The Kokkos Lectures

Module 3: MultiDimensional Loops and Data Structures

July 31, 2020

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.
SAND2020-7908 PE

Online Resources:

- https://github.com/kokkos:
  - Primary Kokkos GitHub Organization
- https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series:
  - Slides, recording and Q&A for the Lectures
- https://github.com/kokkos/kokkos/wiki:
  - Wiki including API reference
- https://kokkosteam.slack.com:
  - Slack channel for Kokkos.
  - Please join: fastest way to get your questions answered.
  - Can whitelist domains, or invite individual people.

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

Kokkos EcoSystem
Building Kokkos
Data Parallelism:

- Simple parallel loops use the `parallel_for` pattern:

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

- Reductions combine contributions from loop iterations

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

Recording: https://bit.ly/kokkos-lecture-series-1

Kokkos View

- Multi Dimensional Array.
- Compile and Runtime Dimensions.
- Reference counted like a `std::shared_ptr` to an array.

```
Kokkos::View<int*[5]> a("A", N);
a(3,2) = 7;
```

Execution Spaces

- Parallel operations execute in a specified **Execution Space**
- Can be controlled via template argument to **Execution Policy**
- If no Execution Space is provided use `DefaultExecutionSpace`

```
// Equivalent:
parallel_for("L", N, functor);
parallel_for("L",
  RangePolicy<DefaultExecutionSpace>(0, N), functor);
```

Memory Spaces

- Kokkos Views store data in **Memory Spaces**.
- Provided as template parameter.
- If no Memory Space is given, use `Kokkos::DefaultExecutionSpace::memory_space`.
- deep_copy is used to transfer data: no hidden memory copies by Kokkos.

```
View<int*, CudaSpace> a("A", M);
// View in host memory to load from file
auto h_a = create_mirror_view(a);
load_from_file(h_a);
// Copy
deep_copy(a,h_a);
```

Layouts

- Kokkos Views use an index mapping to memory determined by a **Layout**.
- Provided as template parameter.
- If no **Layout** is given, derived from the execution space associated with the memory space.
- Defaults are good if you parallelize over left most index!

```
View<int**, LayoutLeft> a("A", N, M);
View<int**, LayoutRight> b("B", N, M);

parallel_for("Fill", N, KOKKOS_LAMBDA(int i) {
  for(int j = 0; j < M; j++) {
    a(i,j) = i * 1000 + j; // coalesced
    b(i,j) = i * 1000 + j; // cached
  }
});
```

Advanced Reductions

- `parallel_reduce` defaults to summation
- Kokkos reducers can be used to reduce over arbitrary operations
- Reductions over multiple values are supported
- Only reductions into scalar arguments are guaranteed to be synchronous

```
parallel_reduce("Join", n,
  KOKKOS_LAMBDA(int i, double& a, int& b) {
    int idx = foo();
    if(idx > b) b = idx;
    a += bar();
  }, result, Kokkos::Max<int>{my_max});
```

MultiDimensional Loops

How to parallelize tightly nested loops using the MDRangePolicy?

Subviews and Unmanaged Views

How to get slices of Views, View assignment rules and interoperating with external memory.

Atomic Data Access

Using atomic functions. Implement an optimal scatter contribute pattern.

DualView

Managing data synchronization without global understanding of data flow.

Tightly Nested Loops with MDRangePolicy

Learning objectives:

- Demonstrate usage of the MDRangePolicy with tightly nested loops.
- Syntax - Required and optional settings
- Code demo and example

Motivating example: Consider the nested for loops:

```
for ( int i = 0; i < N0; ++i )
for ( int j = 0; j < N1; ++j )
for ( int k = 0; k < N2; ++k )
  some_init_fcn(i, j, k);
```

Based on Kokkos lessons thus far, you might parallelize this as

```
Kokkos::parallel_for("Label", N0,
                     KOKKOS_LAMBDA (const i) {
                       for ( int j = 0; j < N1; ++j )
                       for ( int k = 0; k < N2; ++k )
                        some_init_fcn(i, j, k);
                     }
                     );
```

- This only parallelizes along one dimension, leaving potential parallelism unexploited.
- What if Ni is too small to amortize the cost of constructing a parallel region, but Ni*Nj*Nk makes it worthwhile?

OpenMP has a solution: the collapse clause

```
#pragma omp parallel for collapse(3)
for (int64_t i = 0; i < N0; ++i) {
  for (int64_t j = 0; j < N1; ++j) {
    for (int64_t k = 0; k < N2; ++k) {
      /* loop body */
    }
  }
}
```

OpenMP has a solution: the collapse clause

```
#pragma omp parallel for collapse(3)
for (int64_t i = 0; i < N0; ++i) {
  for (int64_t j = 0; j < N1; ++j) {
    for (int64_t k = 0; k < N2; ++k) {
      /* loop body */
    }
  }
}
```

Note this changed the policy by adding a ‘collapse‘ clause.

OpenMP has a solution: the collapse clause

```
#pragma omp parallel for collapse(3)
for (int64_t i = 0; i < N0; ++i) {
  for (int64_t j = 0; j < N1; ++j) {
    for (int64_t k = 0; k < N2; ++k) {
      /* loop body */
    }
  }
}
```

Note this changed the policy by adding a ‘collapse‘ clause.

With Kokkos you also change the policy:

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
     /* loop body */
});
```

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.
- As with Kokkos Views: only rectangular iteration spaces.

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.
- As with Kokkos Views: only rectangular iteration spaces.
- Provide initializer lists for begin and end values.

MDRangePolicy

MDRangePolicy can parallelize tightly nested loops of 1 to 6 dimensions.

```
parallel_for("L", MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
   KOKKOS_LAMBDA(int64_t i, int64_t j, int64_t k) {
      /* loop body */
});
```

- Specify the dimensionality of the loop with $Rank < DIM >$.
- As with Kokkos Views: only rectangular iteration spaces.
- Provide initializer lists for begin and end values.
- The functor/lambda takes matching number of indicies.

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
      /* loop body */
  lsum += something;
}, result);
```

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.
- Can do other reductions with reducers.

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.
- Can do other reductions with reducers.
- Can use ‘View‘s as reduction argument.

You can also do Reductions:

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
      /* loop body */
  lsum += something;
}, result);
```

- The Policy doesn't change the rules for ‘parallel_reduce‘.
- Additional Thread Local Argument.
- Can do other reductions with reducers.
- Can use ‘View‘s as reduction argument.
- Multiple reducers not yet implemented though.

In structured grid applications a **tiling** strategy is often used to help with caching.

Tiling

MDRangePolicy uses a tiling strategy for the iteration space.

- Specified as a third initializer list.
- For GPUs a tile is handled by a single thread block.
  - If you provide too large a tile size this will fail!
- In Kokkos 3.3 we will add auto tuning for tile sizes.

