Hands On: Kokkos

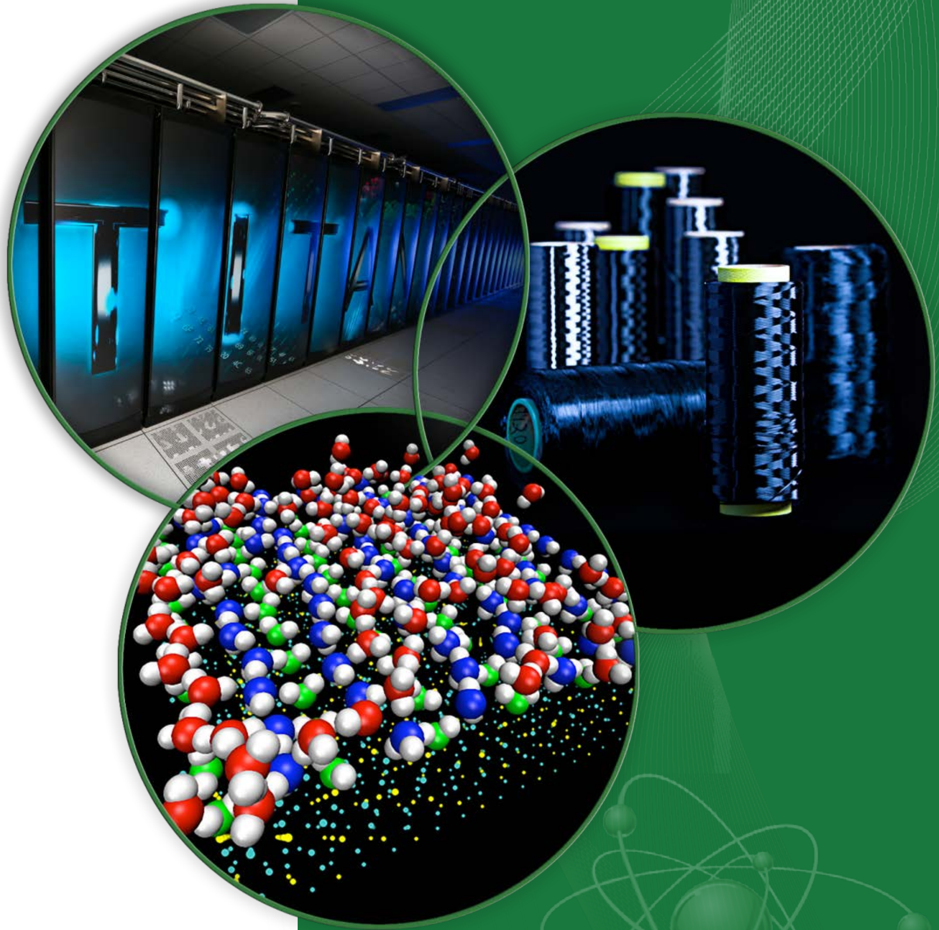
Verónica G. Vergara Larrea

Adam B. Simpson

Tom Papatheodore

CSGF 2017

July 26, 2017



The need for Kokkos

- *HPC architectures are increasingly heterogeneous*
  - Specialized cores, deep layers of hierarchical memory
  - Diverse architectures mitigate risk and are required
- *Portable software solutions exist*
  - Allows a single code base to run on multiple architectures
  - OpenACC, OpenMP 4.x, OpenCL, Kokkos
- *Performance portability not addressed*
  - Require multiple versions of code optimized for each architecture

Kokkos

- **κόκκος** (Greek)
  - “granule” or “grain”
- *C++ templated library*
  - Familiar C++ interface, not a new language or extension
  - Open source and extensible
  - Target architecture abstracted away
  - Goal of having C++20 absorb Kokkos functionality
- *Portable performance*
  - Targets many architectures
    - Multi-core CPU(X86, Power), Intel MIC, NVIDIA CUDA, AMD APU
  - Architecture aware memory access/work scheduling

Parallel dispatch

Parallelism is primarily extracted from for loops so we wish to translate this:

```
for(int i=0; i<count; i++){
   c[i] = a[i] + b[i];
};
```

Into the kokkos parallel equivalent:

```
parallel_for(count,  [=](int i){
   c[i] = a[i] + b[i];
});
```

Parallel Dispatch

Kokkos abstracts the parallel dispatch of work.
Using this abstraction a for loop may be broken down as such:

```
for(int i=0; i<count; i++){
    c[i] = a[i] + b[i];
};
```

Pattern: The operation that will dispatch work

Policy: How the pattern will dispatch work

Body: Code which constitutes a single work item

Pattern

The parallel_for pattern is the most commonly used. It implements a for loop with independent iterations.

```
parallel_for(count, [=](int i){
    c[i] = a[i] + b[i];
});
```

Unlike a traditional serial for loop there is **no** guarantee by Kokkos as to the ordering of work items in a parallel_for pattern, this allows work items to safely be dispatched on parallel architectures.

