Kokkos Tutorial

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

[1]Sandia National Laboratories

[2]Harvey 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.

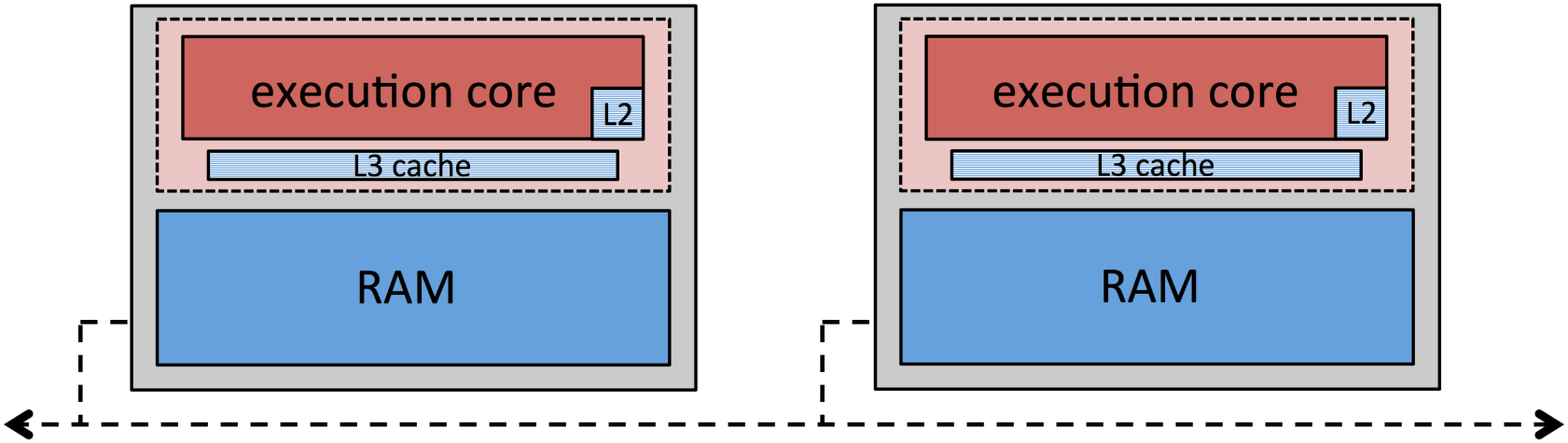
SAND2015-4178 O

Placeholder for introduction section

Recipe to follow (targeting a contemporary cluster of multi-core CPUs):

1. Make sure you're using the correct **algorithm** (e.g., factorization instead of inverse, spatial data structures to find neighbors, etc.).
2. Choose correct **data structures** to minimize time spent on cache misses.
3. Accelerate number crunching (**vectorization**, etc.).
4. Implement **shared-memory** parallelization (threading) to use all the cores of a node.
5. Implement **distributed-memory** parallelization (MPI) to use multiple nodes.

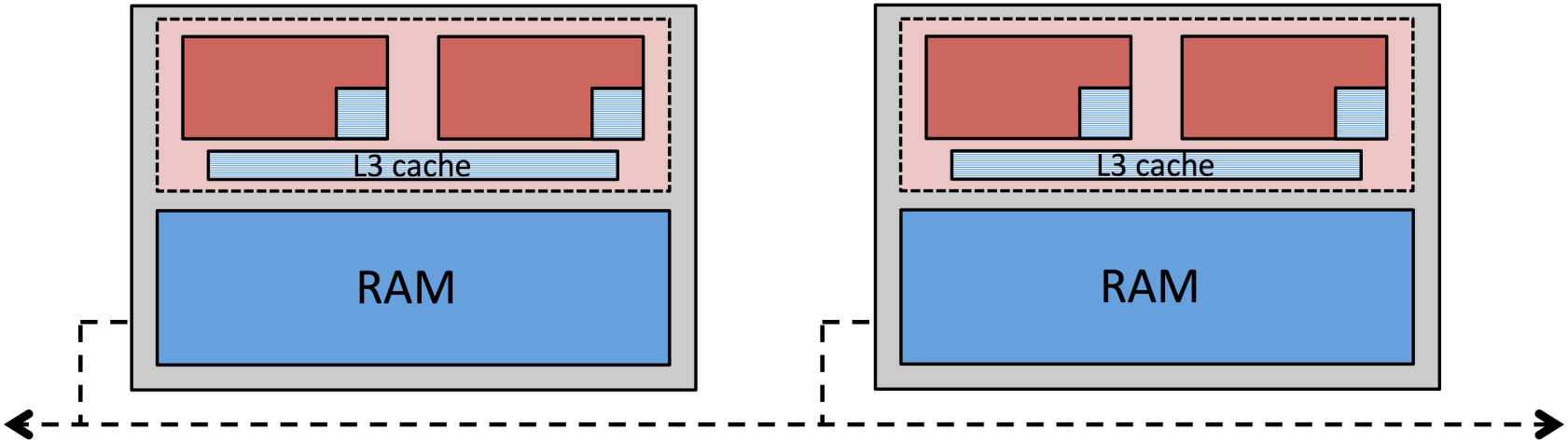
2003:



- Intel **Gallatin** architecture
  1 core, 3.0 GHz, 8 KB L1, 512 KB L2, 4 MB L3
- **MPI-only:**

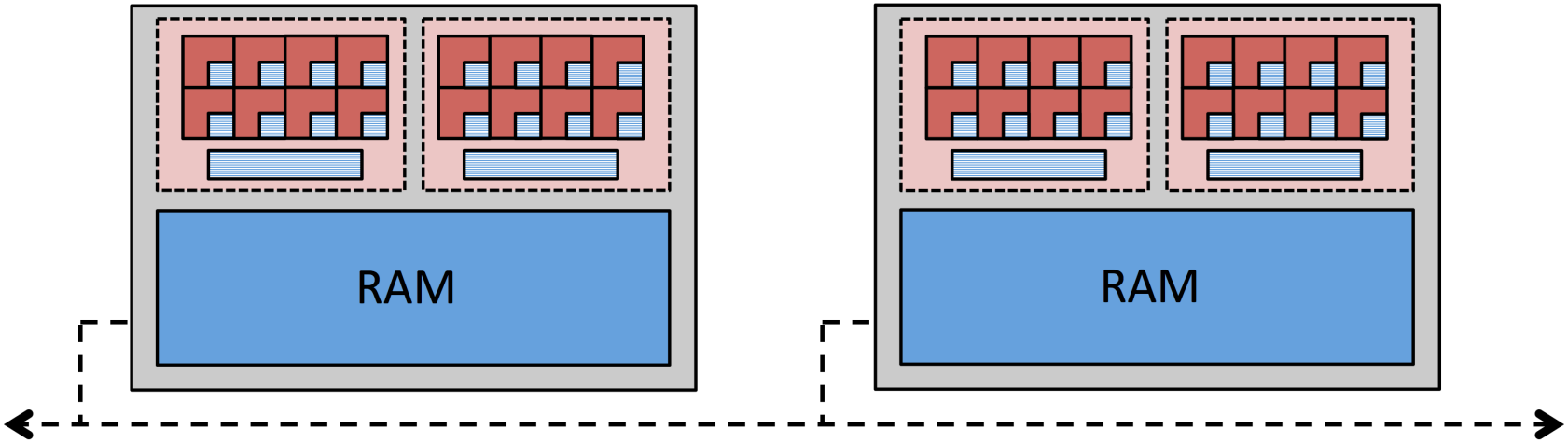
```
mpirun -np numNodes ./program
```

2006:



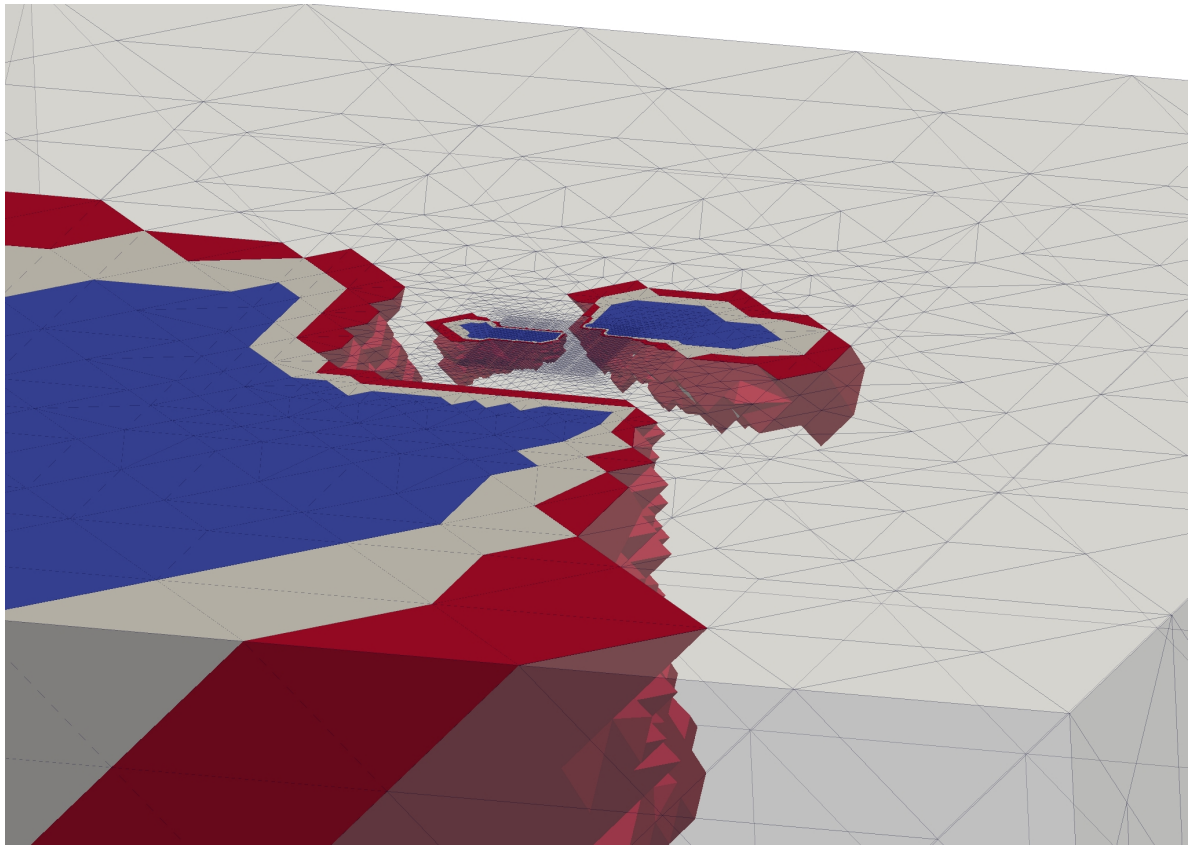
- Intel **Tulsa**
  2 cores, 3.0 GHz, 32 KB L1, 1 MB L2, 8 MB L3
- **MPI-only:**
  `mpirun -np 2*numNodes ./program`
- **MPI+threading** ("hybrid"):
  `mpirun -np  numNodes ./program -numThreads=2`

2014:



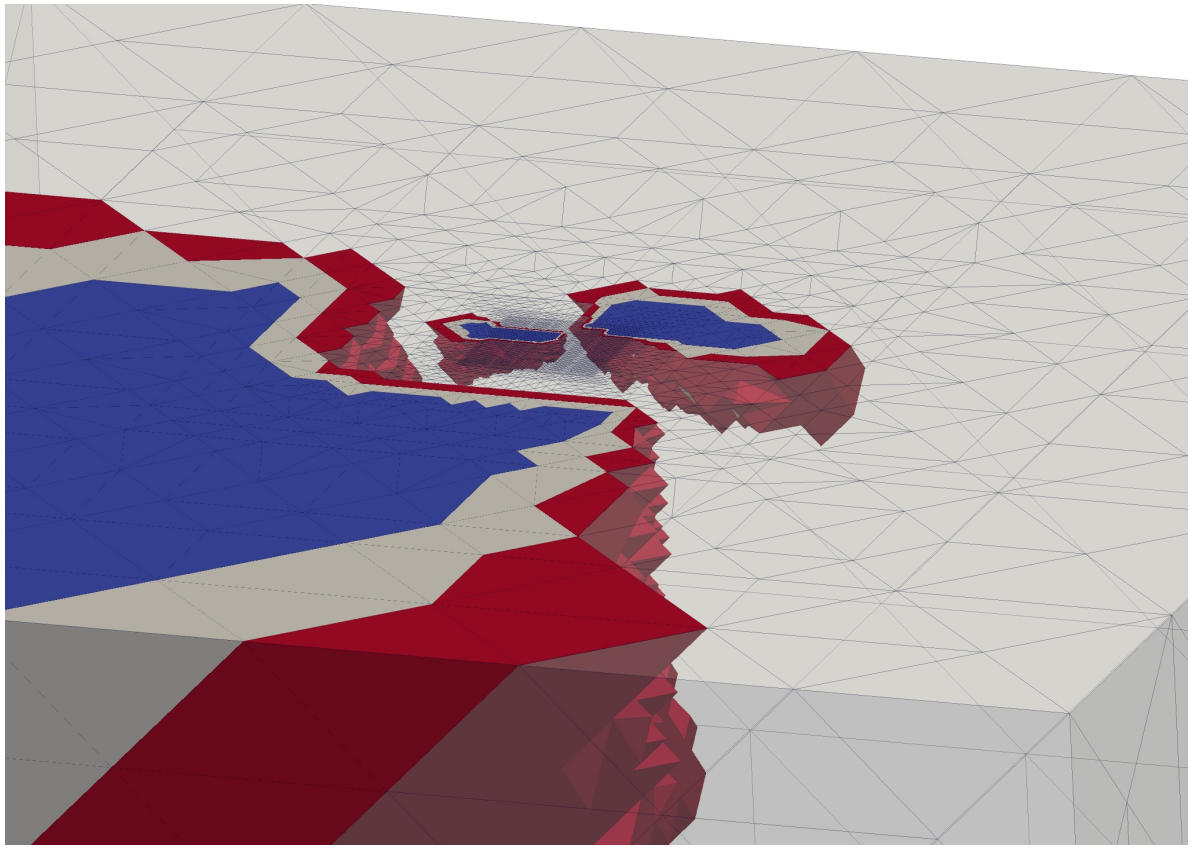
- Intel **Ivy Bridge**
  8 cores, 2.0 GHz, 32 KB L1, 256 KB L2, 16 MB L3
- **MPI-only:**
  `mpirun -np 16*numNodes ./program`
- **MPI+threading** ("hybrid"):
  `mpirun -np numNodes ./program -numThreads=16`

Domain decomposition example



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

Domain decomposition example



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

Moore's law $\Rightarrow$ "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?"

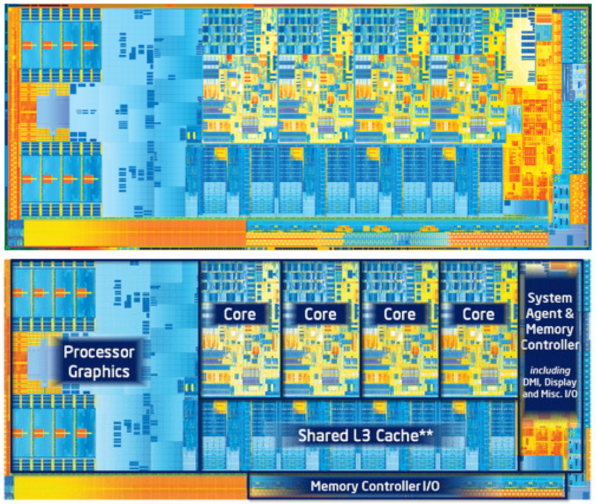
CPU strategy: make a single thread go faster

- 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**.

