
Performance limitation: too little work per team_scan

Solution: local buffers

```
operator()(const member_type & teamMember) {  
    unsigned int localBuffer[LocalBufferSize];  
    check LocalBufferSize numbers  
    do a team_scan  
    if (...) {  
        add indices to buffer  
    }  
}
```

How do you “check LocalBufferSize numbers?”

team_scan with local buffers **performance:**

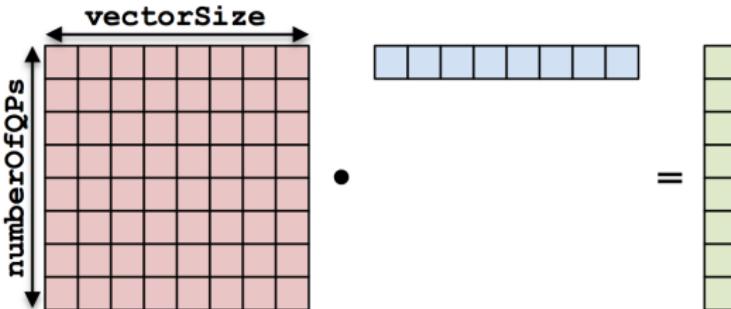


Shared memory

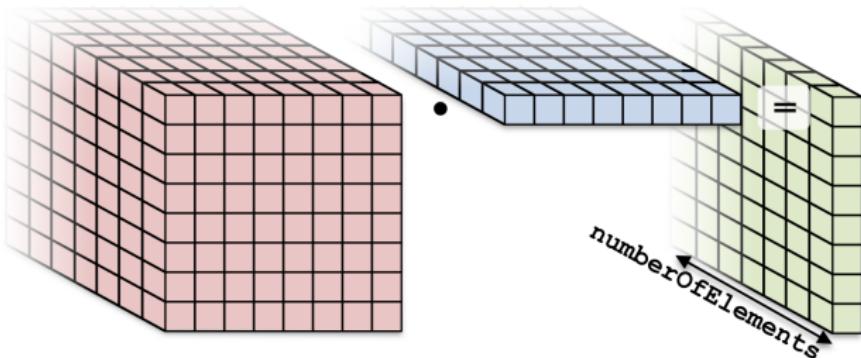
Learning objectives:

- ▶ Understand how shared memory can reduce global memory accesses
- ▶ Recognize when to use shared memory
- ▶ Understand how to use shared memory and why barriers are necessary

One slice of contractDataFieldScalar:



```
for (qp = 0; qp < numberOfQPs; ++qp) {  
    total = 0;  
    for (i = 0; i < vectorSize; ++i) {  
        total += A(qp, i) * B(i);  
    }  
    result(qp) = total;  
}
```

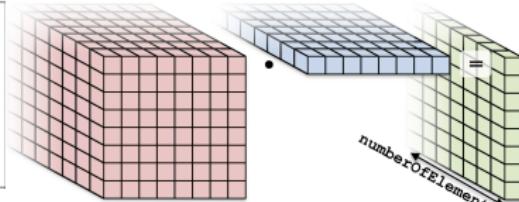
contractDataFieldScalar:

```

for (element = 0; element < number_of_elements; ++element) {
    for (qp = 0; qp < number_of_QPs; ++qp) {
        total = 0;
        for (i = 0; i < vector_size; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```