```
double result;
parallel_reduce("Label",
  MDRangePolicy<Rank<3>>({0,0,0},{N0,N1,N2},{T0,T1,T2}),
  KOKKOS_LAMBDA(int i, int j, int k, double& lsum) {
     /* loop body */
  lsum += something;
}, result);
```

Initializing a Matrix:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

Initializing a Matrix:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2>>({0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

How do I make sure that I get the right access pattern?

Iteration Pattern

MDRangePolicy provides compile time control over iteration patterns.

```
Kokkos::Rank< N, IterateOuter, IterateInner >
```

- **N: (Required)** the rank of the index space (limited from 2 to 6)
- **IterateOuter (Optional)** iteration pattern between tiles
  - **Options:** Iterate::Left, Iterate::Right, Iterate::Default
- **IterateInner (Optional)** iteration pattern within tiles
  - **Options:** Iterate::Left, Iterate::Right, Iterate::Default

Initializing a Matrix fast:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Left,Iterate::Left>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Right,Iterate::Right>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

Initializing a Matrix fast:

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Left,Iterate::Left>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});

View<double**,LayoutRight> B("B",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Right,Iterate::Right>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = 1000.0 * i + 1.0*j;
});
```

Default Patterns Match

Default iteration patterns match the default memory layouts!

Exercise - mdrange: Initialize multi-dim views with MDRangePolicy

Details:

- Location: `Exercises/mdrange/Begin/`
- This begins with the `Solution` of 02
- Initialize the device Views x and y directly on the device using a parallel for and RangePolicy
- Initialize the device View matrix A directly on the device using a parallel for and MDRangePolicy

```
# Compile for CPU
make -j KOKKOS_DEVICES=OpenMP
# Compile for GPU (we do not need UVM anymore)
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./mdrange_exercise.cuda -S 26
```

Template Parameters common to ALL policies.

- ExecutionSpace: control where code executes
  - **Options:** Serial, OpenMP, Threads, Cuda, HIP, ...
- Schedule<Options>: set scheduling policy.
  - **Options:** Static, Dynamic
- IndexType<Options>: control internal indexing type
  - **Options:** int, long, etc
- WorkTag: enables multiple operators in one functor

```
struct Foo {
  struct Tag1{}; struct Tag2{};
  KOKKOS_FUNCTION void operator(Tag1, int i) const {...}
  KOKKOS_FUNCTION void operator(Tag2, int i) const {...}
  void run_both(int N) {
    parallel_for(RangePolicy<Tag1>(0,N),*this);
    parallel_for(RangePolicy<Tag2>(0,N),*this);
  }
});
```

MDRangePolicy

- allows for tightly nested loops similar to OpenMP's collapse clause.
- requires functors/lambdas with as many parameters as its rank is.
- works with `parallel_for` and `parallel_reduce`.
- uses a tiling strategy for the iteration space.
- provides compile time control over iteration patterns.

Subviews: Taking slices of Views

Learning objectives:

- Introduce Kokkos::subview—basic capabilities and syntax
- Suggested usage and practices
- View assignment rules

Sometimes you have to call functions on a subset of data:

Sometimes you have to call functions on a subset of data:

Example: call a frobenius norm on a matrix slice of a rank-3 tensor:

```
double special_norm(View<double***> tensor, int i) {

  auto matrix = ???;
  // Call a function that takes a matrix:
  return some_library::frobenius_norm(matrix);
}
```

Sometimes you have to call functions on a subset of data:

Example: call a frobenius norm on a matrix slice of a rank-3 tensor:

```
double special_norm(View<double***> tensor, int i) {

  auto matrix = ???;
  // Call a function that takes a matrix:
  return some_library::frobenius_norm(matrix);
}
```

In Fortran or Matlab or Python you can get such a slice:

```
    tensor(i,:,:)
```

Sometimes you have to call functions on a subset of data:

Example: call a frobenius norm on a matrix slice of a rank-3 tensor:

```
double special_norm(View<double***> tensor, int i) {

  auto matrix = ???;
  // Call a function that takes a matrix:
  return some_library::frobenius_norm(matrix);
}
```

In Fortran or Matlab or Python you can get such a slice:

```
    tensor(i,:,:)
```

Kokkos can do that too!

Subview

`Kokkos::subview` can be used to get a view to a subset of an existing `View`.

Subview description:

- A subview is a “slice” of a View

Subview description:

- A subview is a "slice" of a View
  - The function template `Kokkos::subview()` takes a `View` and a slice for each dimension and returns a `View` of the appropriate shape.

Subview description:

- A subview is a "slice" of a View
  - The function template `Kokkos::subview()` takes a `View` and a slice for each dimension and returns a `View` of the appropriate shape.
  - The subview and original View point to the same data—no extra memory allocation nor copying

Subview description:

- A subview is a “slice” of a View
  - The function template `Kokkos::subview()` takes a `View` and a slice for each dimension and returns a `View` of the appropriate shape.
  - The subview and original View point to the same data—no extra memory allocation nor copying
- Can be constructed on host or within a kernel, since no allocation of memory occurs

Subview description:

- A subview is a "slice" of a View
  - The function template `Kokkos::subview()` takes a `View` and a slice for each dimension and returns a `View` of the appropriate shape.
  - The subview and original View point to the same data—no extra memory allocation nor copying
- Can be constructed on host or within a kernel, since no allocation of memory occurs
- Similar capability as provided by Matlab, Fortran, Python, etc., using "colon" notation

Introductory Usage Demo:

Given a `View`:

```
Kokkos::View<double***> v("v", N0, N1, N2);
```

Introductory Usage Demo:

Given a `View`:

```
Kokkos::View<double***> v("v", N0, N1, N2);
```

Say we want a 2-dimensional slice at an index `i0` in the first dimension—that is, in Matlab/Fortran/Python notation:

```
slicei0 = v(i0, :, :);
```

Introductory Usage Demo:

Given a `View`:

```
Kokkos::View<double***> v("v", N0, N1, N2);
```

Say we want a 2-dimensional slice at an index `i0` in the first dimension—that is, in Matlab/Fortran/Python notation:

```
slicei0 = v(i0, :, :);
```

This can be accomplished in Kokkos using a subview as follows:

```
auto sv_i0 =
  Kokkos::subview(v, i0, Kokkos::ALL, Kokkos::ALL);

// Just like in Python, you can do the same thing with
// the equivalent of v(i0, 0:N1, 0:N2)
auto sv_i0_other =
  Kokkos::subview(v, i0, Kokkos::make_pair(0, N1),
                         Kokkos::make_pair(0, N2));
```