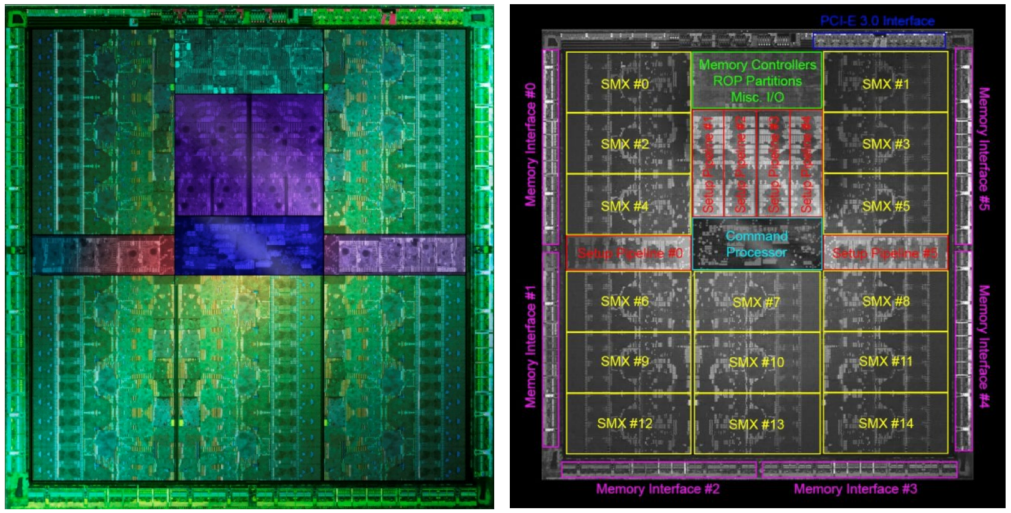
But what if instead we support many (>1000) simultaneous (slower) threads? ⇒ the **many-core revolution**

Ivy Bridge



anantech.com

Kepler



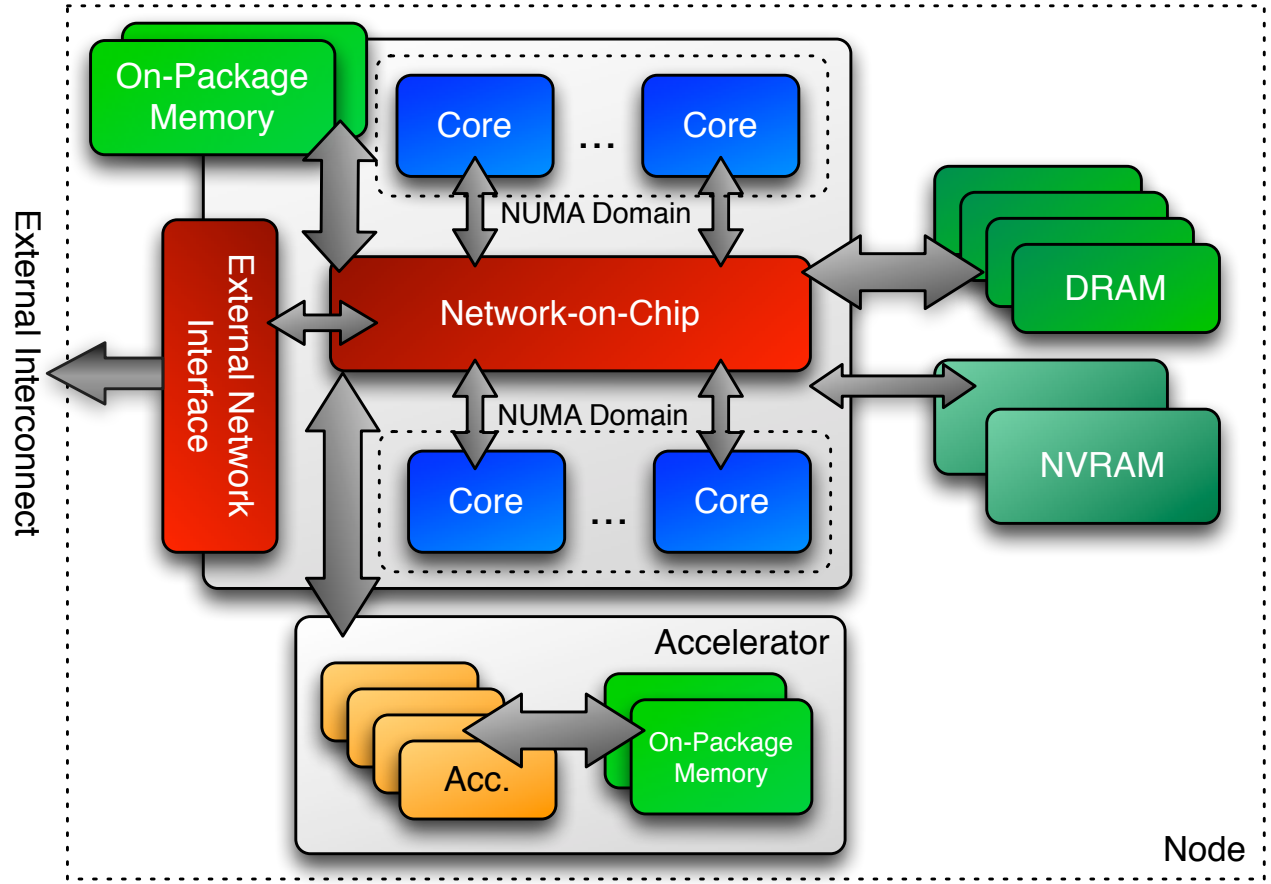
pcper.com

	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**

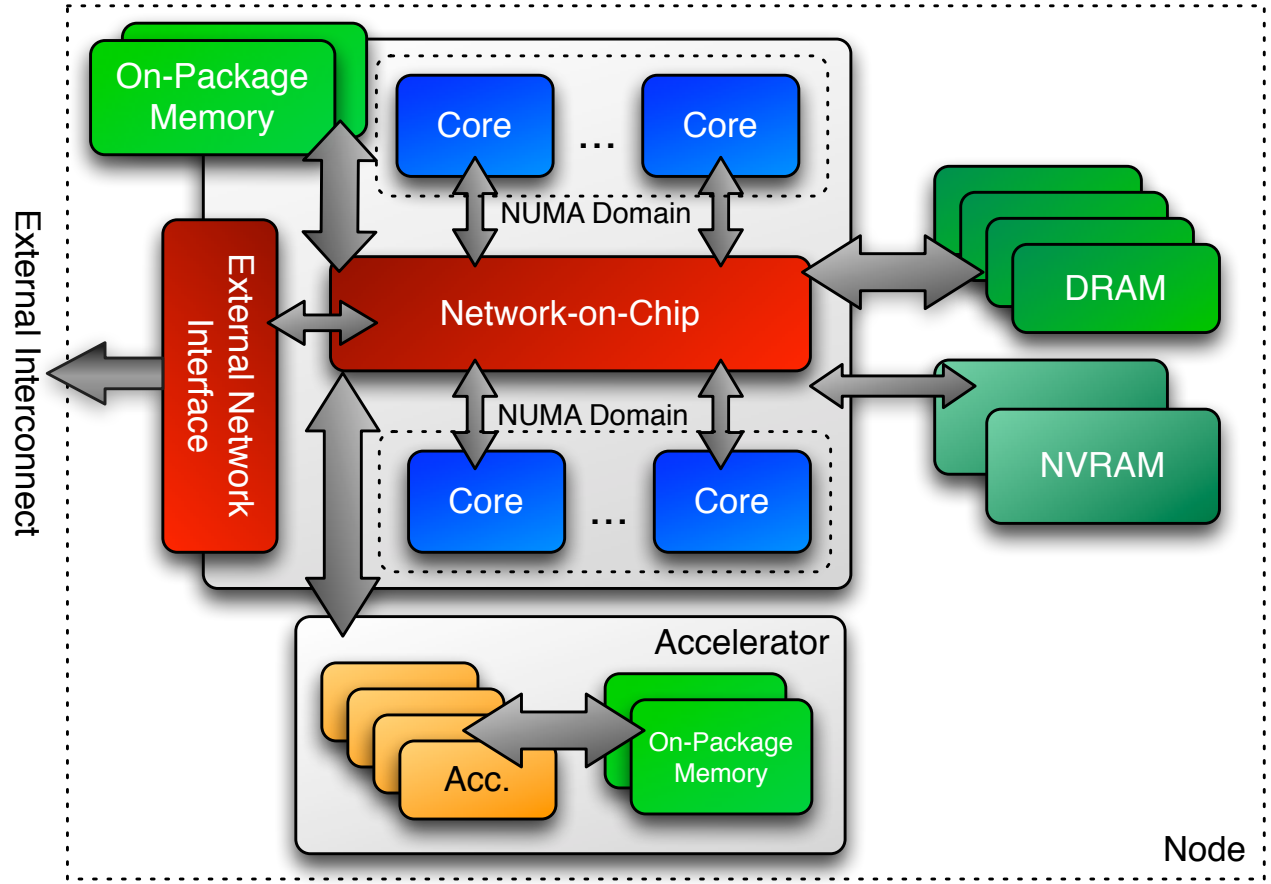
Key characteristics of GPUs:

- 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 **ship data**
- GPUs have expensive memory which provides **5-10X the bandwidth** to GPU memory as CPUs have to CPU memory.
- GPUs rely on parallelism instead of caches to **hide memory latency** with fast context switching.
- 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:



The 20-year "just recompile" **free ride is over**.

Operating assumptions:

- Compute nodes have ˜50 complex cores, ˜5000 simple cores, or any combination (**heterogenous** in cores *and* memory).
- **Heterogenous-node programming** on the node, **message-passing** between nodes.

Operating assumptions:

- Compute nodes have ~50 complex cores, ~5000 simple cores, or any combination (**heterogenous** in cores *and* memory).
- **Heterogenous-node programming** on the node, **message-passing** between nodes.

Problem: implementations may target particular architectures and are not *thread scalable*.

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

Operating assumptions:

- Compute nodes have ~50 complex cores, ~5000 simple cores, or any combination (**heterogenous** in cores *and* memory).
- **Heterogenous-node programming** on the node, **message-passing** between nodes.

Problem: implementations may target particular architectures and are not *thread scalable*.

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

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, or any combination (**heterogenous** in cores *and* memory).
- **Heterogenous-node programming** on the node, **message-passing** between nodes.

Problem: implementations may target particular architectures and are not *thread scalable*.

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

Solutions:

- Maintain **separate versions** for each target architecture (Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)
- Write in language or with a library that runs on multiple architectures (e.g., openmp, openacc, opencl, **kokkos**)

Operating assumptions:

- Compute nodes have ~50 complex cores, ~5000 simple cores, or any combination (**heterogenous** in cores *and* memory).
- **Heterogenous-node programming** on the node, **message-passing** between nodes.

Problem: implementations may target particular architectures and are not *thread scalable*.

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

Solutions:

- ~~Maintain **separate versions** for each target architecture (Xeon, Xeon Phi, GPU, GPU with NVLink, etc.)~~
- Write in language or with a library that runs on multiple architectures (e.g., openmp, openacc, opencl, **kokkos**)

Example: parallel loops

Pattern Policy

Body

```
for (size_t i = 0; i < N; ++i) {
  const double x = someFunction(i, ...);
  const double plasticUpdate = otherFunction(x, ...data...);
  plasticStrains[i] = (plasticUpdate > 0) ? ...
}
```

Terminology:

- **Pattern**: structure of user's computations
  for, reduction, scan, task-graph, ...
- **Execution Policy**: how computations are executed
  static scheduling, dynamic, thread teams, ...
- **Computational Body**: logic which performs one of the pieces of the work

The pattern and policy drive the computational body.

Data parallel loop bodies are prime candidates for parallelization.

Test: do you get the same answer if loop is run backwards?

Data parallel loop bodies are prime candidates for parallelization.

Test: do you get the same answer if loop is run backwards?

Examples:

- Forces in MD:

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

Data parallel loop bodies are prime candidates for parallelization.

Test: do you get the same answer if loop is run backwards?

Examples:

- Forces in MD:

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

- Thermodynamic quantities at quadrature points in FEA:

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

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

(This is a change in the *policy* from "serial" to "parallel.")

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

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

(This is a change in the *policy* from "serial" to "parallel.")

Openmp is very effective at threading for multi-core CPUs, but what if we then want to do this on a **GPU** too?

Option 1: OpenMP

```
#pragma omp target data map(...)
#pragma omp teams num_teams(...) num_threads(...) private(...)
#pragma omp distribute
for (element = 0; element < numberOfElements; ++element) {
  total = 0
#pragma omp parallel for
  for (qp = 0; qp < numberOfQuadraturePoints; ++qp) {
    for (i = 0; i < vectorSize; ++i) {
      total +=
        left[element*numberOfQuadraturePoints * vectorSize +
             qp * vectorSize +
             i] *
        right[element*numberOfQuadraturePoints * vectorSize +
              qp * vectorSize +
              i];
    }
  }
  elementValues[element] = total;
}
```

Option 2: OpenACC

```
#pragma acc parallel copy(...) num_gangs(...) vector_length(...)
#pragma acc loop gang vector
for (element = 0; element < numberOfElements; ++element) {
  total = 0;
  for (qp = 0; qp < numberOfQuadraturePoints; ++qp) {
    for (i = 0; i < vectorSize; ++i) {
      total +=
        left[element * numberOfQuadraturePoints * vectorSize +
             qp * vectorSize +
             i] *
        right[element * numberOfQuadraturePoints * vectorSize +
              qp * vectorSize +
              i];
    }
  }
  elementValues[element] = total;
}
```

Data layout problem: CPU memory access pattern reduces GPU performance by more than 10X.

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

Data layout problem: CPU memory access pattern reduces GPU performance by more than 10X.

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

⇒ For performance, memory layouts *must* depend on architecture.

How does Kokkos address these problems?

Kokkos is a *user-accessible*, *portable*, *performant*, *shared-memory* programming environment.

- is a templated C++ **library**, not a new language.
- 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.
- uses multi-dimensional arrays with architecture-dependent **layouts**
  i.e. it *solves the data layout problem*.

- 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 model and **library** allow you to run efficient code on multiple architectures with minimal changes to switch.
- Kokkos provides *thread-scalable* parallel **patterns** and execution **policies** for flexible algorithm expression.
- Kokkos **solves the data layout** problem with multi-dimensional arrays tuned to the underlying memory system.

Data parallel patterns and work

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

Data parallel patterns and work

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

- Kokkos maps "work" to cores, but **what is "work?"**

Data parallel patterns and work

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

- Kokkos maps "work" to cores, but **what is "work?"**
  - Indices for the **body** to perform.
- Given a total number of iterations, Kokkos intelligently maps **indices** to cores

Data parallel patterns and work

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

- Kokkos maps "work" to cores, but **what is "work?"**
  - Indices for the **body** to perform.
- Given a total number of iterations, Kokkos intelligently maps **indices** to cores

So, what information must we provide to Kokkos?

Data parallel patterns and work

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

- Kokkos maps "work" to cores, but **what is "work?"**
  - Indices for the **body** to perform.
- Given a total number of iterations, Kokkos intelligently maps **indices** to cores

So, what information must we provide to Kokkos?

Important concept: Work mapping

You give a **computational body** to Kokkos, Kokkos runs the body across cores, giving it indices of work to perform.

What is a computational body?

User code (instructions), but in what form?

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

What is a computational body?

User code (instructions), but in what form?

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

Defining the functor:

```
struct Body {
  ...
  void operator()(somehow, an index assignment) const {
  }
  ...
};
```

Using the functor:

```
Body body;
Kokkos::parallel_for(numberOfIterations, body);
```

Passing indices to bodies

Intel Threading Building Blocks' approach:

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

Passing indices to bodies

Intel Threading Building Blocks' approach:

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

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

Kokkos uses the **simplest interface** possible:

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

Passing indices to bodies

Intel Threading Building Blocks' approach:

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

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

Kokkos uses the **simplest interface** possible:

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

Warning: concurrency and order

No concurrency or order is guaranteed by the Kokkos runtime.

Passing data to bodies (functors)

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

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

Passing data to bodies (functors)

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

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

How does the body access the data?

Passing data to bodies (functors)

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

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

How does the body access the data?

Important concept

The bodies (functors) must have access to all the data they need through **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 the functor**?

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

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

The complete picture (using functors):

Defining the functor (operator+data):

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

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

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

Using the functor:

```
Body body(atomForces, data);
Kokkos::parallel_for(numberOfAtoms, body);
```

Lambdas

C++11 introduces support for anonymous functors.
("lambdas," "closures")

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

Lambdas

C++11 introduces support for anonymous functors.
("lambdas," "closures")

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

Warning: Lambdas

Kokkos lambdas must capture variables by value

Lambdas

C++11 introduces support for anonymous functors.
("lambdas," "closures")

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

Warning: Lambdas

Kokkos lambdas must capture variables by value

Note: lambdas are **not magic**, they are simply **auto-generated functors**.