Pattern

Not all for loops map to the parallel_for pattern though:

```
for(i=0; i<count; i++) {
   sum += a[i];
};
```

```
parallel_for(count,  [&](int i){
   sum += a[i];
});
```

This work_item creates a dependency between the i-1 and i work items. To ensure no race condition is encountered the appropriate pattern must be applied, in this case a reduction:

```
parallel_reduce(count,  [=](int i, auto local_sum){
   local_sum = a[i];
}, sum)
```

Here local_sum is a thread private variable passed to our work item and then reduced into sum

Pattern

Not all for loops map to the parallel_for pattern though:

```
for(i=1; i<count; i++) {
   a[i] = a[i] + a[i-1];
};
```

```
parallel_for(count,  [=](int i){
   a[i] = a[i] + a[i-1];
});
```

This work_item creates a loop carried dependency between the i-1 and i work items. To ensure no race condition is encountered the appropriate pattern must be applied, in this case a scan:

```
parallel_scan(count, PrefixSum(...));
```

Here PrefixSum() is customizable by the user and can be used for scan based patterns

Policy

The most basic policy is to provide the pattern a single integer, count in this example, as the first argument. This specifies the pattern should be applied across the integer range [0,count)

```
for(int i=0; i<count; i++){
...
};
```

$\rightarrow$

```
parallel_for(count, [=](int i){
...
});
```

Policy

The most basic policy is to provide the pattern a single integer, count in this example, as the first argument. This specifies the pattern should be applied across the integer range [0,count)

```
for(int i=0; i<count; i++){
...
};
```

→

```
parallel_for(count, [=](int i){
...
});
```

This creates an implicit RangePolicy(), the most basic execution policy.

```
for(int i=0; i<count; i++){
...
};
```

→

```
parallel_for(
   RangePolicy(0, count),
   [=](int i){
     ...
});
```

Body

Kokkos allows the computational body to be provided through either a lambda or functor passed to the pattern. The signature of the lamda/functor depends on the specified pattern. For example parallel_for passes a single integer value to the computational body which returns void

```
float *a,b,c = new float[count];
parallel_for(count,  [=](int i) {
    c[i] = a[i] + b[i];
});
```

Kokkos semantics require lambda variables be captured by value only. Additionally some architectures require special lambda decorators. As such it's recommended that the capture list, [=], be replaced by the portable macro KOKKOS_LAMBDA

```
parallel_for(count, KOKKOS_LAMBDA (int i) {
    c[i] = a[i] + b[i];
});
```

Body

Using a functor with parallel_for requires creating a class containing a function with the following signature: KOKKOS_INLINE_FUNCTION void operator() (integer_type) const

Body

Using a functor with parallel_for requires creating a class containing a function with the following signature: KOKKOS_INLINE_FUNCTION void operator() (integer_type) const

```
float *a,b,c = new float[count]
MyFunctor body(a, b, c);
parallel_for(count, body);

class MyFunctor {
 public:
  MyFunctor(float *a, float *b, float *c) : a{a}, b{b}, c{c} {}

  KOKKOS_INLINE_FUNCTION
  void operator() (int i) const {
     c[i] = a[i] + b[i];
   }

 private:
  float *a, *b, *c;
};
```

Body

KOKKOS_INLINE_FUNCTION is a macro defined to enable a portable way to designate a function may be used in a computational body. Any function used in a parallel region must have this decorator applied to it.

```
KOKKOS_INLINE_FUNCTION
float my_rand() {
  return 7.0;
}

parallel_for(count, KOKKOS_LAMBDA (int i) {
    a[i] = my_rand();
});
```

This means the C++ standard library is largely **unavailable** for use. The exception to this is that most transcendental math operations are available(*sin, cos, log, pow,*…).