Subview can take three types of slice arguments:

- **Index**
  - For every index i the rank of the resulting `View` is decreased by one.
  - Must be between $0 <= i < extent(dim)$
- **Kokkos::pair**
  - References a half-open range of indices.
  - The begin and end must be within the extents of the original view.
- **Kokkos::ALL**
  - References the entire extent in that dimension.
  - Equivalent to providing `make_pair(0,v.extent(dim))`

Usage notes:

- Use `auto` for the type of a subview (unless you can't)

Usage notes:

- Use `auto` for the type of a subview (unless you can't)

Usage notes:

- Use `auto` for the type of a subview (unless you can't)
  - The return type of `Kokkos::subview()` is implementation defined for performance reasons

Usage notes:

- Use `auto` for the type of a subview (unless you can't)
    - The return type of `Kokkos::subview()` is implementation defined for performance reasons
    - You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere

Usage notes:

- Use `auto` for the type of a subview (unless you can't)
  - The return type of `Kokkos::subview()` is implementation defined for performance reasons
  - You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- Use `Kokkos::pair` for partial ranges if subview created within a kernel

Usage notes:

- Use `auto` for the type of a subview (unless you can't)
  - The return type of `Kokkos::subview()` is implementation defined for performance reasons
  - You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- Use `Kokkos::pair` for partial ranges if subview created within a kernel
- Constructing subviews in inner loop code can have performance implications (for now. . .)

Usage notes:

- Use `auto` for the type of a subview (unless you can't)
  - The return type of `Kokkos::subview()` is implementation defined for performance reasons
  - You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- Use `Kokkos::pair` for partial ranges if subview created within a kernel
- Constructing subviews in inner loop code can have performance implications (for now...)
  - This will likely be far less of an issue in the future.

Usage notes:

- Use `auto` for the type of a subview (unless you can't)
  - The return type of `Kokkos::subview()` is implementation defined for performance reasons
  - You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- Use `Kokkos::pair` for partial ranges if subview created within a kernel
- Constructing subviews in inner loop code can have performance implications (for now. . .)
  - This will likely be far less of an issue in the future.
  - Prioritize readability and maintainability first, then make changes only if you see a performance impact.

Usage notes:

- Use `auto` for the type of a subview (unless you can't)
  - The return type of `Kokkos::subview()` is implementation defined for performance reasons
  - You can also use `decltype(subview(/*...*/))` if you really need to spell name of the type somewhere
- Use `Kokkos::pair` for partial ranges if subview created within a kernel
- Constructing subviews in inner loop code can have performance implications (for now...)
  - This will likely be far less of an issue in the future.
  - Prioritize readability and maintainability first, then make changes only if you see a performance impact.

Details:

- Location: `Exercises/subview/Begin/`
- This begins with the `Solution` of 04
- In the parallel reduce kernel, create a subview for row j of view A
- Use this subview when computing A(j,:)*x(:) rather than the matrix A

```
# Compile for CPU
make -j KOKKOS_DEVICES=OpenMP
# Compile for GPU (we do not need UVM anymore)
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./subview_exercise.cuda -S 26
```

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)`

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay

`View::operator=()` just does the "Right Thing"™

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)` => Okay, checked at runtime

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`
  => Okay, checked at runtime
- `View<int*[5]> a; a = View<int*[3]>("b", 4)`

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`
  => Okay, checked at runtime
- `View<int*[5]> a; a = View<int*[3]>("b", 4)`
  => Compilation error

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`
  => Okay, checked at runtime
- `View<int*[5]> a; a = View<int*[3]>("b", 4)`
  => Compilation error
- `View<int*[5]> a; a = View<int**>("b", 4, 3)`

Aside: View Assignment (1)

`View::operator=()` just does the "Right Thing"™

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`
  => Okay, checked at runtime
- `View<int*[5]> a; a = View<int*[3]>("b", 4)`
  => Compilation error
- `View<int*[5]> a; a = View<int**>("b", 4, 3)`
  => Runtime error

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`
  => Okay, checked at runtime
- `View<int*[5]> a; a = View<int*[3]>("b", 4)`
  => Compilation error
- `View<int*[5]> a; a = View<int**>("b", 4, 3)`
  => Runtime error
- `View<int*, CudaSpace> a;`
  `a = View<int*, HostSpace>("b", 4)`

Aside: View Assignment (1)

`View::operator=()` **just does the “Right Thing”™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`
  => Okay, checked at runtime
- `View<int*[5]> a; a = View<int*[3]>("b", 4)`
  => Compilation error
- `View<int*[5]> a; a = View<int**>("b", 4, 3)`
  => Runtime error
- `View<int*, CudaSpace> a;`
  `a = View<int*, HostSpace>("b", 4)`
  => Compilation error
- `View<int**, LayoutLeft> a;`
  `a = View<int**, LayoutRight>("b", 4, 5)`

`View::operator=()` **just does the "Right Thing"™**

- `View<int**> a; a = View<int*[5]>("b", 4)` => Okay
- `View<int*[5]> a; a = View<int**>("b", 4, 5)`
  => Okay, checked at runtime
- `View<int*[5]> a; a = View<int*[3]>("b", 4)`
  => Compilation error
- `View<int*[5]> a; a = View<int**>("b", 4, 3)`
  => Runtime error
- `View<int*, CudaSpace> a;`
  `a = View<int*, HostSpace>("b", 4)`
  => Compilation error
- `View<int**, LayoutLeft> a;`
  `a = View<int**, LayoutRight>("b", 4, 5)`
  => Compilation error

`View::operator=()` **just does the "Right Thing"™**

- `View<const int*> a; a = View<int*>("b", 4)`

`View::operator=()` **just does the "Right Thing"**™

- `View<const int*> a; a = View<int*>("b", 4)`
  => Okay

`View::operator=()` **just does the "Right Thing"™**

- `View<const int*> a; a = View<int*>("b", 4)`
  => Okay
- `View<int*> a; a = View<const int*>("b", 4)`

`View::operator=()` **just does the "Right Thing"™**

- `View<const int*> a; a = View<int*>("b", 4)`
  => Okay
- `View<int*> a; a = View<const int*>("b", 4)`
  => Compilation error

`View::operator=()` **just does the "Right Thing"™**

- `View<const int*> a; a = View<int*>("b", 4)`
  => Okay
- `View<int*> a; a = View<const int*>("b", 4)`
  => Compilation error
- `View<int*[5], LayoutStride> a;`
  `a = View<int*[5], LayoutLeft>("b", 4)`