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;
}
```

Example: saxpy

OpenMP

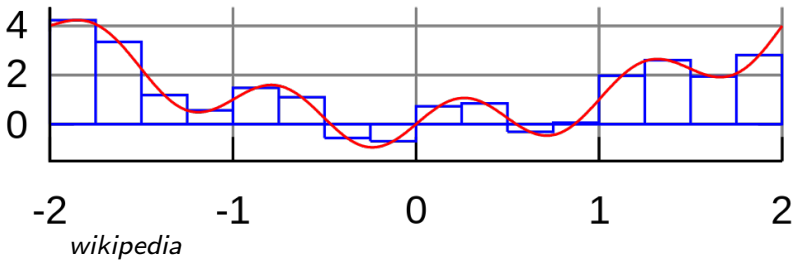
```
double * x = new double[N]; // also y
#pragma omp parallel for
for (size_t i = 0; i < N; ++i) {
  y[i] = a * x[i] + y[i];
}
```

Kokkos

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

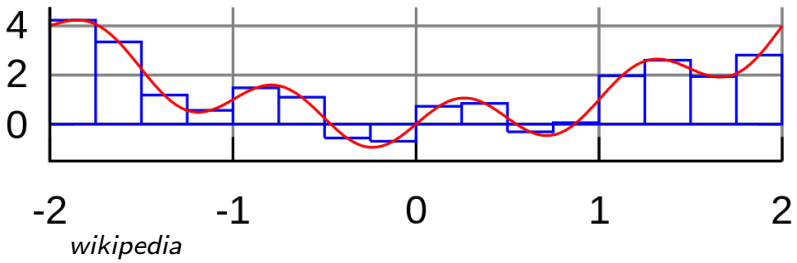
Riemann-sum-style numerical integration:

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



Riemann-sum-style numerical integration:

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



Pattern

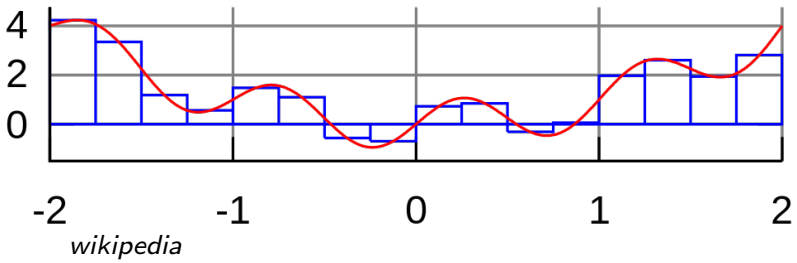
Policy

Body

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

Riemann-sum-style numerical integration:

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



Pattern

Policy

Body

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

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!)

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 of a set of threads.

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

Important concept: Reduction

Reductions combine the results of a set of threads.

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 of a set of threads.

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 += ...
}
```

Example: Scalar integration

OpenMP

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

Kokkos

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

Example: Maximum point norm

OpenMP

```
double maxPointNorm = 0;
#pragma omp parallel for reduction(max:maxPointNorm)
for (size_t i = 0; i < numberOfPoints; ++i) {
  maxPointNorm = std::max(maxPointNorm, norm(points[i]));
}
```

Kokkos

```
double maxPointNorm = 0;
parallel_reduce(numberOfPoints,
  [=] (const size_t i, double & valueToUpdate) {
    valueToUpdate = std::max(valueToUpdate, norm(points[i]));
  },
  maxPointNorm);
```

Parallel_reduce (using lambdas):

```
ReductionType reducedValue = 0;
Kokkos::parallel_reduce(numberOfIterations,
  [=] (const size_t index,
       ReductionType & valueToUpdate) {
    valueToUpdate = [operator-specific logic]
  },
  reducedValue);
```

Parallel_reduce (using lambdas):

```
ReductionType reducedValue = 0;
Kokkos::parallel_reduce(numberOfIterations,
  [=] (const size_t index,
       ReductionType & valueToUpdate) {
    valueToUpdate = [operator-specific logic]
  },
  reducedValue);
```

Limitation: the reduced value starts at 0 and is combined with `operator+`.

Parallel_reduce (using lambdas):

```
ReductionType reducedValue = 0;
Kokkos::parallel_reduce(numberOfIterations,
  [=] (const size_t index,
       ReductionType & valueToUpdate) {
    valueToUpdate = [operator-specific logic]
  },
  reducedValue);
```

Limitation: the reduced value starts at 0 and is combined with `operator+`.

For anything else, you need to use a **general reduction** functor.

How do you do reductions on arbitrary types?

Example: finding center of mass of particles

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

OpenMP 3.1: Not supported

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 centerOfMass = {{0., 0., 0.}};
#pragma omp parallel for reduction(+:centerOfMass)
for (size_t i = 0; i < numberOfPoints; ++i) {
  centerOfMass += points[i];
}
centerOfMass /= numberOfPoints;
```

You don't have to use the reduction clause:

Manual reductions:

```
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:

```
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;
```

Problem: false sharing

Correct (CPU) manual reductions:

```
double totalIntegral = 0;
#pragma omp parallel
{
  double localIntegral = 0;
#pragma omp for
  for (size_t i = 0; i < numberOfIntervals; ++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 < numberOfIntervals; ++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 environments 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`")

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_types`

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_types`
- How to **initialize** a `value_type`

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_types
- How to **initialize** a value_type

```
struct Body {
  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 {...}
}
```

Execution spaces

How do I control where parallel kernels are run?

Thought experiment: Consider this kernel execution:

section 1

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

section 2

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

Thought experiment: Consider this kernel execution:

section 1

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

section 2

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

- Where will section 1 be run? CPU? GPU? Xeon Phi?
- Where will section 2 be run? CPU? GPU? Xeon Phi?
- In general, how do I **control** where code is executed?

Thought experiment: Consider this kernel execution:

section 1

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

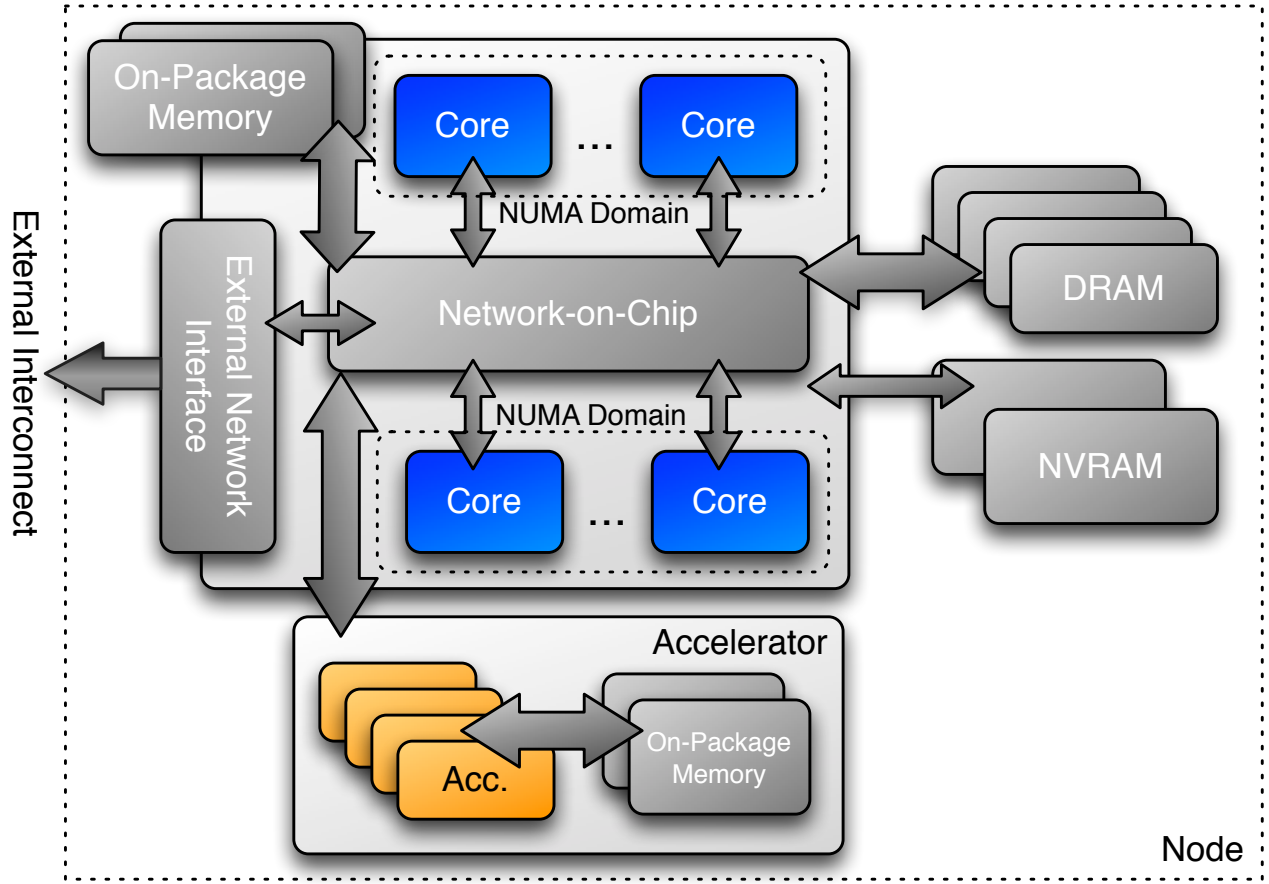
section 2

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

- Where will section 1 be run? CPU? GPU? Xeon Phi?
- Where will section 2 be run? CPU? GPU? Xeon Phi?
- In general, how do I **control** where code is executed?

$\Rightarrow$ **Execution spaces**

Execution space: logical grouping of identical computational units



Execution spaces:

`Serial`, `Threads`, `OpenMP`, `Cuda`, ... more to come

Execution spaces (2)

Host

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

Parallel

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

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 be run? CPU? GPU? Xeon Phi?

  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 be run? CPU? GPU? Xeon Phi?
  The **host process**
- Where will Parallel be run? CPU? GPU? Xeon Phi?
  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 be run? CPU? GPU? Xeon Phi?
  The **host process**
- Where will Parallel be run? CPU? GPU? Xeon Phi?
  The **default execution space**
- In general, how do I **control** where Parallel is executed?
  Changing the default execution space (**compilation**), or specifying an execution space in the **policy**.

Changing the parallel execution space:

Default

```
parallel_for(
  numberOfIntervals,
  [=] (const size_t i) {
    ...
  });
```

Custom

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

Changing the parallel execution space:

Default

```
parallel_for(
  numberOfIntervals,
  [=] (const size_t i) {
    ...
  });
```

Custom

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

Details necessary to enable different execution spaces:

- Kokkos must be **compiled** with the execution spaces enabled.
- Kokkos must be **initialized** and **finalized** within programs.
- Kernels (and called **functions**) must be marked with a **macro**.
- **Lambdas** must be marked with a **macro**.

Initializing and finalizing Kokkos:

```
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

Kokkos portability macros:

KOKKOS_LAMBDA annotates lambdas:

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

Kokkos portability macros:

KOKKOS_LAMBDA annotates lambdas:

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

KOKKOS_INLINE_FUNCTION annotates functions:

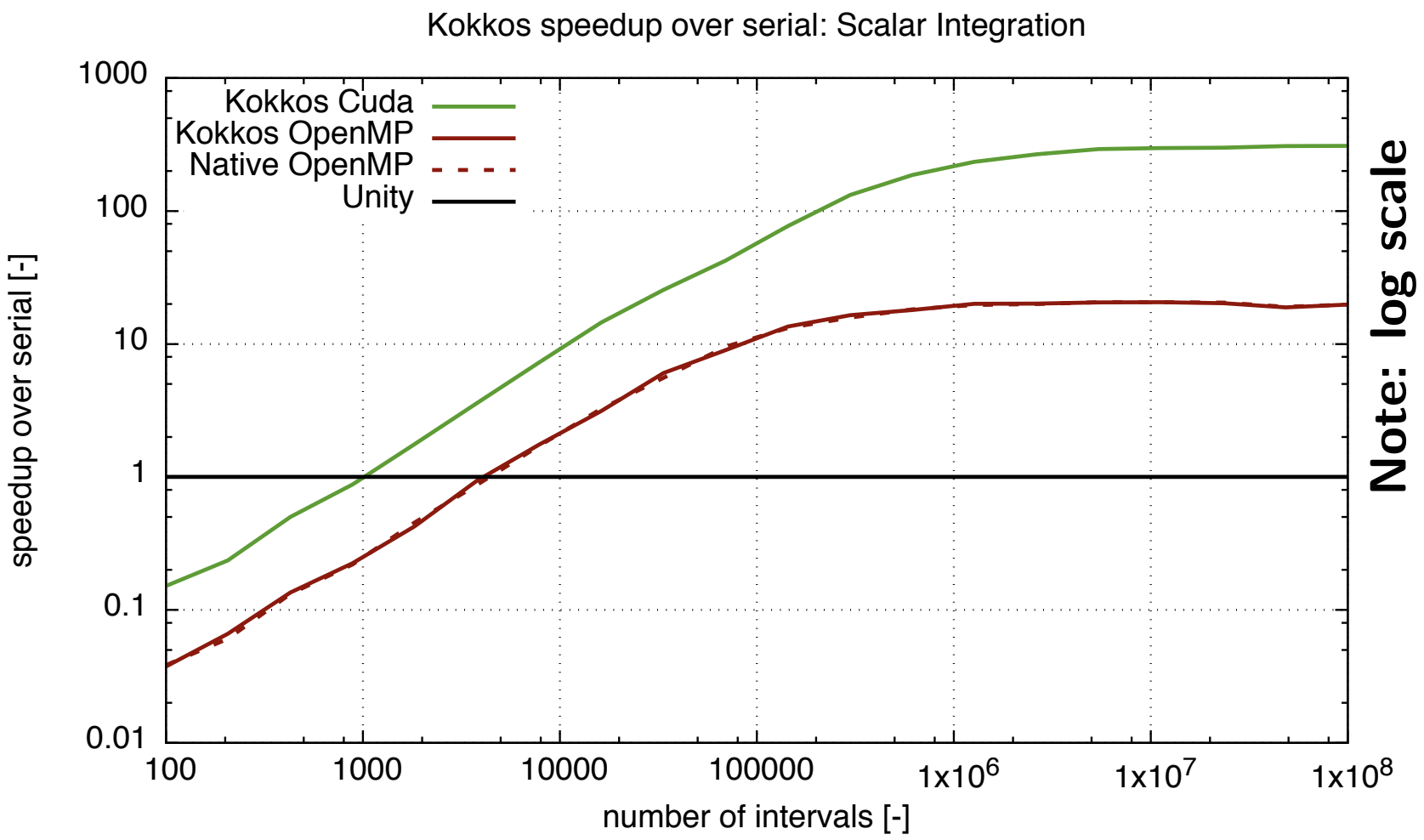
```
KOKKOS_INLINE_FUNCTION
double helperFunction(const size_t s) {...}

struct Body {
  ...
  KOKKOS_INLINE_FUNCTION
  void operator()(const size_t index) const {
    ...
    helperFunction(index);
  }
}
```

Task: Implement scalar integration using Kokkos.

Details:

- All files are contained in `Examples/ScalarIntegration`.
- `ScalarIntegration.cc` contains all testing and timing logic.
- `ScalarIntegration.cc` calls functions defined in `Version_Serial.cc` and `Version_Kokkos.cc`.
- Only `Version_Kokkos.h` needs modification.
- Compile with `make`, run with `./ScalarIntegration`, generate plot with `gnuplot makePlot.gnuplot`.

Results:


Kokkos speedup over serial: Scalar Integration
Kokkos Cuda
Kokkos OpenMP
Native OpenMP
Unity
speedup over serial [-]
1000
100
10
1
0.1
0.01
100
1000
10000
100000
1x10^6
1x10^7
1x10^8
number of intervals [-]
Note: log scale

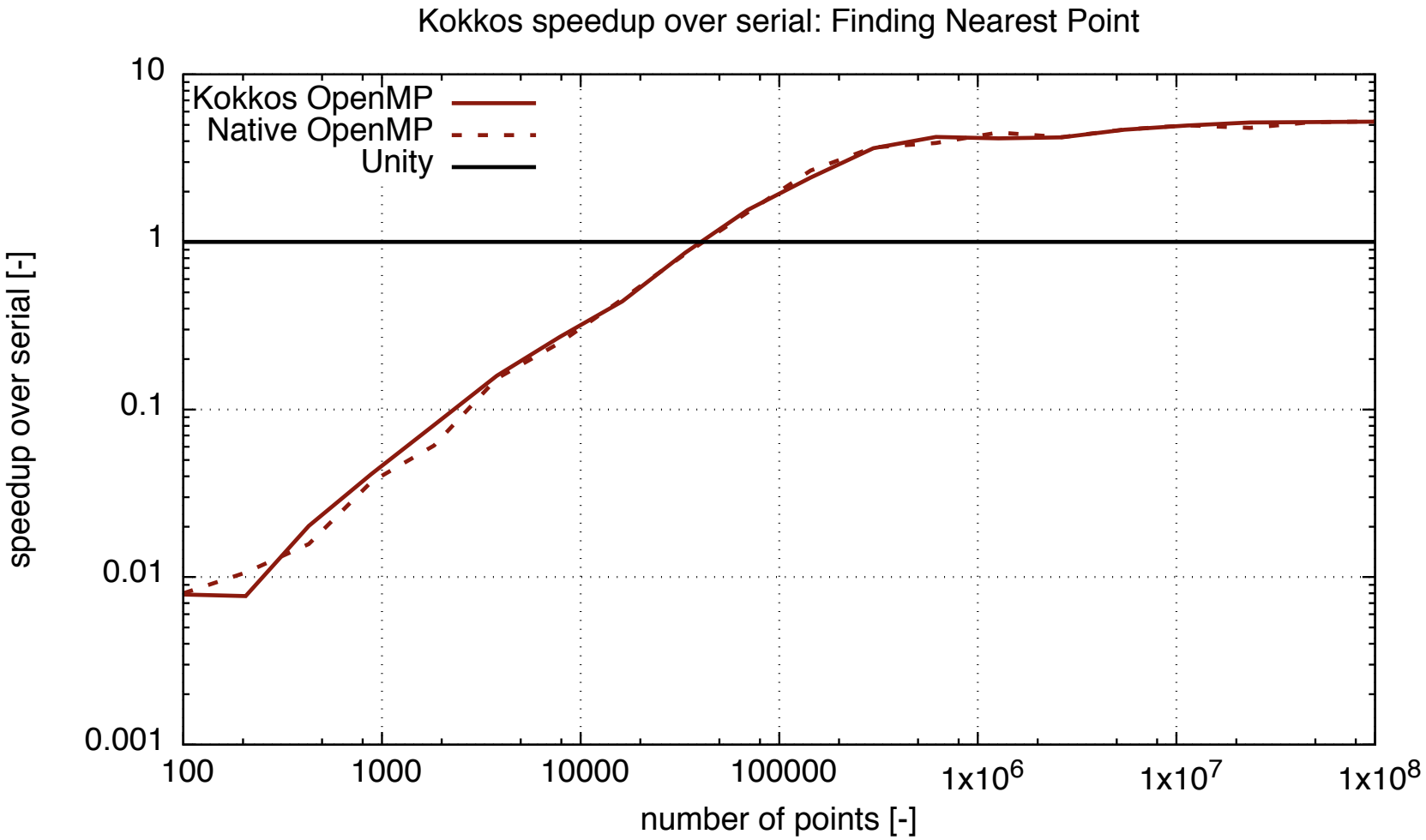
Exercise: finding nearest point (0)

Task: Finding the index of a point in a set that is closest to a search location

Details:

- All files are contained in `Examples/NearestPoint`.
- `NearestPoint.cc` contains all testing and timing logic.
- `NearestPoint.cc` calls functions defined in `Version_Serial.cc` and `Version_Kokkos.cc`.
- Only `Version_kokkos.h` needs modification.
- Compile with `make`, run with `./NearestPoint`, generate plot with `gnuplot makePlot.gnuplot`.
- Hint: You'll have to pass the points to the functor as data.