Execution Space

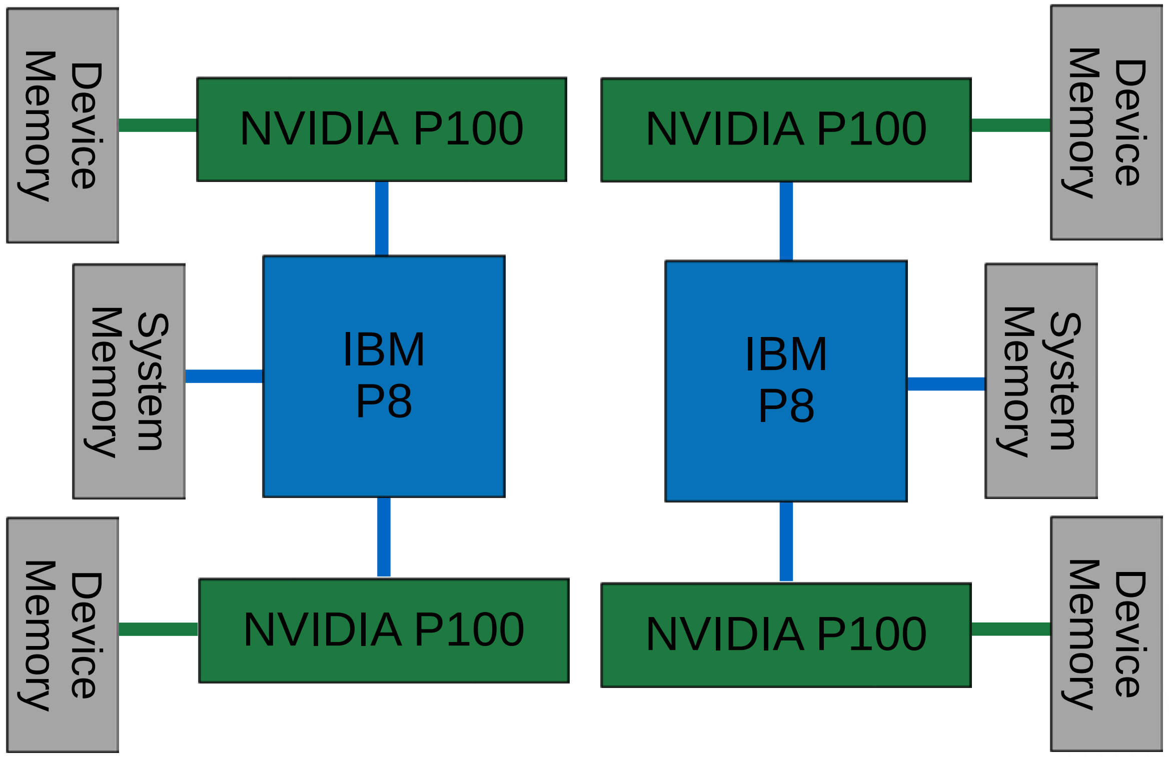
Where will the following work be executed?

```
float *a,b,c = new float[count];
parallel_for(count, KOKKOS_LAMBDA (int i) {
   c[i] = a[i] + b[i];
});
```

To answer this question Kokkos uses the abstraction of execution spaces which describe “places” code can execution. These spaces are comprised of not only physical hardware but also the backend frameworks that enable code to execute on them.

Execution Space

On Summit several execution spaces exist that target the Power8 CPU and the NVIDIA GPU.



- CUDA
- OpenMP
- Pthreads
- Serial

Execution Space

Kokkos sets the **default** execution space at compile time by searching through the following list and picking the first space that's enabled:

- CUDA
- OpenMP
- Pthreads
- Serial

This is the space in which parallel work will be dispatched to unless specified otherwise.

Execution Space

The execution space may be explicitly set through the execution policy template parameter:

```
parallel_for(
   RangePolicy<CUDA>(0, count),
   KOKKOS_LAMBDA(int i){
     ...
});
```

```
parallel_for(
   RangePolicy<OpenMP>(0, count),
   KOKKOS_LAMBDA(int i){
     ...
});
```

```
parallel_for(
   RangePolicy<Serial>(0, count),
   KOKKOS_LAMBDA(int i){
     ...
});
```

If the template parameter is not set the default execution space is used

Views

What happens when the following code runs?

```
float *a = new float[count];

parallel_for(
   RangePolicy<CUDA>(0, count),
   KOKKOS_LAMBDA(int i){
     a[i] = i;
});
```

Views

What happens when the following code runs?

```
float *a = new float[count];

parallel_for(
   RangePolicy<CUDA>(0, count),
   KOKKOS_LAMBDA(int i){
     a[i] = i;
});
```

SEGFAULT! We’ve passed a host memory pointer, a, to work which will be dispatched to the GPU. The GPU does not generally have access to host memory and so this will result in a runtime failure.