`View::operator=()` **just does the "Right Thing"™**

- `View<const int*> a; a = View<int*>("b", 4)`
  => Okay
- `View<int*> a; a = View<const int*>("b", 4)`
  => Compilation error
- `View<int*[5], LayoutStride> a;`
  `a = View<int*[5], LayoutLeft>("b", 4)` => Okay, converting compile-time strides into runtime strides

`View::operator=()` **just does the "Right Thing"™**

- `View<const int*> a; a = View<int*>("b", 4)`
  => Okay
- `View<int*> a; a = View<const int*>("b", 4)`
  => Compilation error
- `View<int*[5], LayoutStride> a;`
  `a = View<int*[5], LayoutLeft>("b", 4)` => Okay, converting compile-time strides into runtime strides
- `View<int*[5], LayoutLeft> a;`
  `a = View<int*[5], LayoutStride>("b", 4)`

`View::operator=()` **just does the “Right Thing”^{TM}**

- `View<const int*> a; a = View<int*>("b", 4)`
  => Okay
- `View<int*> a; a = View<const int*>("b", 4)`
  => Compilation error
- `View<int*[5], LayoutStride> a;`
  `a = View<int*[5], LayoutLeft>("b", 4)` => Okay, converting compile-time strides into runtime strides
- `View<int*[5], LayoutLeft> a;`
  `a = View<int*[5], LayoutStride>("b", 4)` => Okay, but only if strides match layout left (checked at runtime)

Given a `View`:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ```
  View<int***> a;
  a = Kokkos::subview(v, ALL, 42, ALL);
  ```
  ```

Given a `View`:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ```
  View<int***> a;
  a = Kokkos::subview(v, ALL, 42, ALL);
  ```
  => Compilation error
  ```

Given a `View`:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- `View<int***> a;`
  `a = Kokkos::subview(v, ALL, 42, ALL);`
  => Compilation error
- `View<int*> a;`
  `a = Kokkos::subview(v, ALL, 5, 42);`

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- `View<int***> a;`
  `a = Kokkos::subview(v, ALL, 42, ALL);`
  => Compilation error
- `View<int*> a;`
  `a = Kokkos::subview(v, ALL, 5, 42);`
  => Okay for LayoutLeft but => Compilation error for LayoutRight

Given a `View`:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- `View<int***> a;`
  `a = Kokkos::subview(v, ALL, 42, ALL);`
  => Compilation error
- `View<int*> a;`
  `a = Kokkos::subview(v, ALL, 5, 42);`
  => Okay for LayoutLeft but => Compilation error for LayoutRight
- `View<int**> a;`
  `a = Kokkos::subview(v, ALL, 15, ALL);`

Given a `View`:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- `View<int***> a;`
  `a = Kokkos::subview(v, ALL, 42, ALL);`
  => Compilation error
- `View<int*> a;`
  `a = Kokkos::subview(v, ALL, 5, 42);`
  => Okay for LayoutLeft but => Compilation error for LayoutRight
- `View<int**> a;`
  `a = Kokkos::subview(v, ALL, 15, ALL);`
  => Runtime error (!)

Given a View:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- ```
  View<int***> a;
  a = Kokkos::subview(v, ALL, 42, ALL);
  ```
  => Compilation error
- ```
  View<int*> a;
  a = Kokkos::subview(v, ALL, 5, 42);
  ```
  => Okay for LayoutLeft but => Compilation error for LayoutRight
- ```
  View<int**> a;
  a = Kokkos::subview(v, ALL, 15, ALL);
  ```
  => Runtime error (!)
- ```
  View<int**, LayoutStride> a;
  a = Kokkos::subview(v, ALL, 15, ALL);
  ```

Given a `View`:

```
Kokkos::View<int***> v("v", n0, n1, n2);
```

- `View<int***> a;`
  `a = Kokkos::subview(v, ALL, 42, ALL);`
  => Compilation error
- `View<int*> a;`
  `a = Kokkos::subview(v, ALL, 5, 42);`
  => Okay for LayoutLeft but => Compilation error for LayoutRight
- `View<int**> a;`
  `a = Kokkos::subview(v, ALL, 15, ALL);`
  => Runtime error (!)
- `View<int**, LayoutStride> a;`
  `a = Kokkos::subview(v, ALL, 15, ALL);`
  => Okay

- Use subviews to get a portion of a `View`. Helps with:
  - code reuse
  - code readability
  - library function compatibility

- Use subviews to get a portion of a `View`. Helps with:
  - code reuse
  - code readability
  - library function compatibility
- Kokkos supports slicing Views similar to Python/Matlab/Fortran slicing syntax

```
auto sv = Kokkos::subview(v, 42, ALL, std::make_pair(3, 17));
```

- Use subviews to get a portion of a `View`. Helps with:
  - code reuse
  - code readability
  - library function compatibility
- Kokkos supports slicing Views similar to Python/Matlab/Fortran slicing syntax

```
auto sv = Kokkos::subview(v, 42, ALL, std::make_pair(3, 17));
```

- The return type of `subview` is complicated. Use **auto**!!
- `View::operator=()` just does the "Right Thing"™
  - So generally don't worry about it at first! This is advanced stuff, and more for future reference.

Unmanaged Views: Dealing with external memory

Learning objectives:

- Why do you need unmanaged views
- Introduce unmanaged Views - basic capabilities and syntax
- Suggested usage and practices

Sometimes your Kokkos code can't control all allocations!

- Obviously best to avoid that unpleasant situation ...

But say you use some external function like IO classes:

```
struct MatrixReader {
  int N, M;
  std::vector<double> values;
  void read_file(std::string name) {...}
};
```

Sometimes your Kokkos code can't control all allocations!

- Obviously best to avoid that unpleasant situation ...

But say you use some external function like IO classes:

```
struct MatrixReader {
  int N, M;
  std::vector<double> values;
  void read_file(std::string name) {...}
};
```

How can you get this to the GPU without extra allocation?

Sometimes your Kokkos code can't control all allocations!

- Obviously best to avoid that unpleasant situation ...

But say you use some external function like IO classes:

```
struct MatrixReader {
  int N, M;
  std::vector<double> values;
  void read_file(std::string name) {...}
};
```

How can you get this to the GPU without extra allocation?

Unmanaged Views

Views can wrap existing allocations as Unmanaged Views.

Unmanaged View description:

- Normally, Views allocate memory and manage.
- Instead, Views can use externally controlled memory

Unmanaged View description:

- Normally, Views allocate memory and manage.
- Instead, Views can use externally controlled memory
- Caveats
  - No reference counting
  - No deallocation in the constructor
  - No check for the correct memory space
- Usages
  - Layout-punning: e.g., treat multidimensional View as one-dimensional views without copying
  - Use `std::vector` or memory from CUDA library, e.g. `cuSPARSE`

Back to our IO example:

```
struct MatrixReader {
  int N, M;
  std::vector<double> values;
  void read_file(std::string name) {...}
};
```

To create an unmanaged View:

- Provide a pointer as the first constructor argument.
- Give all the runtime dimensions.
- Make sure Layout and MemorySpace match!
- Unmanaged Views do NOT get a label!

Back to our IO example:

```
struct MatrixReader {
  int N, M;
  std::vector<double> values;
  void read_file(std::string name) {...}
};
```

To create an unmanaged View:

- Provide a pointer as the first constructor argument.
- Give all the runtime dimensions.
- Make sure Layout and MemorySpace match!
- Unmanaged Views do NOT get a label!