Results:


Kokkos speedup over serial: Finding Nearest Point
Kokkos OpenMP
Native OpenMP
Unity
speedup over serial [-]
10
1
0.1
0.01
0.001
100
1000
10000
100000
1x10^6
1x10^7
1x10^8
number of points [-]

- 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*.

```
Body body(...);
parallel_scan(RangePolicy<Space>(0, N), body);
```

- 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*.

```
Body body(...);
parallel_scan(RangePolicy<Space>(0, N), body);
```

- 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*.

```
Body body(...);
parallel_scan(RangePolicy<Space>(0, N), body);
```

- 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*.

```
Body body(...);
parallel_scan(RangePolicy<Space>(0, N), body);
```

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

- **Simple** use is **similar to openmp**, but advanced flexibility is available
- Three common **data-parallel patterns** are `parallel_for`, `parallel_reduce`, and `parallel_scan`.
- A parallel dispatch is characterized by its **pattern**, **policy**, and **body**.
- **User computational bodies** are provided as functors or lambdas, which handle a single item of the work.
- Heterogenous nodes have one or more **execution spaces**.
- You control in which execution space parallel code is run by a **template parameter** on the policy or by changes in **compilation**.

Views

Learning objectives:

- Motivation behind the `View` abstraction.
- Key `View` concepts and 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 Body {
  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 Body {
  const Point * _points;

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

Problem: `_points` (data *and* the pointer) reside in CPU memory.

Why couldn't we run `NearestPoint` on the GPU?

`Version_kokkos.h`:

```
struct Body {
  const Point * _points;

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

Problem: `_points` (data *and* the pointer) reside in CPU memory.

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

Why couldn't we run `NearestPoint` on the GPU?

`Version_kokkos.h`:

```
struct Body {
  const Point * _points;

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

Problem: `_points` (data *and* the pointer) reside in CPU memory.

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

$\Rightarrow$ **Views**

High-level usage of Views in NearestPoints:

```
struct Body {
  const View<...> _points;

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

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


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

Body body(points);
parallel_reduce(N, body);
```

`View` **overview:**

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

`View` overview:

- **Multi-dimensional arrays** of 0 or more dimensions
  scalar (0), vector (1), matrix (2), etc.
- Arrays are **rectangular**, not ragged.
- **Number of dimensions (rank)** is fixed at compile-time.
  e.g., scalar, vector, matrix, etc.
- **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 does this snippet print?

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 does this snippet print?

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?

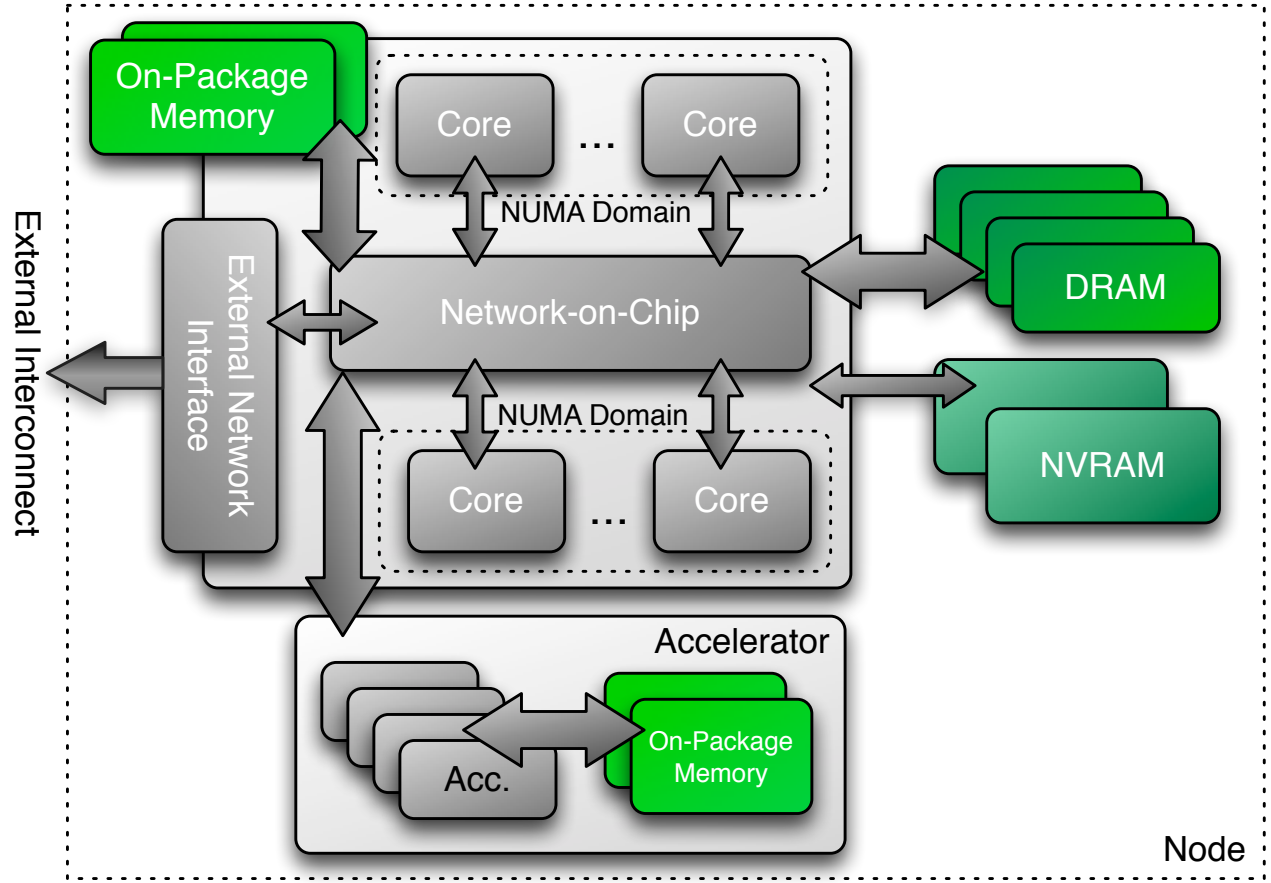
$\Rightarrow$ **Memory Spaces**

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: how would you define this, Carter?



Important concept: Memory spaces

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

Important concept: Memory spaces

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

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

Important concept: Memory spaces

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

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

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

Important concept: Memory spaces

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

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

  `HostSpace, CudaSpace, CudaUVMSpace, ...` more
- Each **execution space** has a default memory space, which is used if you pass the execution space

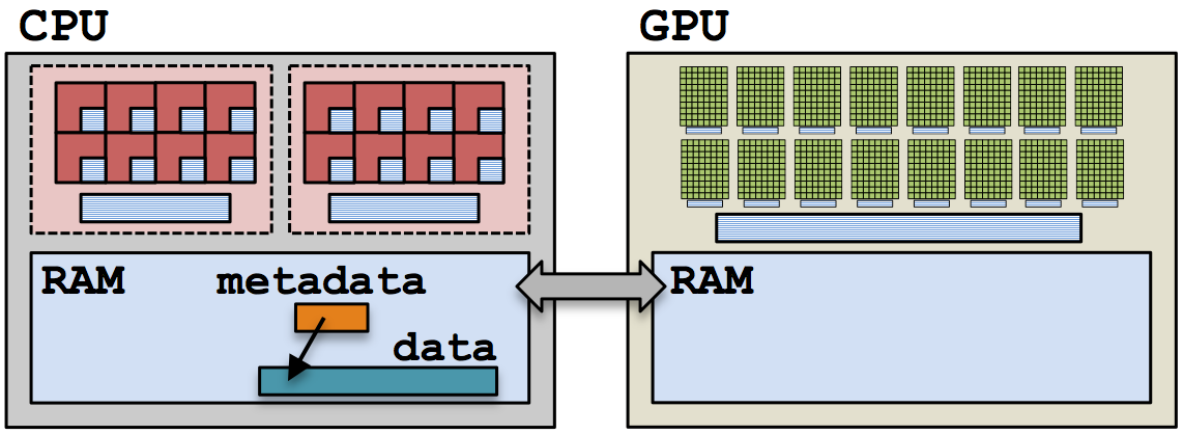
Important concept: Memory spaces

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

- `View<double***, [Memory]Space> data(...);`
- Available **memory spaces**:
  `HostSpace, CudaSpace, CudaUVMSpace, ...` more
- Each **execution space** has a default memory space, which is used if you pass the execution space
- If no `Space` is provided, the view is made in the **default memory space** of the **default execution space**.

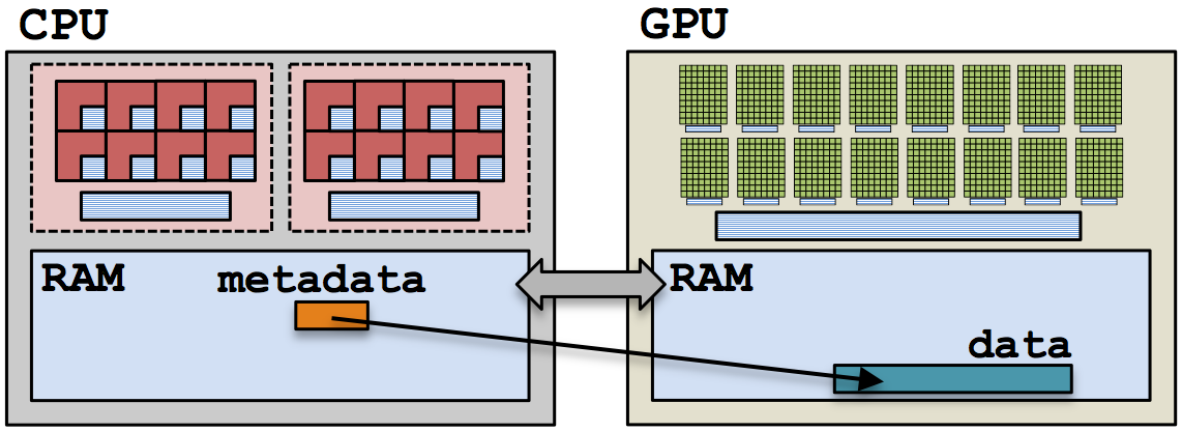
Example: HostSpace

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



Example: CudaSpace

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



Example (redux): summing an array with the GPU

(failed) Attempt 1:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 1:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2:

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

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

Example (redux): summing an array with the GPU

(failed) Attempt 2:

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

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

What's the solution?

Important concept: Mirrors

Mirrors are views into (conceptually) the same data, from different execution spaces.

Important concept: Mirrors

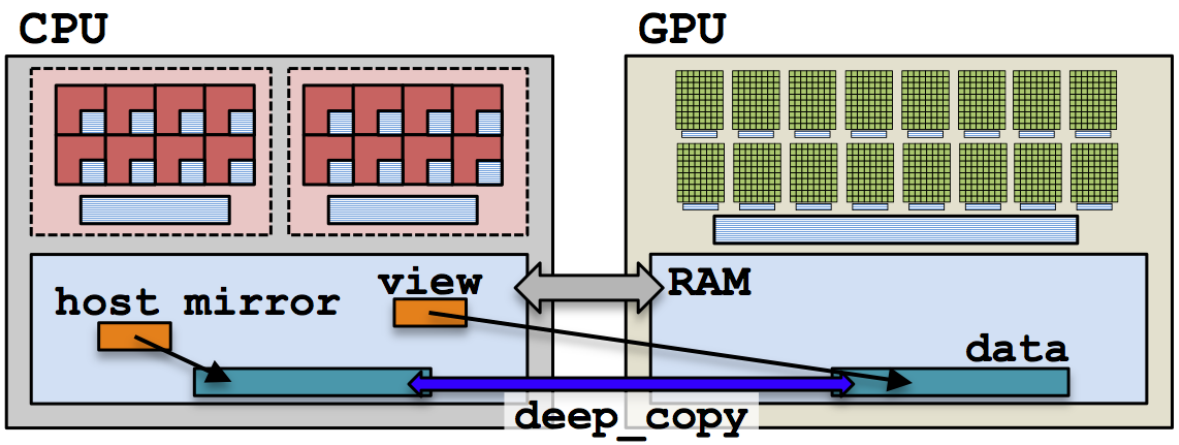
Mirrors are views into (conceptually) the same data, from different execution spaces.

Common pattern:

1. **Allocate** a `view`.
2. **Make** a host mirror of the view, `hostView`.
3. **Populate** `hostView` on the host (from file, etc.).
4. **Deep copy** contents of `hostView` to `view`.
5. **Launch** a kernel processing with the `view`.
6. If needed, **deep copy** the (updated) contents of `view` back to the `hostView` and use them (write file, etc.).

Mirroring schematic

```
typedef Kokkos::View<double**, Device> DeviceViewType;
DeviceViewType deviceView(...);
DeviceViewType::HostMirror hostView =
  Kokkos::create_mirror_view(deviceView);
```



Syntax:

```
typedef Kokkos::View<double*, Device> DeviceViewType;
DeviceViewType deviceView(...);
DeviceViewType::HostMirror hostView =
  Kokkos::create_mirror_view(deviceView);

populate hostView somehow

Kokkos::deep_copy(deviceView, hostView);

Kokkos::parallel_for(
  RangePolicy<Device>(0, size),
  KOKKOS_LAMBDA (...) { use and change deviceView });

Kokkos::deep_copy(hostView, deviceView);

post-process hostView somehow
```

```
typedef Kokkos::View<double*, Device> DeviceViewType;
DeviceViewType deviceView(``test'', 10);
DeviceViewType::HostMirror hostView =
  Kokkos::create_mirror_view(deviceView);

hostView(0) = 3.14;

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

Experiment: What is printed if `Device` is `Cuda`? `OpenMP`?

```
typedef Kokkos::View<double*, Device> DeviceViewType;
DeviceViewType deviceView(``test'', 10);
DeviceViewType::HostMirror hostView =
  Kokkos::create_mirror_view(deviceView);

hostView(0) = 3.14;

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

Experiment: What is printed if `Device` is `Cuda`? `OpenMP`?

Mirror **details**:

- `create_mirror_view` does not allocate unless necessary to go *across* memory spaces, otherwise it's a **reference**.
- `create_mirror` **always** allocates, even in the *same* space.
- Kokkos *never* performs a **hidden deep copy**.