```
for (element = 0; element < numberElements; ++element) {  
    for (qp = 0; qp < numberQPs; ++qp) {  
        total = 0;  
        for (i = 0; i < vectorSize; ++i) {  
            total += A(element, qp, i) * B(element, i);  
        }  
        result(element, qp) = total;  
    }  
}
```



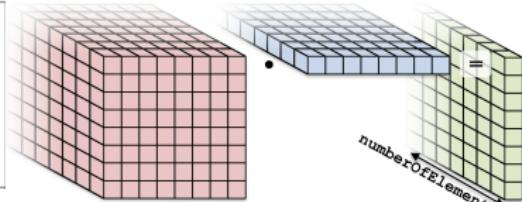
Parallelization approaches:

- ▶ Each thread handles an element.
Threads: numberElements

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



Parallelization approaches:

- ▶ Each thread handles an element.

Threads: numberElements

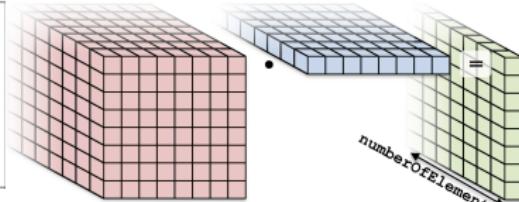
- ▶ Each thread handles a qp.

Threads: $\text{numberElements} * \text{numberQPs}$

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



Parallelization approaches:

- ▶ Each thread handles an element.

Threads: numberElements

- ▶ Each thread handles a qp.

Threads: $\text{numberElements} * \text{numberQPs}$

- ▶ Each thread handles an i.

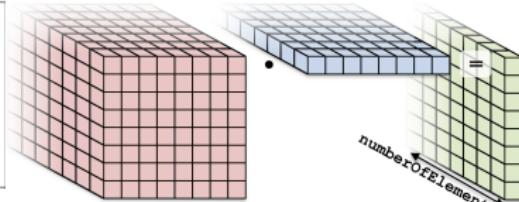
Threads: $\text{numElements} * \text{numQPs} * \text{vectorSize}$

Requires a parallel_reduce.

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



Parallelization approaches:

- ▶ Each thread handles an element.

Threads: numberElements

- ▶ Each thread handles a qp.

Threads: numberElements * numQPs

- ▶ Each thread handles an i.

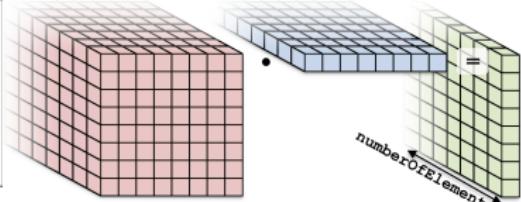
Threads: numElements * numQPs * vectorSize

Requires a parallel_reduce.

```

for (element = 0; element < numberElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



Flat kernel: Each thread handles a quadrature point

```

operator()(int index) {
    int element = extractElementFromIndex(index);
    int qp = extractQPFromIndex(index);
    double total = 0;
    for (int i = 0; i < _vectorSize; ++i) {
        total += _A(element, qp, i) * _B(element, i);
    }
    _result(element, qp) = total;
}

```

TODO:

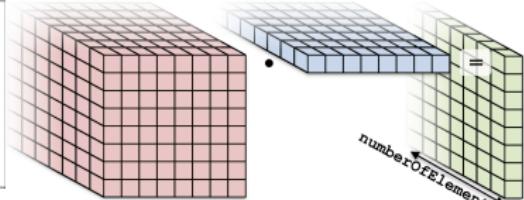
how many ties is B read?

big idea: reduce the reads to global memory by using team scratch space

```

for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



Teams kernel: Each team handles an element

```

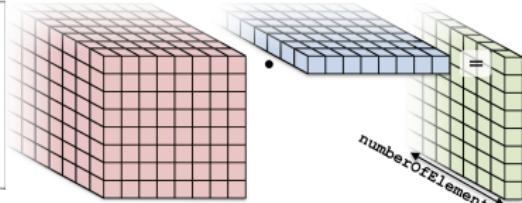
operator()(member_type teamMember) {
    int element = teamMember.league_rank();
    parallel_for(
        TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * _B(element, i);
            }
            _result(element, qp) = total;
        });
}

```