```
MatrixReader reader; reader.read_file("MM");
View<double**,LayoutRight,HostSpace>
  h_a(reader.values.data(),reader.N,reader.M);
```

Back to our IO example:

```
struct MatrixReader {
  int N, M;
  std::vector<double> values;
  void read_file(std::string name) {...}
};
```

To create an unmanaged View:

- Provide a pointer as the first constructor argument.
- Give all the runtime dimensions.
- Make sure Layout and MemorySpace match!
- Unmanaged Views do NOT get a label!

```
MatrixReader reader; reader.read_file("MM");
View<double**,LayoutRight,HostSpace>
  h_a(reader.values.data(),reader.N,reader.M);
```

How do we get this to the device?

In Module 2 we learned the Mirror Pattern

- But the mirror pattern started with a device view!

In Module 2 we learned the Mirror Pattern

- But the mirror pattern started with a device view!

Mirror in any Space

Kokkos::create_mirror_view can take a space argument for location of mirror.

In Module 2 we learned the Mirror Pattern

- But the mirror pattern started with a device view!

Mirror in any Space

Kokkos::create_mirror_view can take a space argument for location of mirror.

```
// Create mirror into default memory space
using space_t = DefaultExecutionSpace::memory_space;
auto a = create_mirror_view(space_t(), h_a);
// Copy values from the host to the device
deep_copy(a, h_a);
```

In Module 2 we learned the Mirror Pattern

- But the mirror pattern started with a device view!

Mirror in any Space

Kokkos::create_mirror_view can take a space argument for location of mirror.

```
// Create mirror into default memory space
using space_t = DefaultExecutionSpace::memory_space;
auto a = create_mirror_view(space_t(), h_a);
// Copy values from the host to the device
deep_copy(a, h_a);
```

Since the “create mirror and then copy” pattern is common we have a shortcut:

```
auto a = create_mirror_view_and_copy(space_t(), h_a);
```

Using pre-allocated scratch memory for temporary data structures is common to:

- Eliminate costly allocation/deallocation operations
- Reduce total memory footprint.

Using pre-allocated scratch memory for temporary data structures is common to:

- Eliminate costly allocation/deallocation operations
- Reduce total memory footprint.

Unmanaged Views of Scratch Allocations

Unmanaged Views can be used to get arrays of different shapes backed by the same memory.

```
void* scratch = kokkos_malloc<>("Scratch", scratch_size);
View<double**> a_scr(scratch, N,M);
View<int*> b_scr(scratch,K);
```

Using pre-allocated scratch memory for temporary data structures is common to:

- Eliminate costly allocation/deallocation operations
- Reduce total memory footprint.

Unmanaged Views of Scratch Allocations

Unmanaged Views can be used to get arrays of different shapes backed by the same memory.

```
void* scratch = kokkos_malloc<>("Scratch", scratch_size);
View<double**> a_scr(scratch, N,M);
View<int*> b_scr(scratch,K);
```

How much memory do you need for a `View`?

```
int scratch_size = View<double**>::required_allocation_size(N,M);
```

- Unmanaged Views wrap existing allocations
  - No ref-counting
  - No deallocation after losing scope
  - No memory space checks

- Unmanaged Views wrap existing allocations
  - No ref-counting
  - No deallocation after losing scope
  - No memory space checks
- Unmanaged view is created with pointer and runtime dimensions

- Unmanaged Views wrap existing allocations
  - No ref-counting
  - No deallocation after losing scope
  - No memory space checks
- Unmanaged view is created with pointer and runtime dimensions

```
void* ptr = kokkos_malloc<>("Alloc", alloc_size);
View<double**> h_a((double*)ptr,N,M);
```

- Unmanaged Views wrap existing allocations
  - No ref-counting
  - No deallocation after losing scope
  - No memory space checks
- Unmanaged view is created with pointer and runtime dimensions

```
void* ptr = kokkos_malloc<>("Alloc", alloc_size);
View<double**> h_a((double*)ptr,N,M);
```

- Unmanaged view uses
  - Access externally controlled memory
  - Access temporary scratch memory
  - Layout pruning - view underlying data using different layout

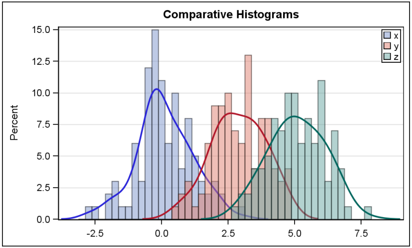
Thread safety and atomic operations

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.

Histogram kernel:

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const size_t bucketIndex = computeBucketIndex(value);
    ++_histogram(bucketIndex);
  });
```

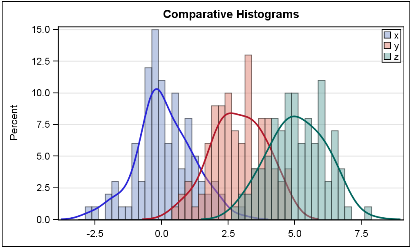


http://www.farmaceuticas.com.br/tag/graficos/

Histogram kernel:

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const size_t bucketIndex = computeBucketIndex(value);
    ++_histogram(bucketIndex);
  });
```

Problem: Multiple threads may try to write to the same location.



http://www.farmaceuticas.com.br/tag/graficos/

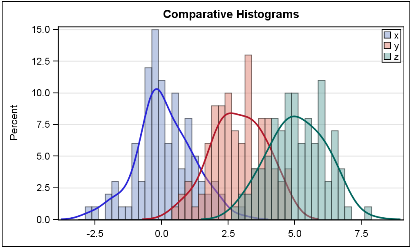
Histogram kernel:

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const size_t bucketIndex = computeBucketIndex(value);
    ++_histogram(bucketIndex);
  });
```

Problem: Multiple threads may try to write to the same location.