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Views

To ensure Kokkos is portable regardless of the execution space used multidimensional array-like containers called views are provided. This view abstraction also ensures performant memory usage as we will see later.

1D

```
float *a = new float[count];
cudaMalloc(&a, count*sizeof(float));
vector<float>(count) a;
```

```
View<float*> a(“a”, count);
```

Views

To ensure Kokkos is portable regardless of the execution space used multidimensional array-like containers called views are provided. This view abstraction also ensures performant memory usage as we will see later.

1D

```
float *a = new float[count];
cudaMalloc(&a, count*sizeof(float));
vector<float>(count) a;
```

$\rightarrow$

```
View<float*> a("a", count);
```

2D

```
vector<
  vector<float>(count_x)>
    (count_y) a;
```

$\rightarrow$

```
View<float**> a("a", count_x,count_y);
```

Views

To ensure Kokkos is portable regardless of the execution space used multidimensional array-like containers called views are provided. This view abstraction also ensures performant memory usage as we will see later.

1D

```
float *a = new float[count];
cudaMalloc(&a, count*sizeof(float));
vector<float>(count) a;
```

→

```
View<float*> a("a", count);
```

2D

```
vector<
 vector<float>(count_x)>
  (count_y) a;
```

→

```
View<float**> a("a", count_x,count_y);
```

3D

```
vector<
 vector<
  vector<float>
(count_x)> (count_y)>(count_z) a;
```

→

```
View<float***> a("a", count_x,count_y,count_z);
```

Views

Data types should be a C++ primitive or Plain Old Data

Number of dimensions

```
View<float**> data("data", count_x, count_y);
```

String label used for debugging

Views

- Accessing memory in a view is done through the () operator

```
View<float**> data("data", count_x, count_y);
float elem = data(i, j);
```

- Views are light weight(*think pointer* + *shape*) and designed to be passed by value. The underlying data is **NOT** copied when a view is copied(no implicit deep copies)

```
View<float*> data("data", count);

parallel_for(count, KOKKOS_LAMBDA(int i){
    data(i) = i;
});
```

- Views follow std::shared_ptr semantics and do not need to be explicitly deallocated.

Memory space

memory spaces describe where memory belonging to a view is located. By default memory belonging to a view will be allocated in the default memory space. This default memory space is accessible to the default execution space so the following should “just work”

```
View<float*> data(“data”, count);

parallel_for(count, KOKKOS_LAMBDA(int i){
    data(i) = i;
});
```

Memory space

The memory space used by a view may be explicitly set as a template parameter:

```
View<float*, Host> data("data", count);
View<float*, Cuda> data("data", count);
View<float*, CudaUVM> data("data", count);
View<float*, OpenMP> data("data", count);
View<float*, ...> data("data", count);
```

Memory Layout

Controlling the memory space provides a portable way to allocate and access data in a heterogeneous environment but doesn’t doesn’t address performant memory access. To ensure performance Kokkos introduces memory layouts which describes how a view maps a multidimensional index a(i,j,k) into an actual location in memory.