```

for (element = 0; element < numberOfElements; ++element) {
    for (qp = 0; qp < numberQPs; ++qp) {
        total = 0;
        for (i = 0; i < vectorSize; ++i) {
            total += A(element, qp, i) * B(element, i);
        }
        result(element, qp) = total;
    }
}

```



Teams kernel: Each team handles an element

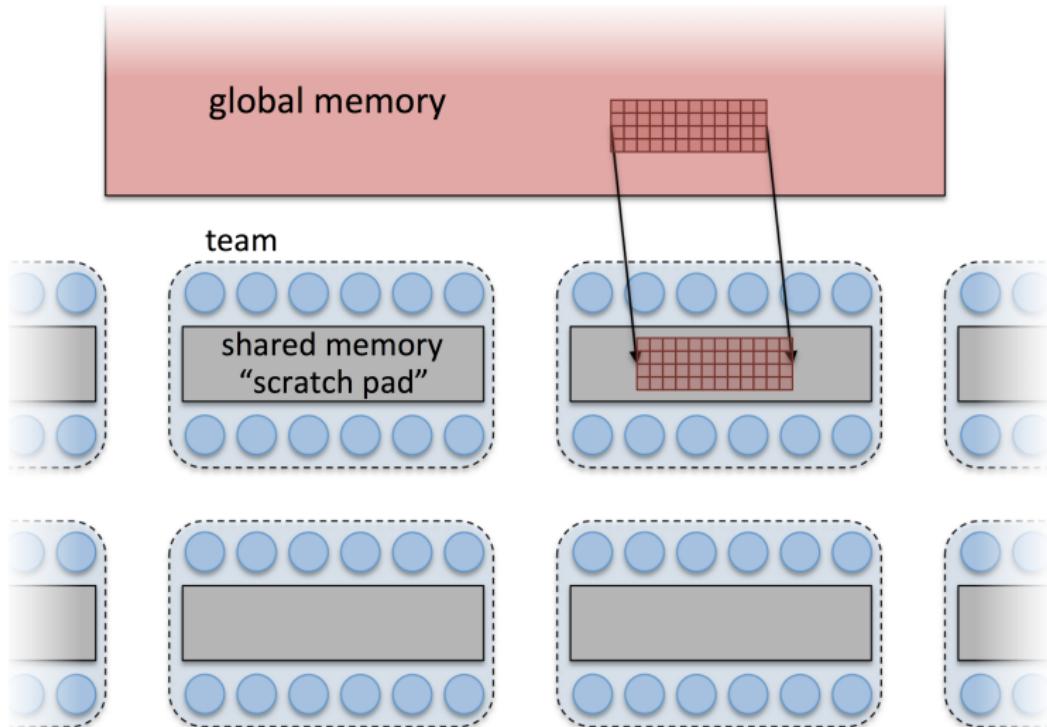
```

operator()(member_type teamMember) {
    int element = teamMember.league_rank();
    parallel_for(
        TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * _B(element, i);
            }
            _result(element, qp) = total;
        });
}

```

No real advantage (yet)

Each team has access to a “scratch pad”.



Shared memory (scratch pad) **details**:

- ▶ Accessing data in shared memory is (usually) **much faster** than global memory.
- ▶ **GPUs** have separate, dedicated, small, low-latency shared memories (*NOT subject to coalescing requirements*).
- ▶ **CPUs** don't have special hardware, but programming with shared memory results in cache-aware memory access patterns.
- ▶ Roughly, it's like a *user-managed L1 cache*.

Shared memory (scratch pad) **details**:

- ▶ Accessing data in shared memory is (usually) **much faster** than global memory.
- ▶ **GPUs** have separate, dedicated, small, low-latency shared memories (*NOT subject to coalescing requirements*).
- ▶ **CPUs** don't have special hardware, but programming with shared memory results in cache-aware memory access patterns.
- ▶ Roughly, it's like a *user-managed L1 cache*.

Important concept

When members of a team read the same data multiple times, it's better to load the data into shared memory and read from there.

To use shared memory, you need to:

1. **Tell Kokkos how much** total shared memory you'll need.
2. **Make** shared memory **views** inside your kernels.

To use shared memory, you need to:

1. **Tell Kokkos how much** total shared memory you'll need.
2. **Make** shared memory **views** inside your kernels.

```
struct ParallelFunctor {
    // shared memory view type
    typedef View< double*
        , execution_space::scratch_memory_space
        , MemoryUnmanaged > SharedViewType ;

    // tell kokkos how much shared memory (in bytes) are needed
    size_t team_shmem_size(int teamSize) const {
        return SharedViewType::shmem_size( _vectorSize );
    }

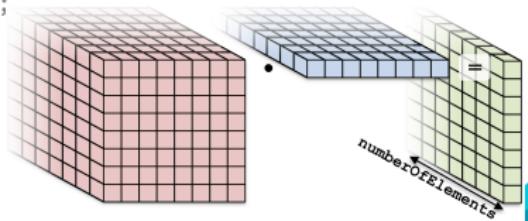
    void operator()(member_type teamMember) const {
        // this creates a view from the pre-existing shared memory
        SharedViewType shared(teamMember.team_shmem(), _vectorSize);
        shared(10) = ...;
    }
};
```

Kernel outline for teams with shared memory:

```
operator()(member_type teamMember) {
    SharedViewType shared(teamMember.team_shmem(), _vectorSize);

    // load slice of _B into shared

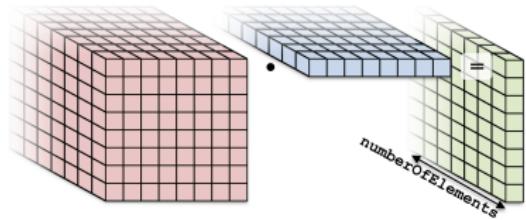
    parallel_for(
        TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * shared(i);
            }
            _result(element, qp) = total;
        });
}
```



How to populate the shared memory?

- ▶ One thread loads it all?