Solution strategies:

- Locks: not feasible on GPU
- Thread-private copies: not thread-scalable
- Atomics



http://www.farmaceuticas.com.br/tag/graficos/

Atomics: the portable and thread-scalable solution

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const int bucketIndex = computeBucketIndex(value);
    Kokkos::atomic_add(&_histogram(bucketIndex), 1);
  });
```

Atomics: the portable and thread-scalable solution

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const int bucketIndex = computeBucketIndex(value);
    Kokkos::atomic_add(&_histogram(bucketIndex), 1);
  });
```

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

Atomics: the portable and thread-scalable solution

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const int bucketIndex = computeBucketIndex(value);
    Kokkos::atomic_add(&_histogram(bucketIndex), 1);
  });
```

- Atomics are the **only scalable** solution to thread safety.
- Locks are **not portable**.

Atomics: the portable and thread-scalable solution

```
parallel_for(N, KOKKOS_LAMBDA(const size_t index) {
    const Something value = ...;
    const int bucketIndex = computeBucketIndex(value);
    Kokkos::atomic_add(&_histogram(bucketIndex), 1);
  });
```

- Atomics are the **only scalable** solution to thread safety.
- Locks are **not portable**.
- Data replication is **not thread scalable**.

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);
}
```

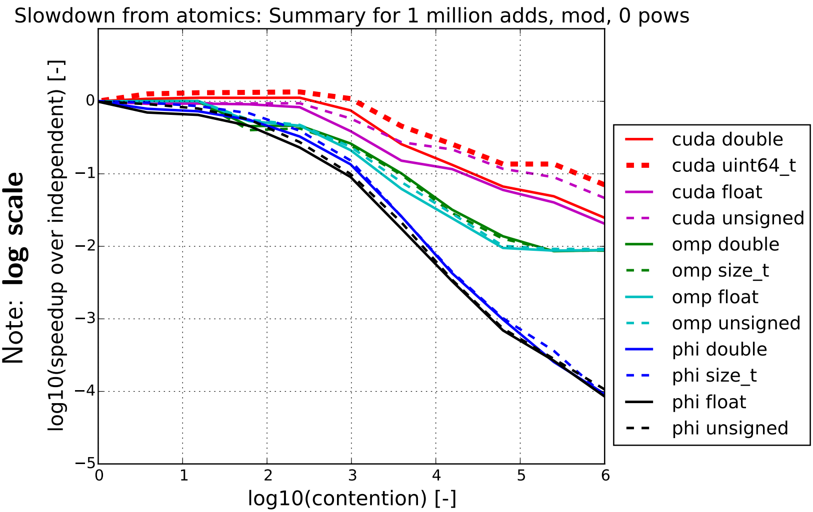
- This is the most extreme case: all coordination and no work.
- Contention is captured by the `atomicStride`.

  `atomicStride` $\rightarrow$ 1 $\Rightarrow$ Scalar integration (bad)

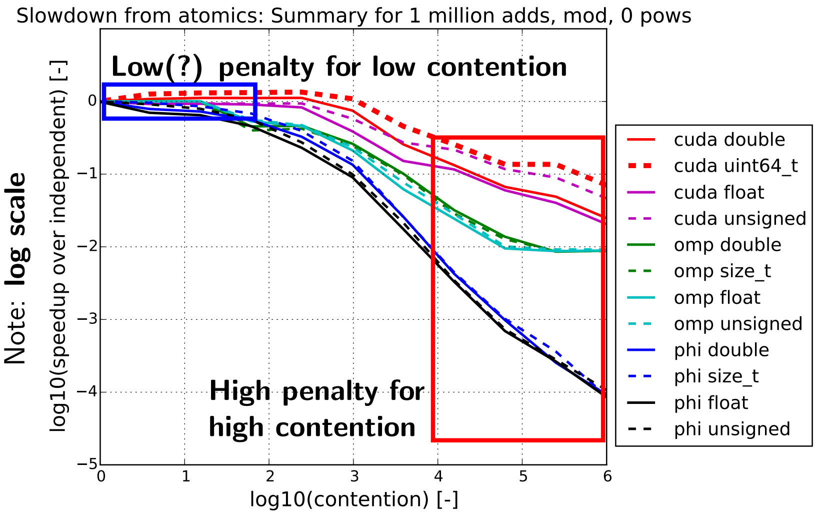
  `atomicStride` $\rightarrow$ large $\Rightarrow$ Independent (good)

Atomics performance: 1 million adds, **no** work per kernel

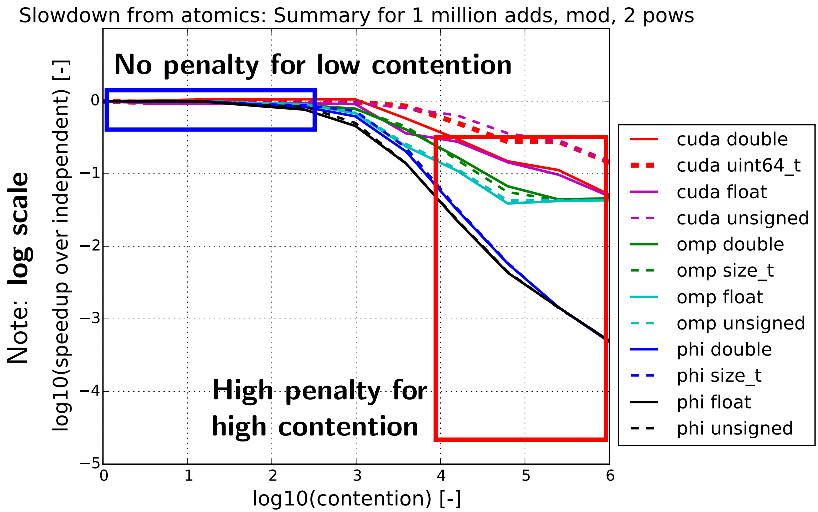
Note: **log scale**



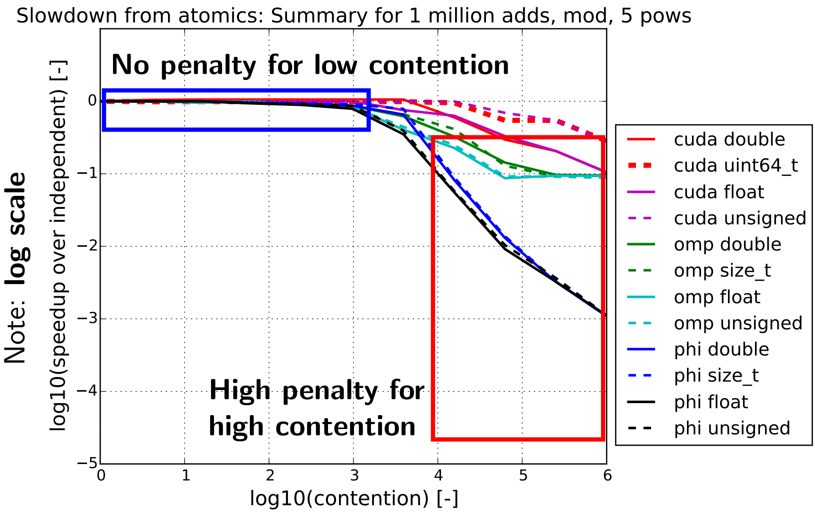
Atomics performance: 1 million adds, **no** work per kernel