2 common layouts

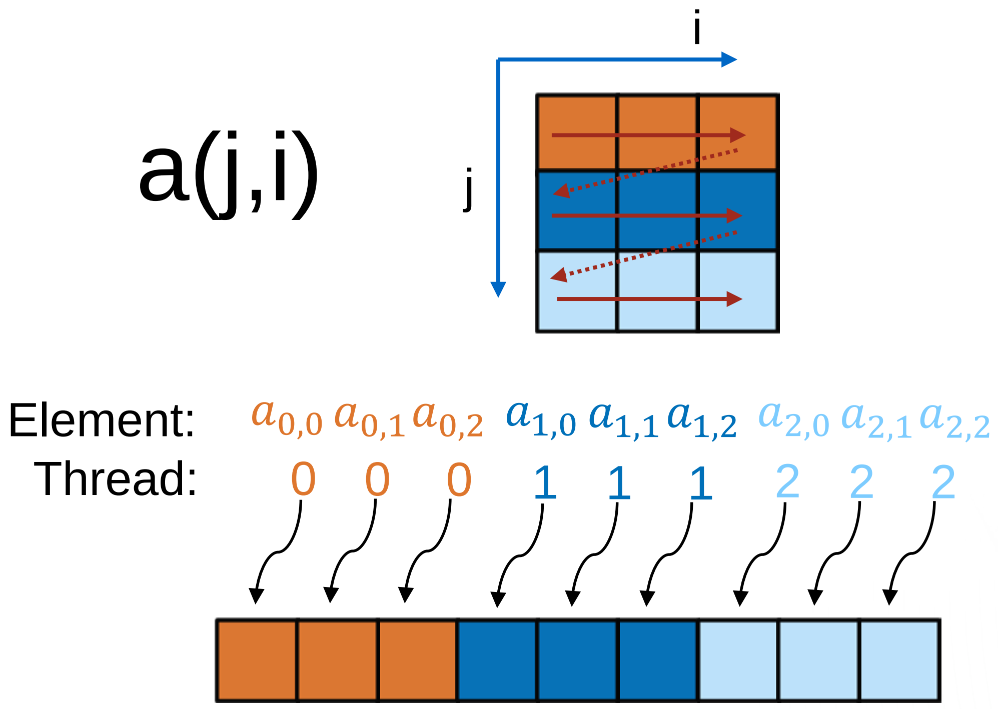
```
View<float*, Cuda> a(“a”, count);   →   View<float*, LayoutRight, Cuda> a(“a”, count);
```

```
View<float*, OpenMP> a(“a”, count);   →   View<float*, LayoutLeft, OpenMP> a(“a”, count);
```

In most cases it is best to let Kokkos choose the layout implicitly. Performant access is achieved as long as the loop iteration indices correspond to the first index in the view.

LayoutRight

LayoutRight, or row major format, is the standard C/C++ array format. The right most index is the fastest running dimension in memory and the left most the slowest. Threads run across the slow memory dimension.

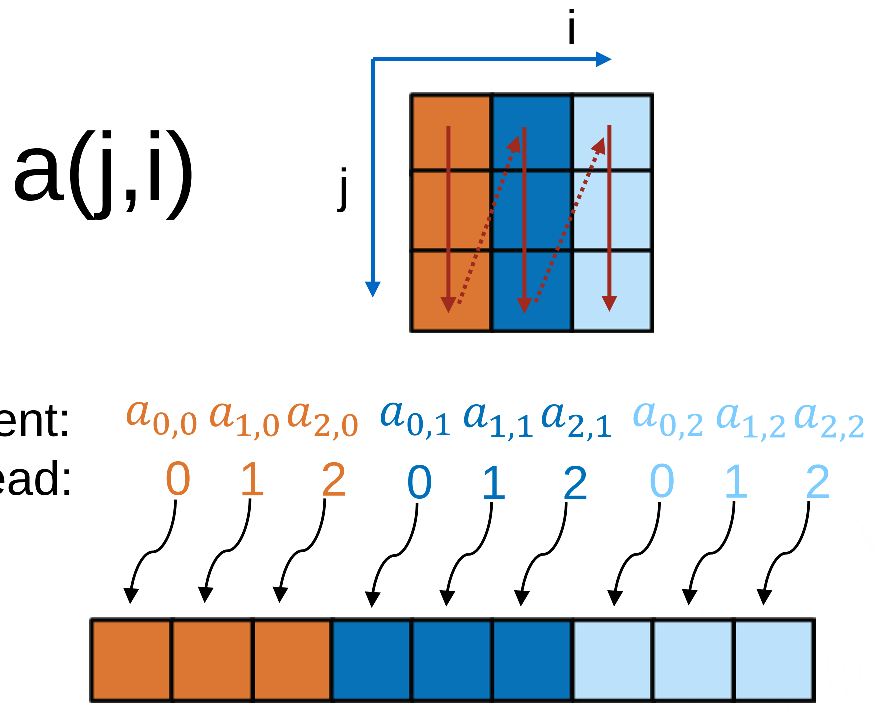


a(j,i)
i
j
Element: $a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{2,0}$ $a_{2,1}$ $a_{2,2}$
Thread: 0 0 0 1 1 1 2 2 2

Each thread access consecutive memory locations resulting in **cached** access. This is required for performant use of CPU like architectures

LayoutLeft

LayoutLeft, or column major format, is the standard Fortran array format. The left most index is the fastest running dimension in memory and the right most the slowest. Threads run across the fast memory dimension.


a(j,i)
i
j
Element: $a_{0,0}$ $a_{1,0}$ $a_{2,0}$ $a_{0,1}$ $a_{1,1}$ $a_{2,1}$ $a_{0,2}$ $a_{1,2}$ $a_{2,2}$
Thread: 0 1 2 0 1 2 0 1 2

Consecutive threads access consecutive memory locations resulting in **coalesced** access. This is required for performant use of GPU like architectures.