```
if (teamMember.team_rank() == 0) {  
    for (int i = 0; i < _vectorSize; ++i) {  
        shared(i) = B(element, i);  
    }  
}
```



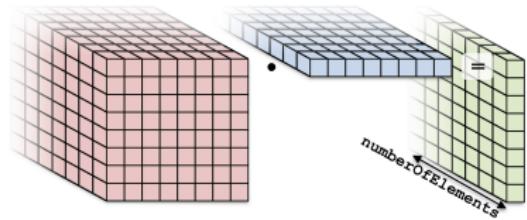
How to populate the shared memory?

- ▶ ~~One thread loads it all?~~ **Serial**

```
if (teamMember.team_rank() == 0) {  
    for (int i = 0; i < _vectorSize; ++i) {  
        shared(i) = B(element, i);  
    }  
}
```

- ▶ Each thread loads one entry?

```
shared(team_rank) = B(element, team_rank);
```



How to populate the shared memory?

- ▶ ~~One thread loads it all?~~ **Serial**

```
if (teamMember.team_rank() == 0) {
    for (int i = 0; i < _vectorSize; ++i) {
        shared(i) = B(element, i);
    }
}
```

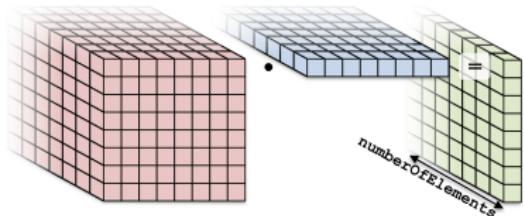
- ▶ ~~Each thread loads one entry?~~ **$\text{teamSize} \neq \text{_vectorSize}$**

```
shared(team_rank) = B(element, team_rank);
```

- ▶ **TeamThreadRange**

```
parallel_for(
    TeamThreadRange(teamMember, _vectorSize),
    [=] (int i) {
        shared(i) = B(element, i);
    });

```



How to populate the shared memory?

- ▶ ~~One thread loads it all?~~ **Serial**

```
if (teamMember.team_rank() == 0) {
    for (int i = 0; i < _vectorSize; ++i) {
        shared(i) = B(element, i);
    }
}
```

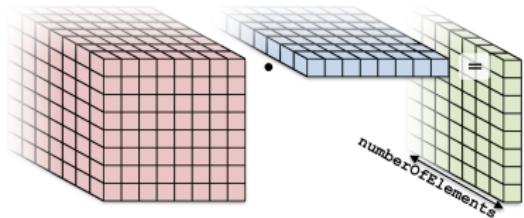
- ▶ ~~Each thread loads one entry?~~ **$\text{teamSize} \neq \text{_vectorSize}$**

```
shared(team_rank) = B(element, team_rank);
```

- ▶ **TeamThreadRange**

```
parallel_for(
    TeamThreadRange(teamMember, _vectorSize),
    [=] (int i) {
        shared(i) = B(element, i);
    });

```



(incomplete) Kernel for teams with shared memory:

```
operator()(member_type teamMember) {
    SharedViewType shared(...);

    parallel_for(TeamThreadRange(teamMember, _vectorSize),
        [=] (int i) {
            shared(i) = _B(element, i);
        });
}

parallel_for(TeamThreadRange(teamMember, _numberOfQPs),
    [=] (int qp) {
        double total = 0;
        for (int i = 0; i < _vectorSize; ++i) {
            total += _A(element, qp, i) * shared(i);
        }
        _result(element, qp) = total;
    });
}
```

(incomplete) Kernel for teams with shared memory:

```
operator()(member_type teamMember) {
    SharedViewType shared(...);

    parallel_for(TeamThreadRange(teamMember, _vectorSize),
        [=] (int i) {
            shared(i) = _B(element, i);
        });
}

parallel_for(TeamThreadRange(teamMember, _numberOfQPs),
    [=] (int qp) {
        double total = 0;
        for (int i = 0; i < _vectorSize; ++i) {
            total += _A(element, qp, i) * shared(i);
        }
        _result(element, qp) = total;
    });
}
```

Problem: threads may start to use **shared** before all threads are done loading.

Kernel for teams with shared memory:

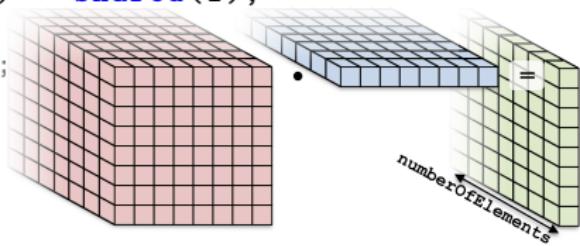
```

operator()(member_type teamMember) {
    SharedViewType shared(...);

    parallel_for(TeamThreadRange(teamMember, _vectorSize),
        [=] (int i) {
            shared(i) = _B(element, i);
        });
    teamMember.team_barrier();

    parallel_for(TeamThreadRange(teamMember, _numberOfQPs),
        [=] (int qp) {
            double total = 0;
            for (int i = 0; i < _vectorSize; ++i) {
                total += _A(element, qp, i) * shared(i);
            }
            _result(element, qp) = total;
        });
}

```

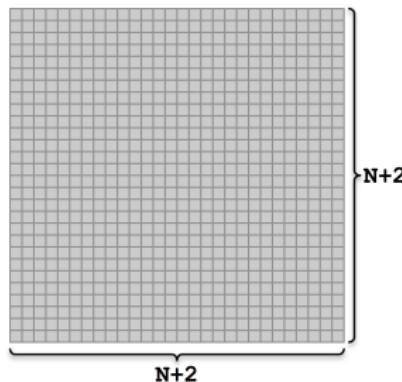
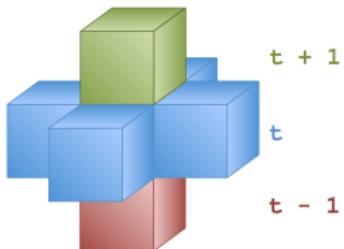


2D finite difference wave equation:

$$\begin{aligned} u_{i,j}^{t+1} &= 2u_{i,j}^t - u_{i,j}^{t-1} \\ &+ \zeta^2 (u_{i+1,j}^t - 2u_{i,j}^t + u_{i-1,j}^t) \\ &+ \zeta^2 (u_{i,j+1}^t - 2u_{i,j}^t + u_{i,j-1}^t) \end{aligned}$$

where

$$\begin{aligned} (\Delta x &= \Delta y), \\ \zeta &= \frac{c\Delta t}{\Delta j}, \\ \text{grid is } &(N+2) \times (N+2). \end{aligned}$$



Update equation:

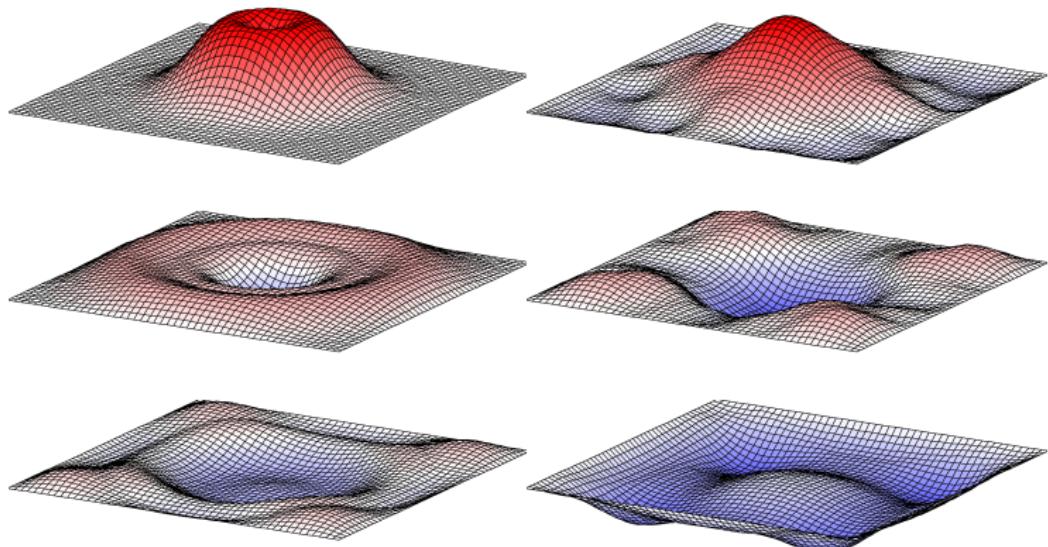
$$\begin{aligned}
 u_{i,j}^{t+1} = & 2 u_{i,j}^t - u_{i,j}^{t-1} \\
 & + \zeta^2 (u_{i+1,j}^t - 2 u_{i,j}^t + u_{i-1,j}^t) \\
 & + \zeta^2 (u_{i,j+1}^t - 2 u_{i,j}^t + u_{i,j-1}^t)
 \end{aligned}$$

Flat kernel:

```

operator()(int index) {
    int i      = index / N + 1;
    int j      = index % N + 1;
    int t      = _timestepIndex % 2;
    int tp1   = (_timestepIndex + 1) % 2;
    _u(tp1,i,j) = 2 * _u(t,i,j) - _u(tp1,i,j) +
    _courant2 * (_u(t,i+1,j) - 2 * _u(t,i,j) + _u(t,i-1,j)) +
    _courant2 * (_u(t,i,j+1) - 2 * _u(t,i,j) + _u(t,i,j-1));
}

```

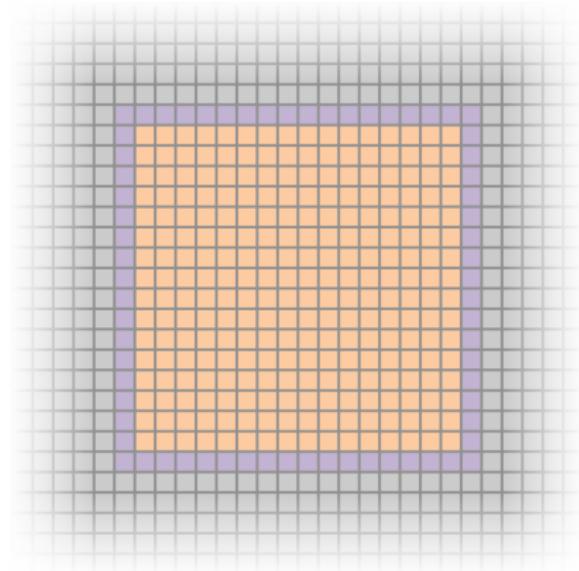


[full video](#)

Note: computational grid is much finer than shown

Exercise: Implement 2D wave equation using shared memory

```
operator()(member_type teamMember) {  
    SharedViewType shared(...);  
  
    load tile with halo into shared  
    teamMember.team_barrier();  
  
    int i = ...;  
    int j = ...;  
    update _u(tp1,i,j) using shared  
}
```



- ▶ There is a **third level** in the hierarchy below TeamThreadRange: TeamVectorRange
 - ▶ Just like for TeamThreadRange, you can perform parallel_for, parallel_reduce, or parallel_scan.
 - ▶ Important for Xeon Phi.
- ▶ Restricting execution to a **single member**:
 - PerTeam: one thread per team
 - PerThread: one vector lane per thread
- ▶ **Multiple shared views** can be made in shared memory.

- ▶ **Hierarchical work** can be parallelized via hierarchical parallelism.
- ▶ Team “worksets” are processed in nested `parallel_for` (or `reduce` or `scan`) calls with a `TeamThreadRange` policy.
- ▶ Teams have access to “scratch pad” **shared memory**.
- ▶ Many algorithms which **fill arrays** can be thread-scalably parallelized with `team_scan`.
(e.g., filling communication buffers,
graph algorithm frontiers, etc.)

DOE's Trinity (ATS-1) value added:

- ▶ General reduction, scan, and task-DAG
- ▶ Memory space for high bandwidth memory
- ▶ Hierarchical parallelism for performant hyperthreads

DOE's Trinity (ATS-1) value added:

- ▶ General reduction, scan, and task-DAG
- ▶ Memory space for high bandwidth memory
- ▶ Hierarchical parallelism for performant hyperthreads

DOE's Sierra (ATS-2) value added:

- ▶ Performance portability!
- ▶ Thread scalable general reduction and scan, task-DAG probable
- ▶ Memory spaces for accelerator
- ▶ Atomics for thread scalability
- ▶ Hierarchical parallelism for performance