Atomics performance: 1 million adds, **some** work per kernel



Atomics performance: 1 million adds, **lots of** work per kernel



Atomics on arbitrary types:

- Atomic operations work if the corresponding operator exists, i.e., `atomic_add` works on any data type with "+".
- Atomic exchange works on any 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);
```

Slight detour: `View` memory traits:

- Beyond a `Layout` and `Space`, `Views` can have memory traits.
- Memory traits either provide **convenience** or allow for certain **hardware-specific optimizations** to be performed.

Example: If all accesses to a `View` will be atomic, use the `Atomic` memory trait:

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

Slight detour: `View` memory traits:

- Beyond a `Layout` and `Space`, `Views` can have memory traits.
- Memory traits either provide **convenience** or allow for certain **hardware-specific optimizations** to be performed.

Example: If all accesses to a `View` will be atomic, use the `Atomic` memory trait:

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

Many memory traits exist or are experimental, including `Atomic`, `Unmanaged`, `Restrict`, and `RandomAccess`.

Example: `RandomAccess` **memory trait:**

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

Example: `RandomAccess` memory trait:

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

In the early days you had to access this 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);
```

Example: `RandomAccess` memory trait:

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

In the early days you had to access this 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);
```

Kokkos can hide mechanisms like that as simple as:

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

Histogram generation is an example of the **Scatter Contribute** pattern.

- Like a reduction but with many results.
- Number of results scales with number of inputs.
- Each results gets contributions from a small number of inputs/iterations.
- Uses an inputs-to-results map not inverse.

Examples:

- Particles contributing to neighbors forces.
- Cells contributing forces to nodes.
- Computing histograms.
- Computing a density grid from point source contributions.

Compute forces on particles via neighbor contributions

This kernel uses Newtons Third Law: Actio = Reactio

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
  int N = x.extent(0);
  int num_neighs = neighs.extent(1);
  parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
    for(int j=0; j<num_neighs; j++) {
      real3 df = force.compute(x(i),x(neighs(i,j)));
      f(i) += df;
      f(j) -= df;
    }
  });
}
```

Compute forces on particles via neighbor contributions

This kernel uses Newtons Third Law: Actio = Reactio

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
  int N = x.extent(0);
  int num_neighs = neighs.extent(1);
  parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
    for(int j=0; j<num_neighs; j++) {
      real3 df = force.compute(x(i),x(neighs(i,j)));
      f(i) += df;
      f(j) -= df;
    }
  });
}
```

This kernel has a race condition on `f` though!

There are two useful algorithms:.

- **Atomics:** thread-scalable but depends on atomic performance.
- **Data Replication:** every thread owns a copy of the output, not thread-scalable but good for low (< 16) threads count architectures.

There are two useful algorithms:.

- **Atomics:** thread-scalable but depends on atomic performance.
- **Data Replication:** every thread owns a copy of the output, not thread-scalable but good for low (< 16) threads count architectures.

Important Capability: ScatterView

ScatterView can transparently switch between **Atomic** and **Data Replication** based scatter algorithms.

There are two useful algorithms:.

- **Atomics:** thread-scalable but depends on atomic performance.
- **Data Replication:** every thread owns a copy of the output, not thread-scalable but good for low (< 16) threads count architectures.

Important Capability: ScatterView

ScatterView can transparently switch between **Atomic** and **Data Replication** based scatter algorithms.

- Abstracts over scatter contribute algorithms.
- Compile time choice with backend-specific defaults.
- Only limited number of operations are supported.
- Part of Kokkos Containers (in Kokkos 3.2 still experimental).

Creating a ScatterView:

Usually a `ScatterView` wraps an existing `View`

Creating a ScatterView:

Usually a `ScatterView` wraps an existing `View`

- Allows the **atomic** variant to work without extra allocation.

Creating a ScatterView:

Usually a `ScatterView` wraps an existing `View`

- Allows the **atomic** variant to work without extra allocation.

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
Kokkos::Experimental::ScatterView<real3*> scatter_f(f);
...
```

Creating a ScatterView:

Usually a `ScatterView` wraps an existing `View`

- Allows the **atomic** variant to work without extra allocation.

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
Kokkos::Experimental::ScatterView<real3*> scatter_f(f);
...
```

Accessing the `ScatterView`

In the kernel obtain an atomic or thread-local accessor.

```
parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
  auto f_a = scatter_f.access();
  for(int j=0; j<num_neighs; j++) {
    real3 df = force.compute(x(i),x(neighs(i,j)));
    f_a(i) += df;
    f_a(j) -= df;
  }
});
```

Creating a ScatterView:

Usually a `ScatterView` wraps an existing `View`

- Allows the **atomic** variant to work without extra allocation.

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
Kokkos::Experimental::ScatterView<real3*> scatter_f(f);
...
```

Accessing the `ScatterView`

In the kernel obtain an atomic or thread-local accessor.

```
parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
  auto f_a = scatter_f.access();
  for(int j=0; j<num_neighs; j++) {
    real3 df = force.compute(x(i),x(neighs(i,j)));
    f_a(i) += df;
    f_a(j) -= df;
  }
});
```

Only the += and -= operators are available!