Exercise: finding nearest point with views (0)

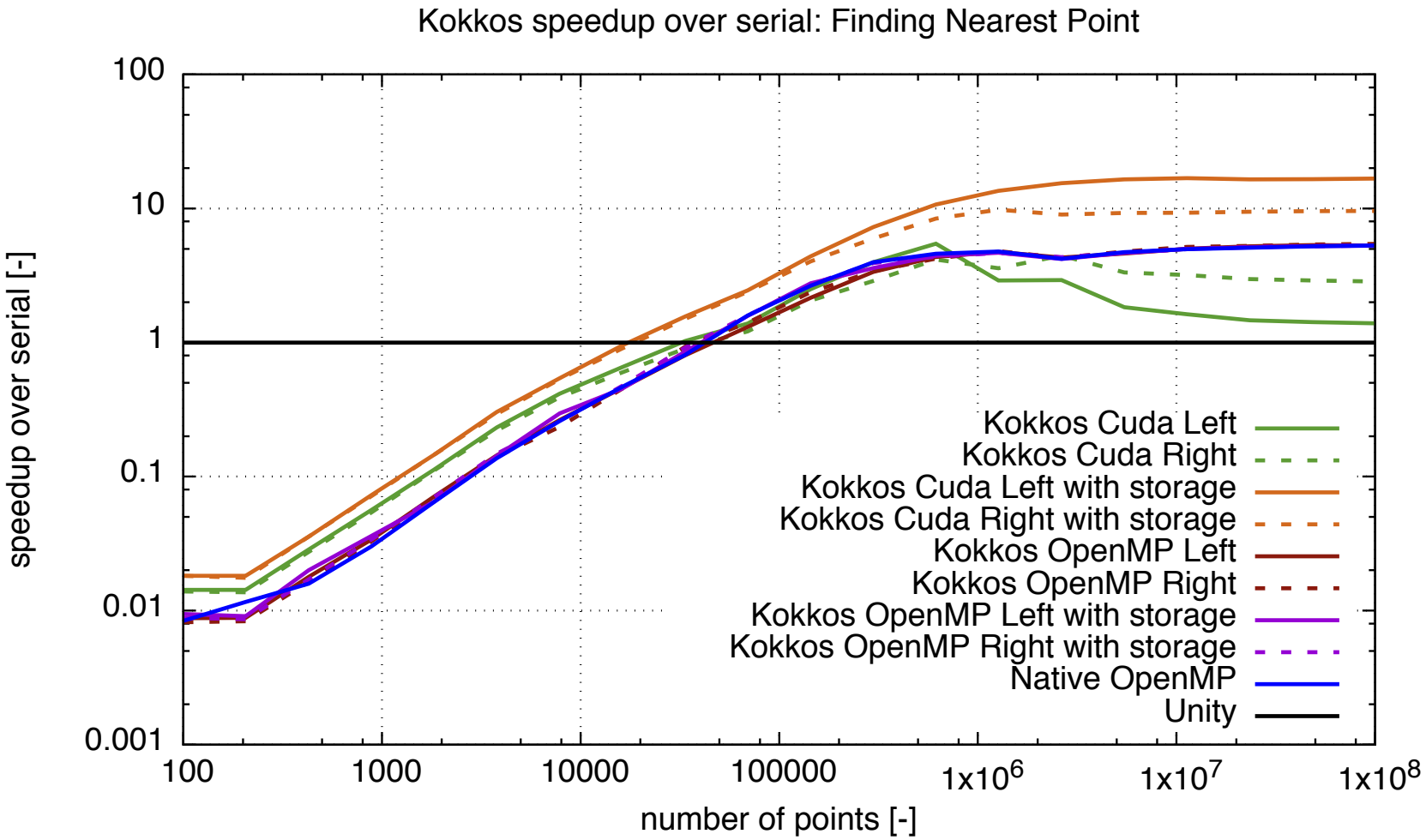
Task: Revisit `NearestPoint` and plot Cuda speedup using views.

Details:

- Modify your previous work in `Examples/NearestPoint`.
- Main logic of the `KokkosFunctor`

```
struct KokkosFunctor {
  PointViewType          _points;
  SearchLocationViewType _searchLocation;

  KokkosFunctor(Point* points, Point searchLocation) :
    _points(...), _searchLocation(...) {
    // create a host mirror of _points
    // populate points_host
    // deep_copy(_points, points_host);
    // do the same for the search location
  }
  operator()(size_t * result) const {
    parallel_reduce(...);
  }
```

Results:


Kokkos speedup over serial: Finding Nearest Point
speedup over serial [-]
100
10
1
0.1
0.01
0.001
100
1000
10000
100000
1x10^6
1x10^7
1x10^8
number of points [-]
Kokkos Cuda Left
Kokkos Cuda Right
Kokkos Cuda Left with storage
Kokkos Cuda Right with storage
Kokkos OpenMP Left
Kokkos OpenMP Right
Kokkos OpenMP Left with storage
Kokkos OpenMP Right with storage
Native OpenMP
Unity

Caching and coalescing

Learning objectives:

- Thread (in)dependence on CPU and GPU architectures.
- The need for coalesced memory access on the GPU.
- How memory access patterns relate to Kokkos mapping parallel indices to computational bodies.
- How the `Layout` template parameter is used to achieve good memory performance on different architectures.
- See a concrete example of the performance of various memory configurations.

Consider the array summation example:

```
View<double*, Device> data(``data'', size);
...populate data...

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

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

Consider the array summation example:

```
View<double*, Device> data(``data'', size);
...populate data...

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

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

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

Consider the array summation example:

```
View<double*, Device> data(``data'', size);
...populate data...

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

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

Chunked:	Strided:
`0, 1, 2, ..., N/P`	`0, N/P, 2*N/P, ...`
CPU	**GPU**

Consider the array summation example:

```
View<double*, Device> data(``data'', size);
...populate data...

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

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

Chunked:	Strided:
`0, 1, 2, ..., N/P`	`0, N/P, 2*N/P, ...`
CPU	**GPU**

Why?

Thread independence:

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

- **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

```
const double d = _data(index);
```

before *any* thread can move on.

Thread independence:

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

- **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.

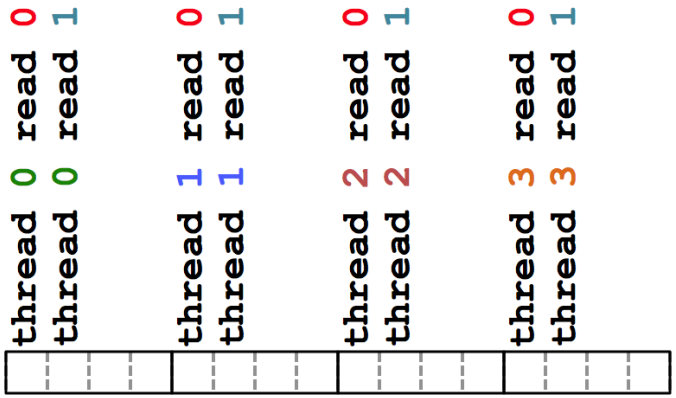
In particular all threads in a group (*warp*) must finished their loads

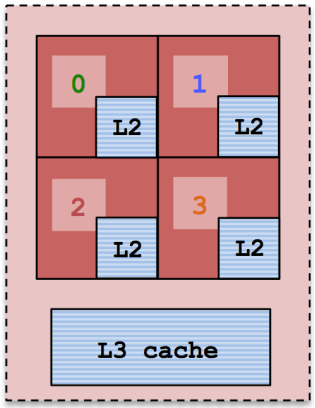
```
const double d = _data(index);
```

before *any* thread can move on.

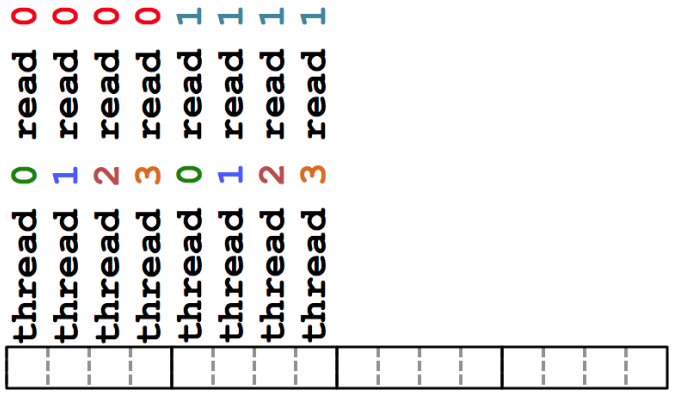
How many cache lines must be fetched before threads can move on?

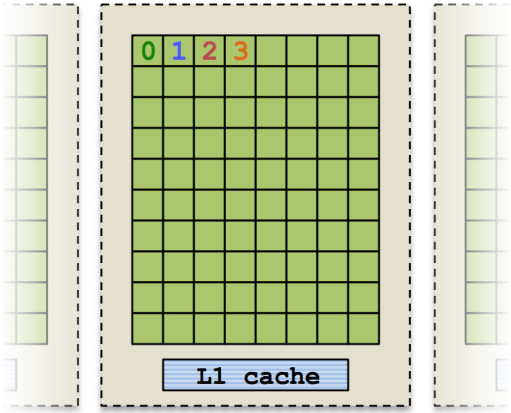
CPUs: few (independent) cores with separate caches:





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`.

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* access in `CudaSpace` is okay through Kokkos `const` views (more later).

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?

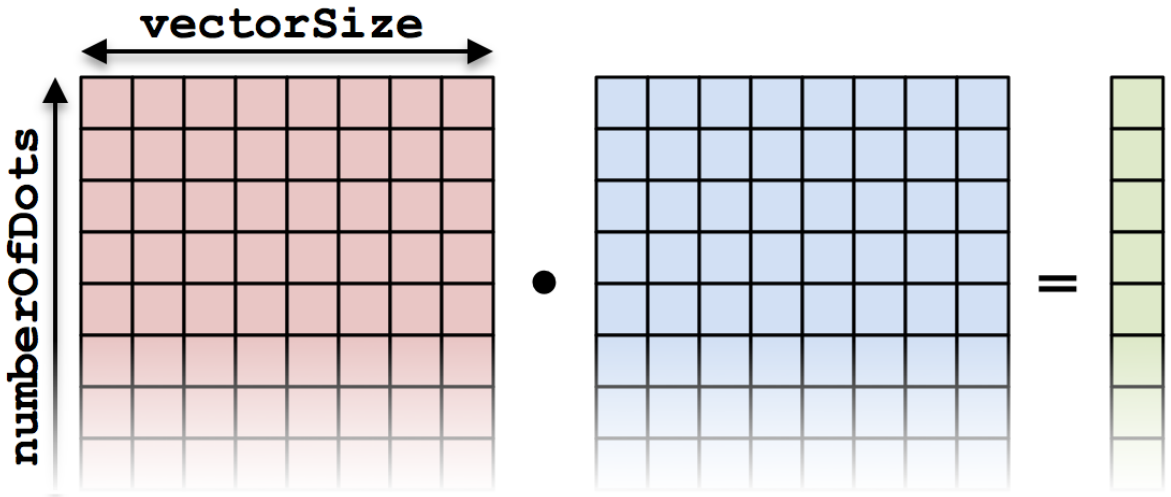
Important point

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

Example: array of dot products (0)

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

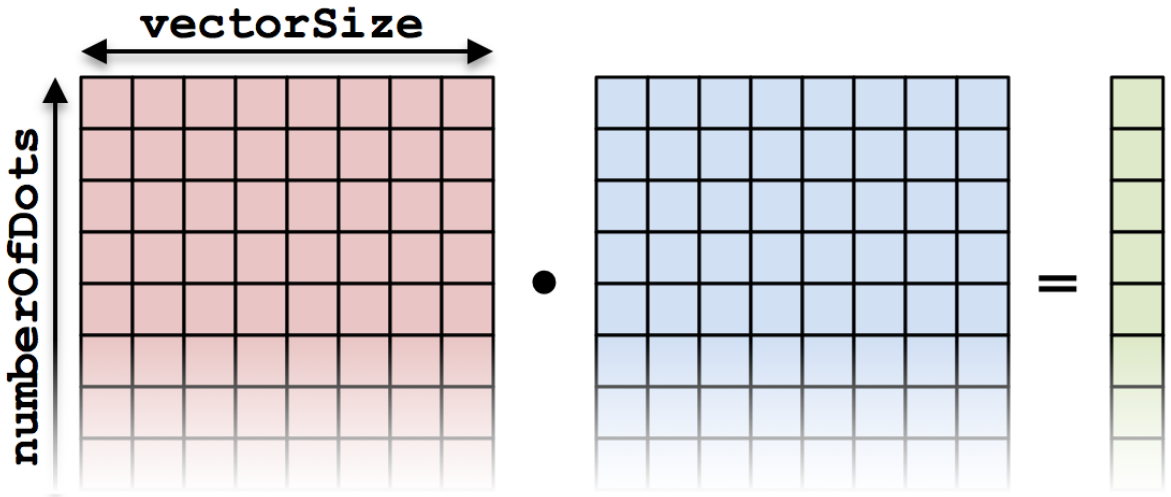
```
Kokkos::parallel_for(
  RangePolicy<ExecutionSpace>(0, numberOfDots),
  KOKKOS_LAMBDA (const size_t dotIndex) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(dotIndex, i) * right(dotIndex, i);
    }
    dotProducts(dotIndex) = total; });
```



Example: array of dot products (0)

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

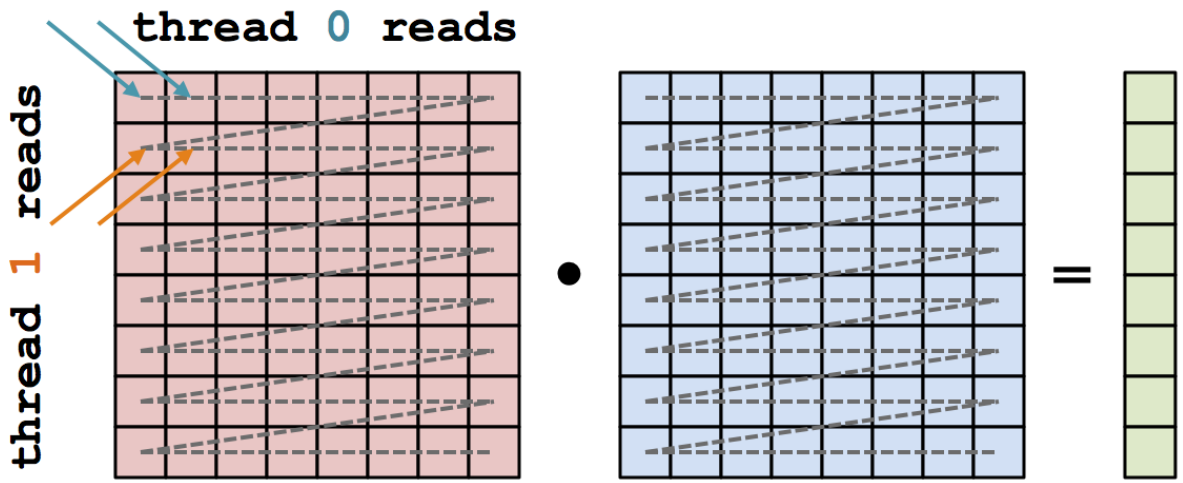
```
Kokkos::parallel_for(
  RangePolicy<ExecutionSpace>(0, numberOfDots),
  KOKKOS_LAMBDA (const size_t dotIndex) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(dotIndex, i) * right(dotIndex, i);
    }
    dotProducts(dotIndex) = total; });
```



How do we represent `left` and `right`?

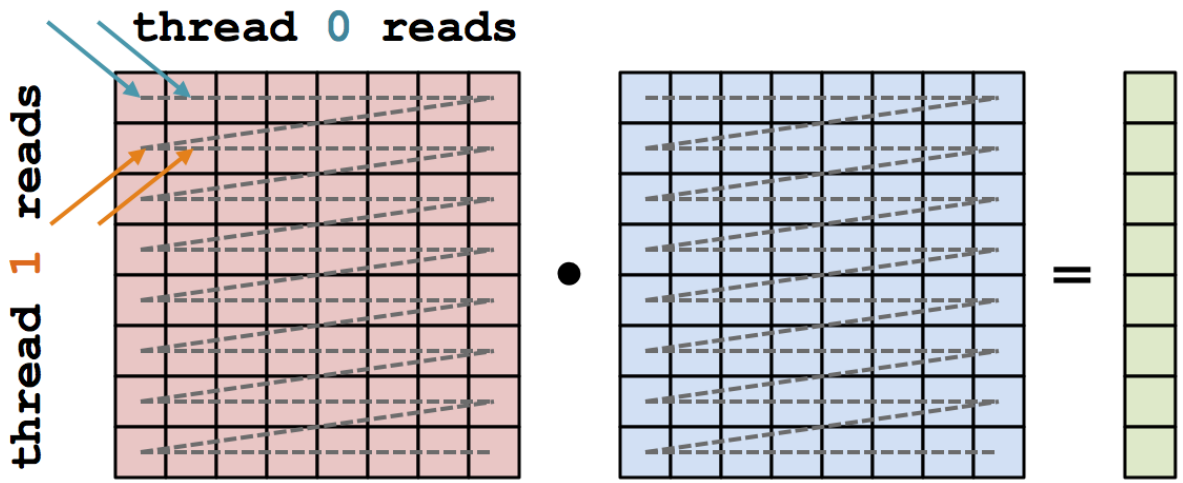
Attempt 0: 1D array, row-major

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



Attempt 0: 1D array, row-major

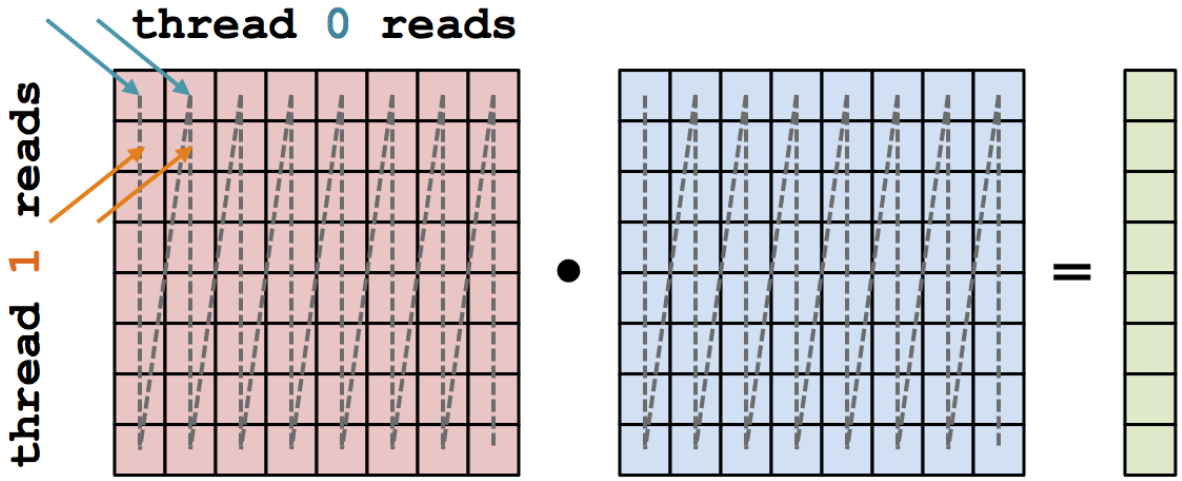
```
View<double*, ExecutionSpace> left(numberOfDots*vectorSize);
View<double*, ExecutionSpace> dotProducts(numberOfDots);
parallel_for(RangePolicy<ExecutionSpace>(0, numberOfDots),
  ...
      total +=
        left (dotIndex * vectorSize + i) *
        right(dotIndex * vectorSize + i);
```



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

Attempt 1: 1D array, col-major

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



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

Can we obtain good memory access on **both** architectures?