We are missing one step though:

We are missing one step though: **Contribute back to the original view.**

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
  Kokkos::Experimental::ScatterView<real3*> scatter_f(f);
  parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
    ...
  });
  Kokkos::Experimental::contribute(f,scatter_f);
```

We are missing one step though: **Contribute back to the original view.**

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
  Kokkos::Experimental::ScatterView<real3*> scatter_f(f);
  parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
    ...
  });
  Kokkos::Experimental::contribute(f,scatter_f);
```

- No-op when `scatter_f` uses **atomic** access
- Combines thread-local arrays in case of data duplication

We are missing one step though: **Contribute back to the original view.**

```
void compute_forces(View<real3*> x, View<real3*> f,
                    View<int**> neighs, Interaction force) {
  Kokkos::Experimental::ScatterView<real3*> scatter_f(f);
  parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
    ...
  });
  Kokkos::Experimental::contribute(f,scatter_f);
```

- No-op when `scatter_f` uses **atomic** access
- Combines thread-local arrays in case of data duplication

Important Point

Reuse ScatterView if possible: creating and destroying data duplicates is costly and should be avoided

When reusing a `ScatterView` the duplicates have to be reset.

```
scatter_f.reset();
```

When reusing a `ScatterView` the duplicates have to be reset.

```
scatter_f.reset();
```

The complete picture:

```
void compute_forces(View<real3*> x, View<real3*> f,
                    ScatterView<real3*> scatter_f,
                    View<int**> neighs, Interaction force) {
  scatter_f.reset();
  int N = x.extent(0);
  int num_neighs = neighs.extent(1);
  parallel_for("ForceCompute", N, KOKKOS_LAMBDA(int i) {
    auto f_a = scatter_f.access();
    for(int j=0; j<num_neighs; j++) {
      real3 df = force.compute(x(i),x(neighs(i,j)));
      f_a(i) += df;
      f_a(j) -= df;
    }
  });
  Kokkos::Experimental::contribute(f,scatter_f);
}
```

But I need something else than a Sum!

But I need something else than a Sum!

`ScatterView` has more options including the reduction op.

```
template<class DataType, class Layout, class Space,
         class Operation, int Duplication, int Contribution>
class ScatterView;
```

- `DataType`, `Layout`, `Space`: as in `Kokkos::View`
- `Operation`: `ScatterSum`, `ScatterProd`, `ScatterMin`, or `ScatterMax`.
- `Duplication`: Whether to duplicate values per thread.
- `Contribution`: Whether to use **atomics**.

- Location: `Exercises/scatter_view/Begin/`
- Assignment: Convert scatter_view_loop to use `ScatterView`.
- Compile and run on both CPU and GPU

```
make -j KOKKOS_DEVICES=OpenMP  # CPU-only using OpenMP
make -j KOKKOS_DEVICES=Cuda    # GPU - note UVM in Makefile
# Run exercise
./scatterview.host
./scatterview.cuda
# Note the warnings, set appropriate environment variables
```

- Compare performance on CPU of the three variants
- Compare performance on GPU of the two variants
- Vary problem size: first and second optional argument

Section Summary

- Atomics are the only thread-scalable solution to thread safety.
  - Locks or data replication are **not portable or scalable**
- 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.
- 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
- Many programs with the **scatter-add** pattern can be thread-scalably parallelized using atomics without much modification.

DualView

Learning objectives:

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

Motivation and Value-added

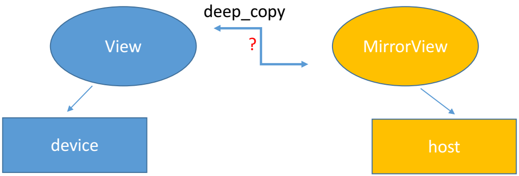
- DualView was designed to help transition codes to Kokkos.

Motivation and Value-added

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

Motivation and Value-added

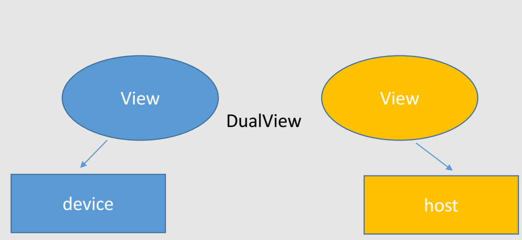
- DualView was designed to help transition codes to Kokkos.
- DualView simplifies the task of managing data movement between memory spaces, e.g., host and device.
- When converting a typical app to use Kokkos, there is usually no holistic view of such data transfers.



Without DualView, could use MirrorViews, but

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

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



There is no automatic tracking of data freshness:

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

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

- Data members for the two views

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

- Retrieve data members

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

- Mark data as modified

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

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

- Sync data in a direction if not in sync

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

- Check sync status

```
void need_sync_host();
void need_sync_device();
```

DualView has templated functions for generic use in templated code

- Retrieve data members

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

- Mark data as modified

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

- Sync data in a direction if not in sync

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

- Check sync status

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

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

Details:

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

```
# Compile for CPU
make -j KOKKOS_DEVICES=OpenMP
# Compile for GPU (we do not need UVM anymore)
make -j KOKKOS_DEVICES=Cuda
# Run on GPU
./dualview.cuda -S 26
```

MDRangePolicy

- Tightly nested loops (similar to OpenMP collapse clause)
- Available with `parallel_for` and `parallel_reduce`
- Tiling strategy over the iteration space
- Control iteration pattern at compile time

```
View<double**,LayoutLeft> A("A",N0,N1);
parallel_for("Label",
  MDRangePolicy<Rank<2,Iterate::Left,Iterate::Left>>(
        {0,0},{N0,N1}),
  KOKKOS_LAMBDA(int i, int j) {
    A(i,j) = 1000.0 * i + 1.0*j;
});
```

Subviews

- Taking slices of Views
- Similar capability as provided by Matlab, Fortran, or Python
- Prefer the use of auto for the type

```
View<int ***> v("v", N0, N1, N2);
auto sv = subview(v, i0, ALL, make_pair(start,end));
```

Unmanaged Views

- Interoperability with externally allocated arrays
- No reference counting, memory not deallocated at destruction
- User is responsible for insuring proper dynamic and/or static extents, MemorySpace, Layout, etc.

```
View<float**, LayoutRight, HostSpace>
  v_unmanaged(raw_ptr, N0, N1);
```

Atomic operations

- Atomic functions available on the host or the device (e.g. `Kokkos::atomic_add`)
- Use `Atomic` memory trait for atomic accesses on Views

```
View<int*> v("v", N0);
View<int*, MemoryTraits<Atomic>> v_atomic = v;
```

- Use `ScatterView` for scatter-add parallel pattern

Dual Views

- For managing data synchronization between host and device
- Helps in codes with no holistic view of data flow
  - In particular when porting codes incrementally

Hierarchical Parallelism

- How to leverage more parallelism through nested loops.
- The concept of Thread-Teams and Vectorlength.

Scratch Space

- Getting temporary workspace in kernels.
- Leveraging GPU Shared Memory.

Unique Token

- How to acquire safely per-thread resources.

Don't Forget: Join our Slack Channel and drop into our office hours on Tuesday.
Updates at: bit.ly/kokkos-lecture-updates
Recordings/Slides: bit.ly/kokkos-lecture-wiki