Can we obtain good memory access on **both** architectures?

Yes, by making memory layout *depend on* the architecture.

Important concept: Layouts

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

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

Can we obtain good memory access on **both** architectures?

Yes, by making memory layout *depend on* the architecture.

Important concept: Layouts

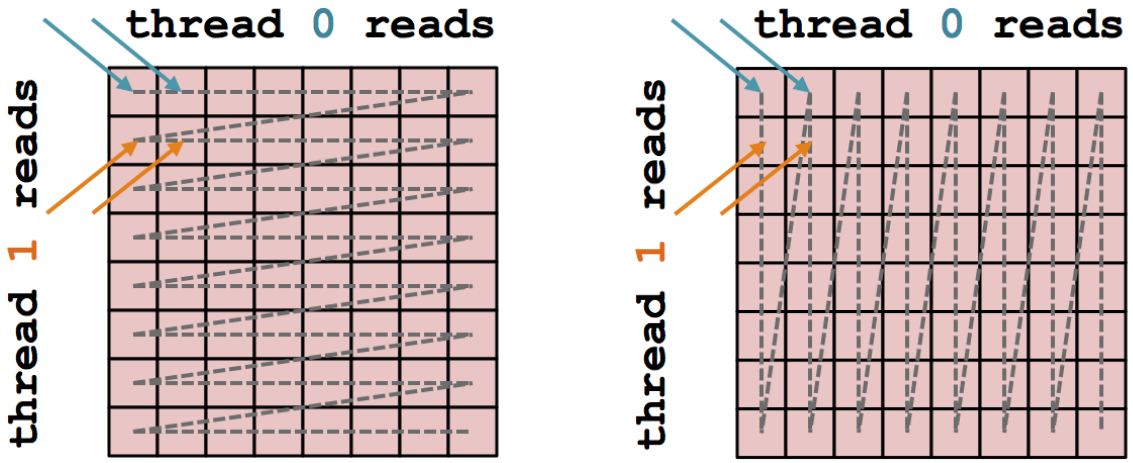
Every `View` has a `Layout` set at compile-time.

```
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`.
- Advanced layouts: `LayoutTiled`, ...more to come

Attempt 2: 2D array, architecture-dependent layout

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



(a) HostSpace (b) CudaSpace

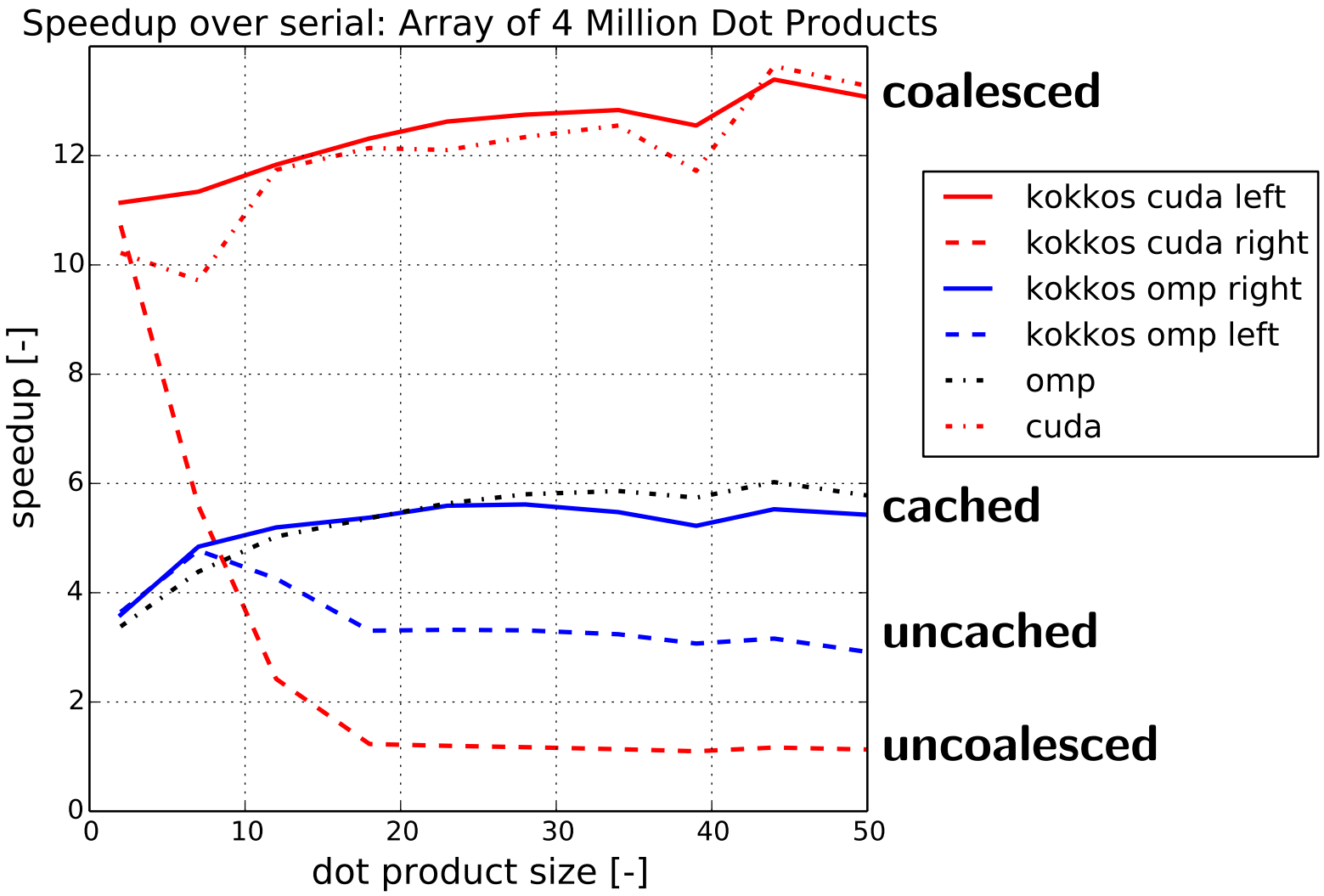
- **HostSpace**: cached (good)
- **CudaSpace**: coalesced (good)

Task: Implement the array of dot products using Kokkos.
do we want them to do this exercise before showing results? it would be only the largest number of dot products we're considering.

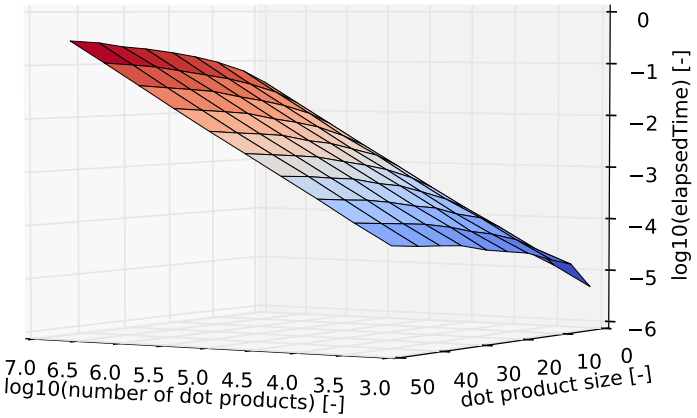
Details:

- All files are contained in `Examples/ArrayOfDotProducts`.
- `ArrayOfDotProducts.cc` contains all testing and timing logic.
- `ArrayOfDotProducts.cc` calls functions defined in `Version_Serial.cc` and `Version_Kokkos.cc`.
- Only `Version_Kokkos.h` needs modification.
- Compile with `make`, run with `./ArrayOfDotProducts`, generate plot with `gnuplot makePlot.gnuplot`.

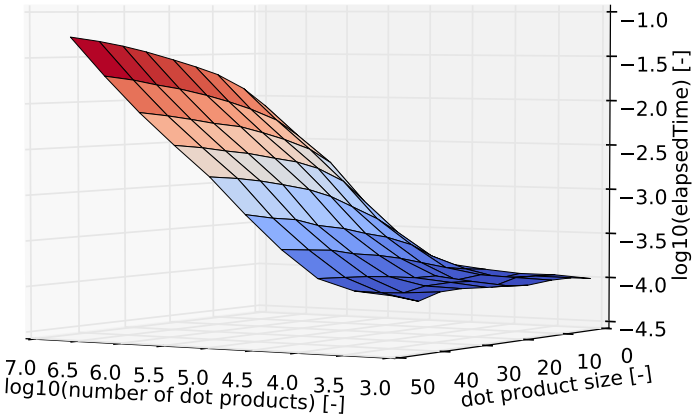
Layout performance



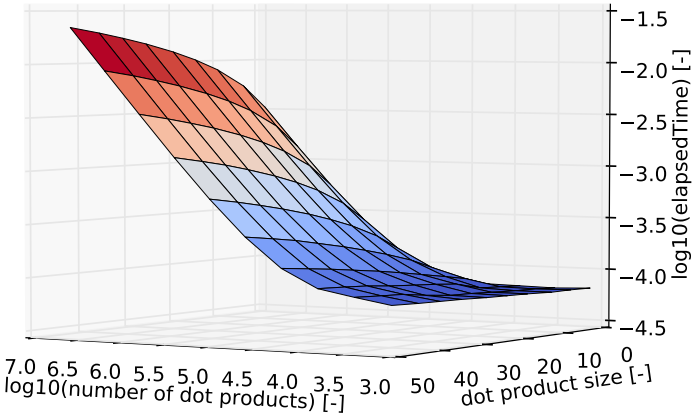
How long of a computation are we considering?



(a) `Serial`



(b) `OpenMP`

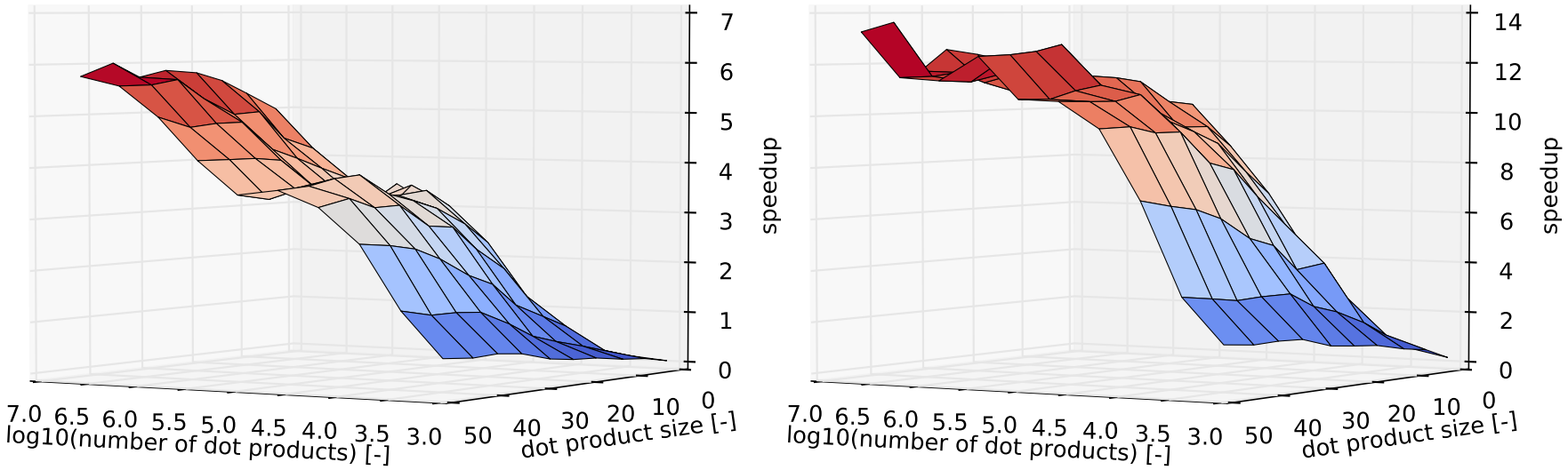


(c) `Cuda`

Note: serial is faster in this corner.

How much do OpenMP and Cuda improve performance?

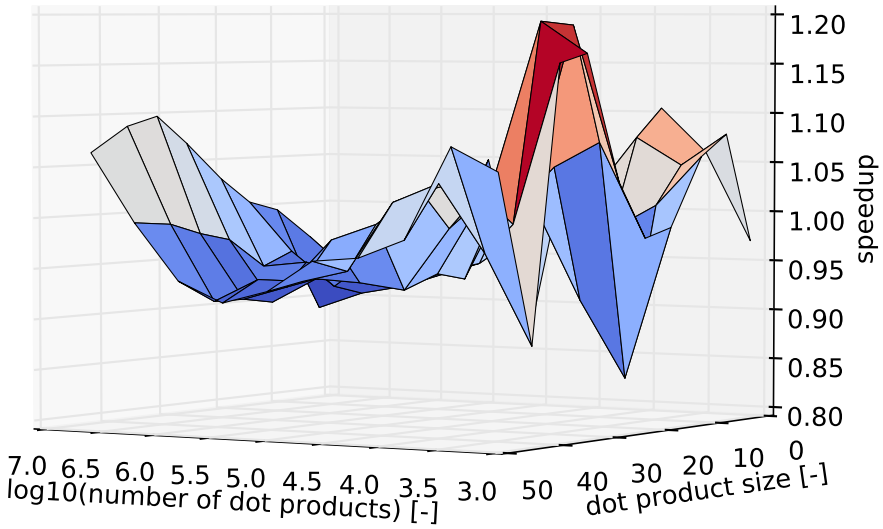
Speedup **over serial**:



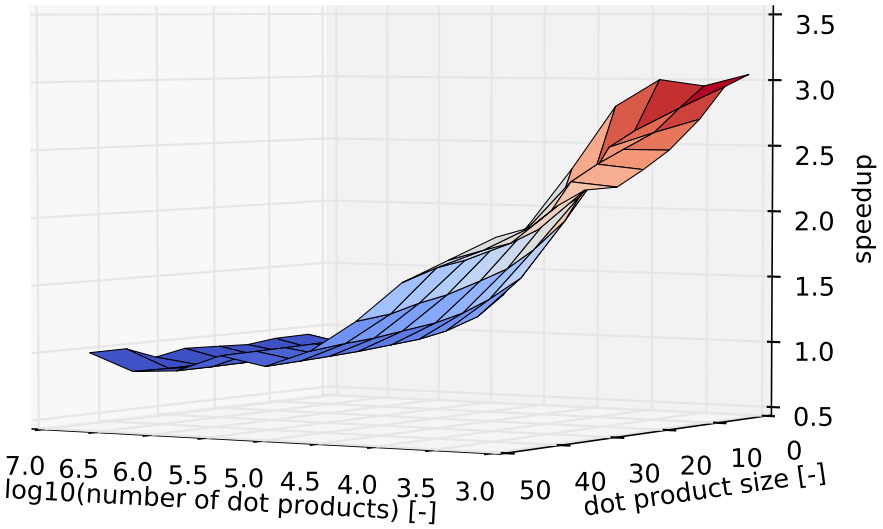
(a) `OpenMP` (b) `Cuda`

How well does Kokkos perform relative to native?

Speedup of **native over Kokkos**:



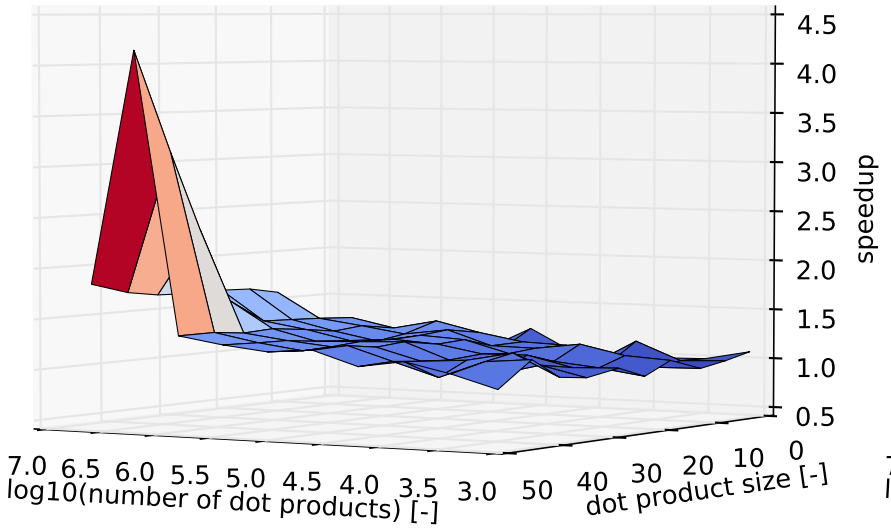
(a) `OpenMP`



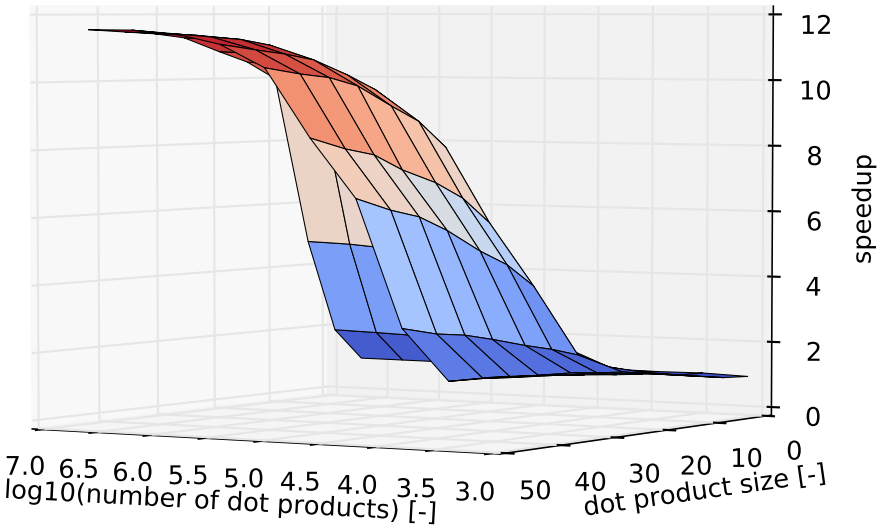
(b) `Cuda`

How much does layout matter?

Speedup of **correct over incorrect layout**:



(a) `OpenMP`



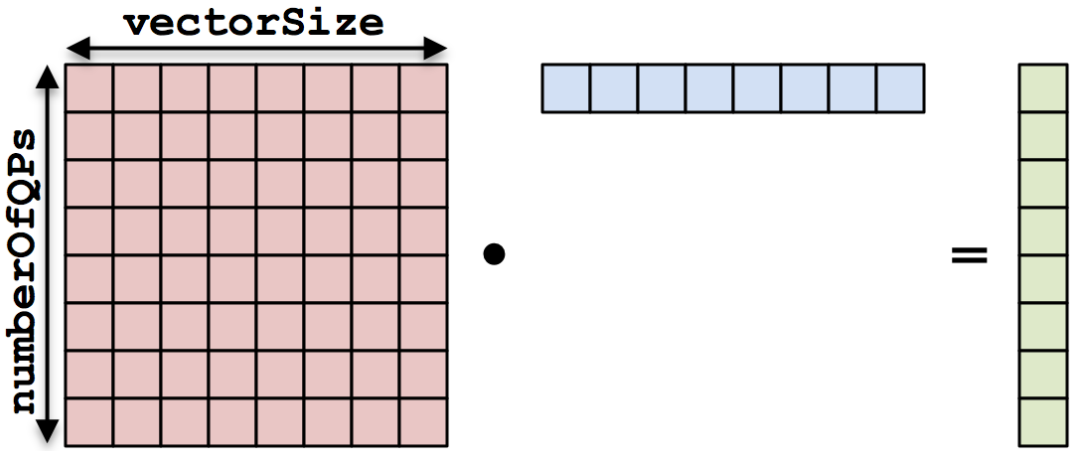
(b) `Cuda`

Example: contractDataFieldScalar

Learning objectives:

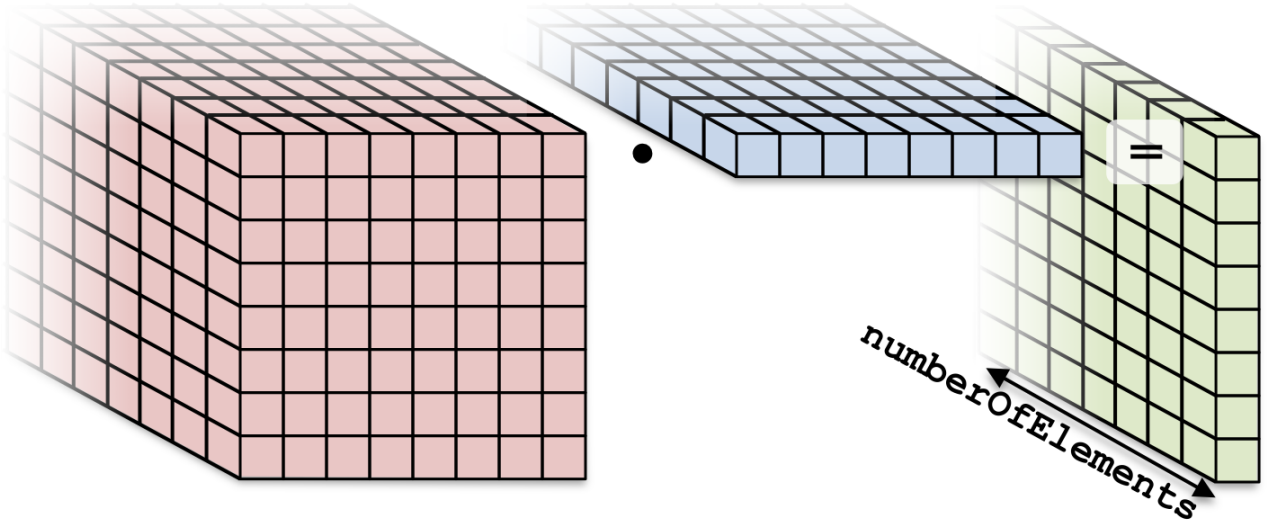
- How to design array shapes such that switching between `Layouts` on different architectures gives optimal performance.
- How to choose between different parallelization approaches.
- Need for and use of multi-dimensional range policies.

One slice of contractDataFieldScalar:




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

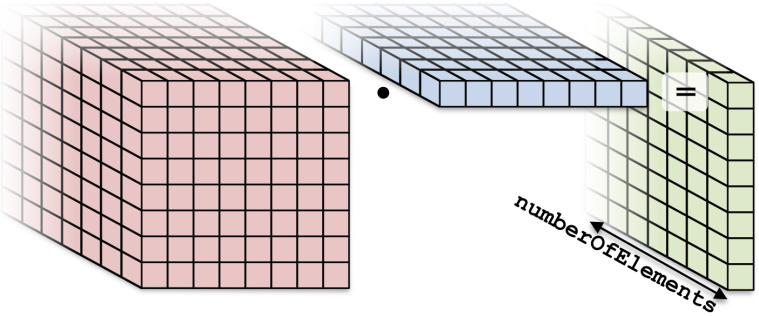
contractDataFieldScalar:




```
for (element = 0; element < numberOfElements; ++element) {
  for (qp = 0; qp < numberOfQPs; ++qp) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  }
}
```

Example: contractDataFieldScalar (3)

```
for (element = 0; element < numberOfElements; ++element) {
  for (qp = 0; qp < numberOfQPs; ++qp) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  }
}
```



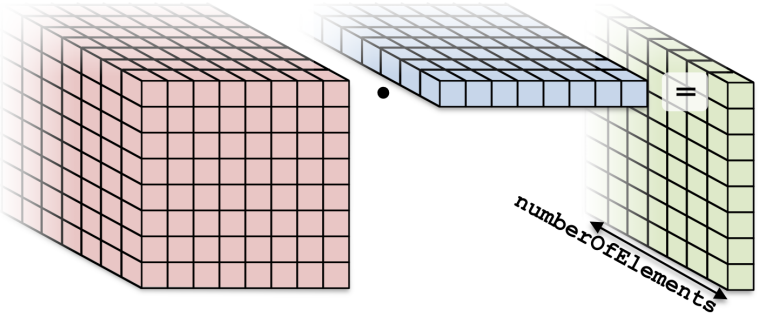
Parallelization approaches:

- Each thread handles an `element`.

  Threads: `numberOfElements`

Example: contractDataFieldScalar (3)

```
for (element = 0; element < numberOfElements; ++element) {
  for (qp = 0; qp < numberOfQPs; ++qp) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  }
}
```

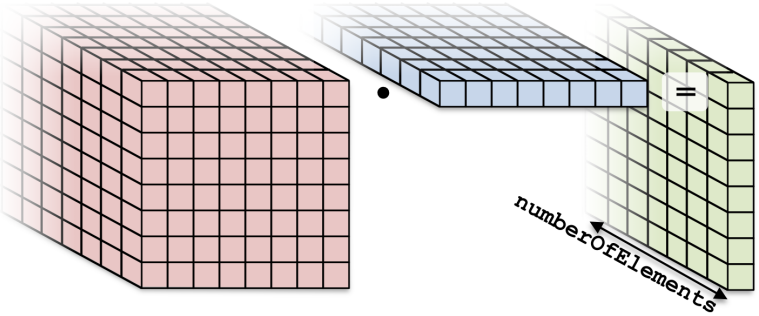


Parallelization approaches:

- Each thread handles an `element`.
  Threads: `numberOfElements`
- Each thread handles a `qp`.
  Threads: `numberOfElements * numberOfQPs`

Example: contractDataFieldScalar (3)

```
for (element = 0; element < numberOfElements; ++element) {
  for (qp = 0; qp < numberOfQPs; ++qp) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  }
}
```

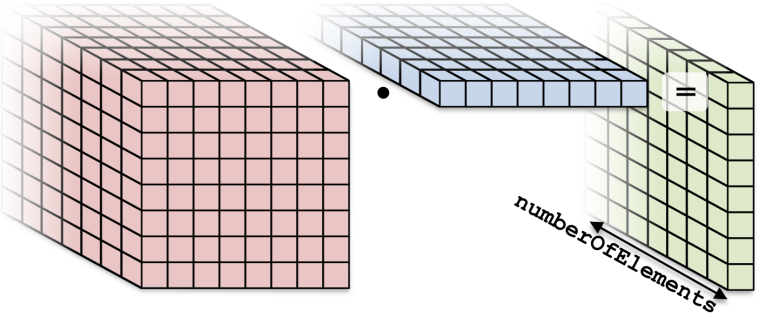


Parallelization approaches:

- Each thread handles an `element`.
  Threads: `numberOfElements`
- Each thread handles a `qp`.
  Threads: `numberOfElements * numberOfQPs`
- Each thread handles an `i`.
  Threads: `numElements * numQPs * vectorSize`
  Requires coordination.

Example: contractDataFieldScalar (3)

```
for (element = 0; element < numberOfElements; ++element) {
  for (qp = 0; qp < numberOfQPs; ++qp) {
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  }
}
```



Parallelization approaches:

- Each thread handles an `element`.
  Threads: `numberOfElements`
- Each thread handles a `qp`.
  Threads: `numberOfElements * numberOfQPs`
- Each thread handles an `i`.
  Threads: `numElements * numQPs * vectorSize`
  Requires coordination.

Example: contractDataFieldScalar (4)

```
parallel_for(numberOfElements * numberOfQPs,
  KOKKOS_LAMBDA (const size_t index) {
    (element, qp) = doThreadingPolicy(index);
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  });
```

Example: contractDataFieldScalar (4)

```
parallel_for(numberOfElements * numberOfQPs,
  KOKKOS_LAMBDA (const size_t index) {
    (element, qp) = doThreadingPolicy(index);
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  });
```

Two design questions to answer:

- **Threading policy**:
  i.e., how map `index` to work (`element, qp`)?

Example: contractDataFieldScalar (4)

```
parallel_for(numberOfElements * numberOfQPs,
  KOKKOS_LAMBDA (const size_t index) {
    (element, qp) = doThreadingPolicy(index);
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  });
```

Two design questions to answer:

- **Threading policy**:
  i.e., how map `index` to work (`element, qp`)?
- `left` **array shape**:
  i.e., `left(element, qp, i)` or
  `left(qp, i, element)` or
  `left(qp, element, i)` or ...

Attempt 0:

- **Threading policy**:

```
element = index / numberOfQPs;
qp      = index % numberOfQPs;
```

- `left` **array shape**:

```
left(element, qp, i)
```

Attempt 0:

- **Threading policy**:

```
element = index / numberOfQPs;
qp      = index % numberOfQPs;
```

- `left` **array shape**:

```
left(element, qp, i)
```

Analysis:

- CPU with `LayoutRight`: cached
- GPU with `LayoutLeft` : uncoalesced

Attempt 1:

- **Threading policy**:

```
element = index / numberOfQPs;
qp      = index % numberOfQPs;
```

- `left` **array shape**:

```
left(qp, element, i)
```

Attempt 1:

- **Threading policy**:

```
element = index / numberOfQPs;
qp      = index % numberOfQPs;
```

- `left` **array shape**:

```
left(qp, element, i)
```

Analysis:

- CPU with `LayoutRight`: (mostly) cached
- GPU with `LayoutLeft` : coalesced

Attempt 1:

- **Threading policy**:

```
element = index / numberOfQPs;
qp      = index % numberOfQPs;
```

- `left` **array shape**:

```
left(qp, element, i)
```

Analysis:

- CPU with `LayoutRight`: (mostly) cached
- GPU with `LayoutLeft` : coalesced

Solution? **Multi-dimensional range policies** with **layouts**.

Multi-dimensional range policies

```
parallel_for(
  RangePolicy<Layout, Space, 2>(numElements, numQPs),
  KOKKOS_LAMBDA (const array<size_t, 2> indices) {
    element = indices[0];
    qp      = indices[1];
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  });
```

Multi-dimensional range policies

```
parallel_for(
  RangePolicy<Layout, Space, 2>(numElements, numQPs),
  KOKKOS_LAMBDA (const array<size_t, 2> indices) {
    element = indices[0];
    qp      = indices[1];
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  });
```

LayoutRight:

```
element = index / numberOfQPs;
qp      = index % numberOfQPs;
```

LayoutLeft:

```
element = index % numberOfElements;
qp      = index / numberOfElements;
```

Attempt 2:

```
View<double***, Layout, Space> left;
View<double**,  Layout, Space> right;
View<double**,  Layout, Space> result;
parallel_for(
  RangePolicy< Layout, Space, 2>(numElements, numQPs),
  KOKKOS_LAMBDA (const array<size_t, 2> indices) {
    element = indices[0];
    qp      = indices[1];
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  });
```

Attempt 2:

```
View<double***, Layout, Space> left;
View<double**,  Layout, Space> right;
View<double**,  Layout, Space> result;
parallel_for(
  RangePolicy< Layout, Space, 2>(numElements, numQPs),
  KOKKOS_LAMBDA (const array<size_t, 2> indices) {
    element = indices[0];
    qp      = indices[1];
    total = 0;
    for (i = 0; i < vectorSize; ++i) {
      total += left(element, qp, i) * right(element, i);
    }
    result(element, qp) = total;
  });
```

Analysis:

- CPU with LayoutRight: cached
- GPU with LayoutLeft : coalesced

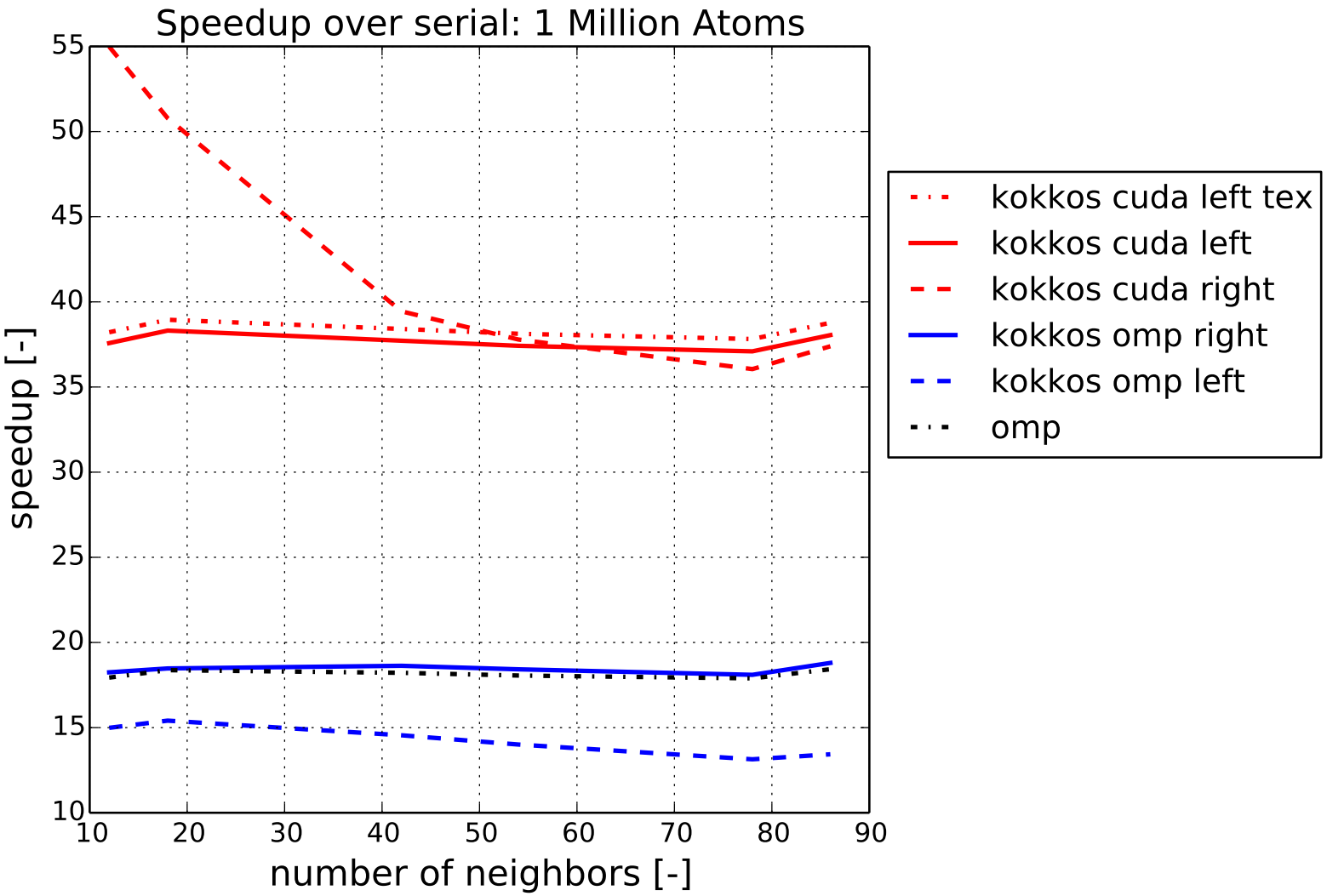
textures are neat

Task: Implement a Lennard-Jones MD force kernel using Kokkos.

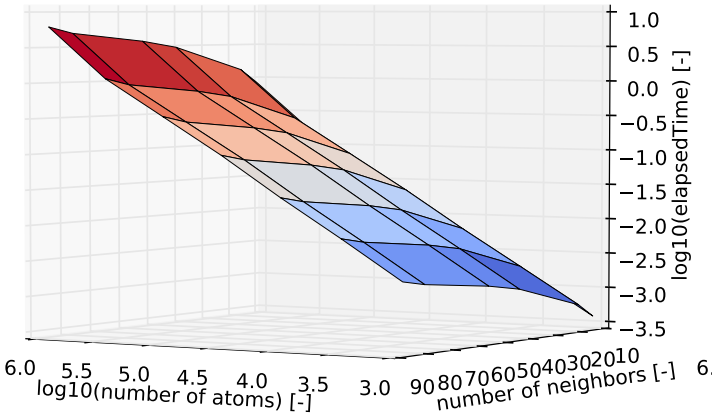
Details:

- All files are contained in `Examples/MDForceKernel`.
- `MDForceKernel.cc` contains all testing and timing logic.
- `MDForceKernel.cc` calls functions defined in `Version_Serial.cc` and `Version_Kokkos.cc`.
- Only `Version_Kokkos.h` needs modification.
- Compile with `make`, run with `./MDForceKernel`, generate plot with `gnuplot makePlot.gnuplot`.

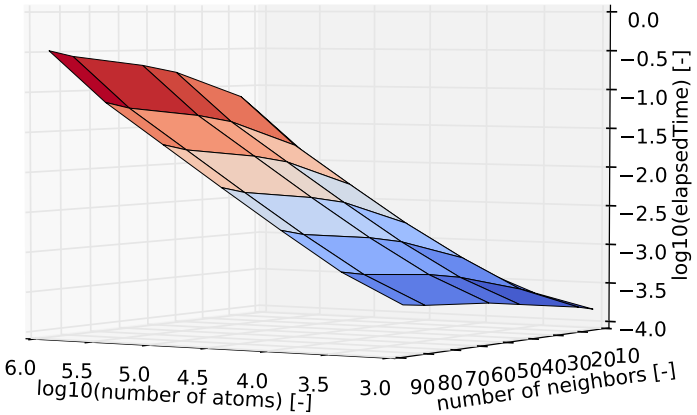
Results:


Speedup over serial: 1 Million Atoms
speedup [-]
number of neighbors [-]
55
50
45
40
35
30
25
20
15
10
10
20
30
40
50
60
70
80
90
kokkos cuda left tex
kokkos cuda left
kokkos cuda right
kokkos omp right
kokkos omp left
omp

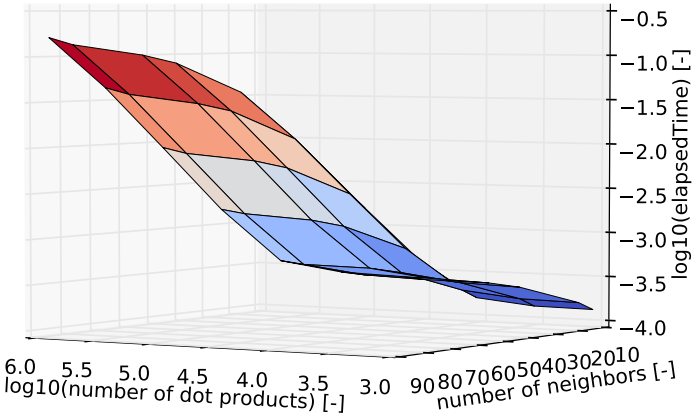
How long of a computation are we considering?



(a) `Serial`



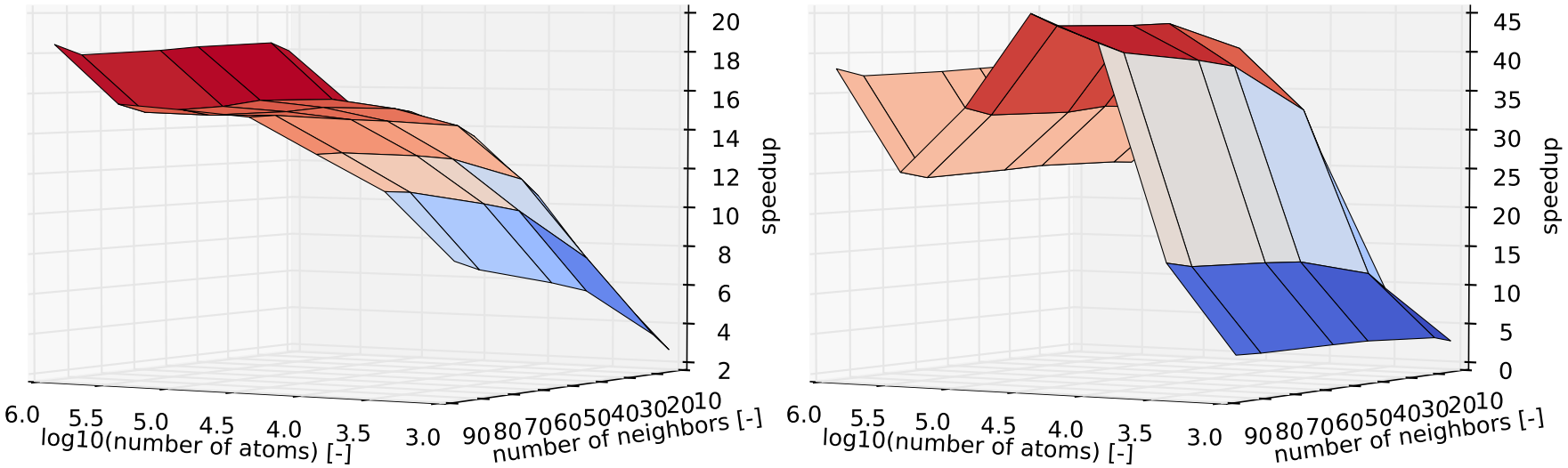
(b) `OpenMP`



(c) `Cuda`

How much do OpenMP and Cuda improve performance?

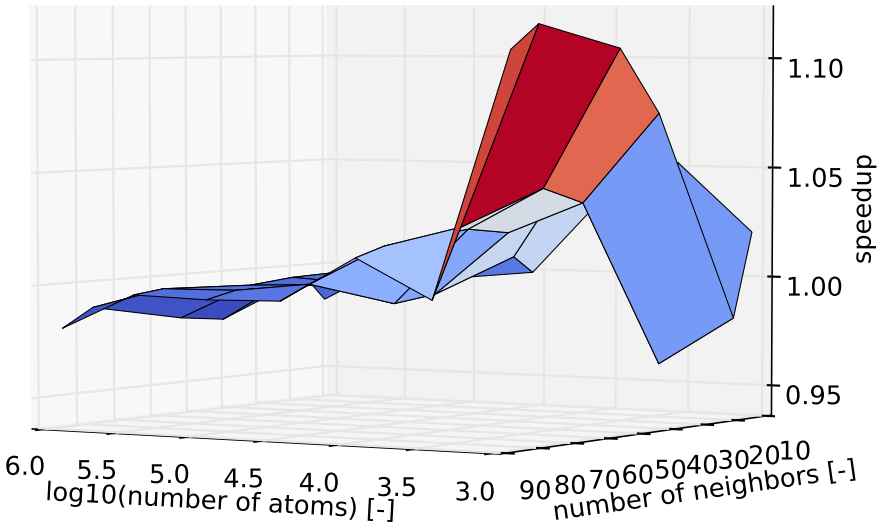
Speedup **over serial**:



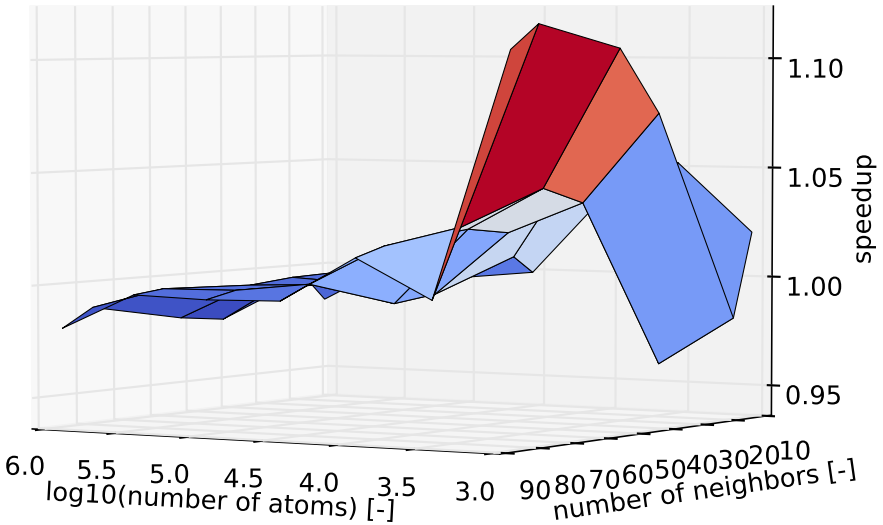
(a) `OpenMP` (b) `Cuda`

How well does Kokkos perform relative to native?

Speedup of **native over Kokkos**:



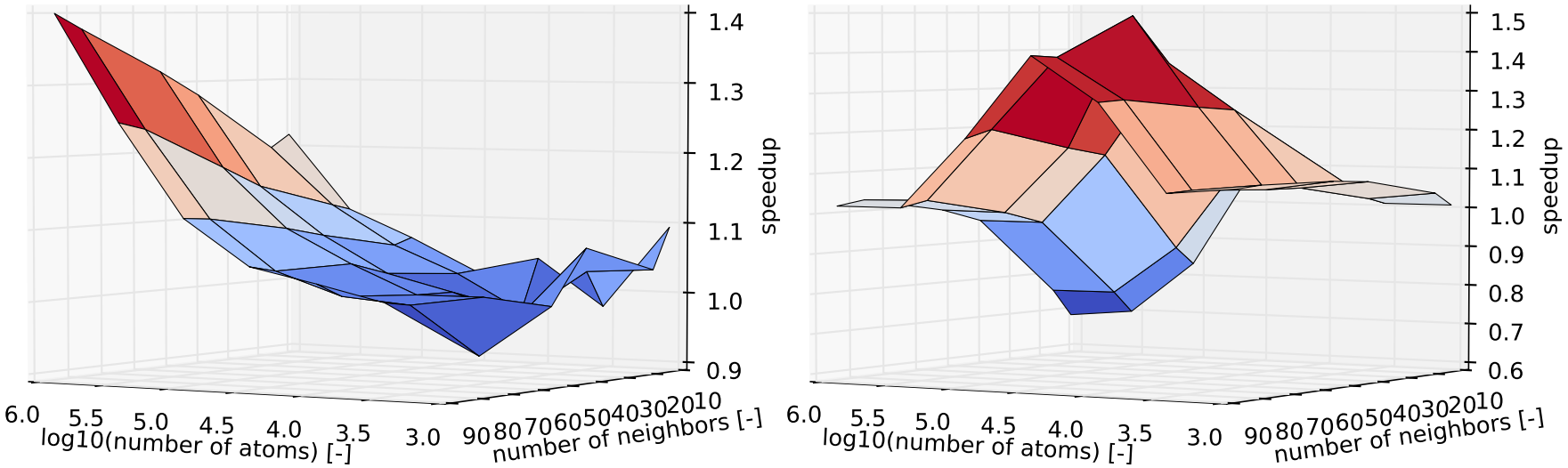
(a) `OpenMP`



(b) `no vanilla cuda yet`

How much does layout matter?

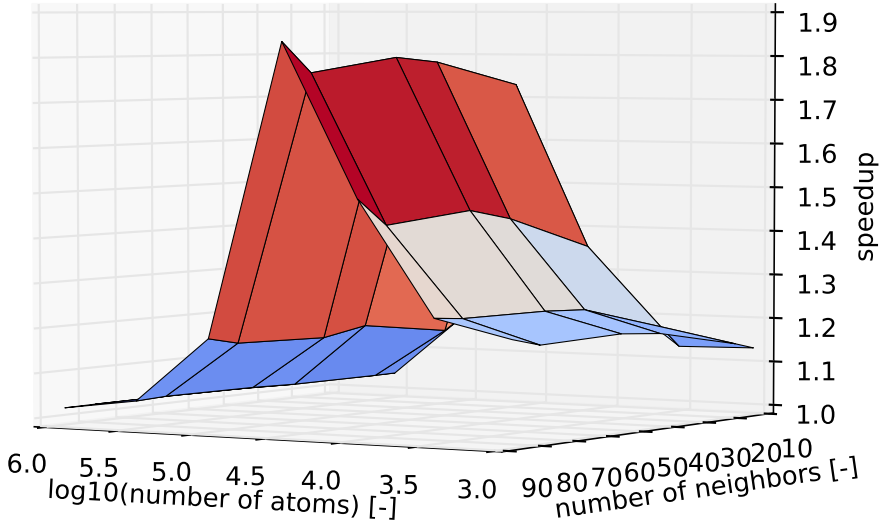
Speedup of **correct over incorrect layout**:



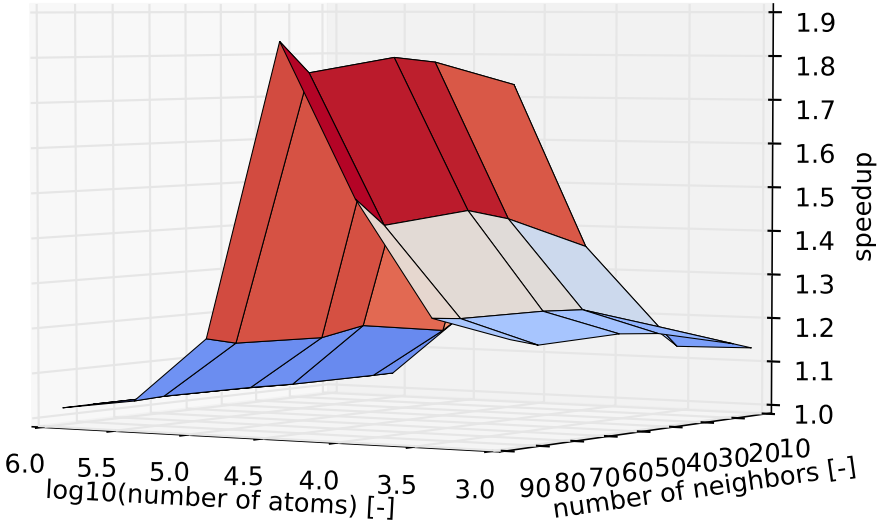
(a) `OpenMP` (b) `Cuda`

How much does RandomAccess matter?

Speedup of **RandomAccess over correct layout**:



(a) no openmp yet, though it'll do nothing



(b) Cuda

Thought experiment: `saxpy` with `std::vector`

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

What happens?