View

Not all execution spaces can access view data contained in a particular memory space. One common example of this is accessing the CUDA memory space from the implicit host execution space.

View

Not all execution spaces can access view data contained in a particular memory space. One common example of this is accessing the CUDA memory space from the implicit host execution space.

```
View<float*, CUDA> data("data", count);

for(int i=0; i<count; i++) {
   some_initilization(data(i));
}

parallel_for(
   RangePolicy<CUDA>(0, count),
   KOKKOS_LAMBDA(int i){
     data(i) = i;
});
```

Although not explicitly set this loop will run in the host execution space, where accessing GPU memory will likely result in a SEGFAULT

View

Not all execution spaces can access view data contained in a particular memory space. One common example of this is accessing the CUDA memory space from the implicit host execution space.

```
View<float*, CUDA> data("data", count);

for(int i=0; i<count; i++) {
   some_initilization(data(i));
}

parallel_for(
   RangePolicy<CUDA>(0, count),
   KOKKOS_LAMBDA(int i){
     data(i) = i;
});
```

Although not explicitly set this loop will run in the host execution space, where accessing GPU memory will likely result in a SEGFAULT

To facilitate accessing data in multiple memory spaces Kokkos provides mirrors

Mirror

Mirrors are views of equivalent arrays residing in different memory spaces. Mirrors allow deep copying between different memory spaces; A feature not available with standard views due to potential differences in data layout and alignment.

```
View<float*, CUDA> data("data", count);
auto host_data = create_mirror_view(data);

for(int i=0; i<count; i++) {
   some_initilization(host_data(i));
}

deep_copy(data, host_data);

parallel_for(
   RangePolicy<CUDA>(0, count),
   KOKKOS_LAMBDA(int i){
     data(i) = i;
});
```

} Create host mirror of data

} Modify host mirror on the host

} Update data with host mirror values

Initialize

Before using any Kokkos features your application must initialize the Kokkos environment. This requires inclusion of the Kokkos_core.hpp header file:

```
Kokkos::initialize(int &argc, char* argv[]);
```

Before program exit your application should make a corresponding call to finalize the Kokkos environment:

```
Kokkos::finalize();
```

Initialization arguments

Kokkos::initialize() parses the application argument list to find Kokkos specific flags. For example on SummitDev "--*kokkos-ndevices" can be used to specify the number of GPUs per node, which will be assigned round-robin per MPI process to be used by Kokkos.*

```
mpirun –n 16 a.out --kokkos-ndevices=4
```

OAK RIDGE National Laboratory | OAK RIDGE LEADERSHIP COMPUTING FACILITY

Building

Although Kokkos is largely a header library there is a runtime component that must be built and linked against your application. Kokkos provides **Makefile.kokkos** which is designed to be included in your applications base Makefile.

```
CXX=g++

default: main

include Makefile.kokkos

main: $(KOKKOS_LINK_DEPENDS) $(KOKKOS_CPP_DEPENDS) main.cpp
   $(CXX) $(KOKKOS_CPPFLAGS) $(KOKKOS_CXXFLAGS) \
   $(KOKKOS_LDFLAGS) $(KOKKOS_LIBS) main.cpp -o main
```

Useful extras

A compilable Kokkos sample that can target both the Power8 as well the Pascal GPU can be found in the following repo: vector addition sample. It is recommended that the included makefile be used as the basis of your Mandelbrot codes makefile.

Useful extras

Kokkos has experimental support for multi-dimensional ranges

```
using range2d_t = Kokkos::Experimental::MDRangePolicy<
                                                    Kokkos::Experimental::Rank<2> ,
                                                     Kokkos::IndexType<int>
                                                  >;
range2d_t range( {0,0}, {WIDTH, HEIGHT} );
Kokkos::Experimental::md_parallel_for(range, KOKKOS_LAMBDA(int i, int j) {
  ...
});
```

Useful extras

Kokkos supports portable STL like complex and pair types

```
Kokkos::complex<double> C(x, y);
double magnitude = Kokkos::abs(C);
auto squared = C*C;
```

```
auto my_pair = Kokkos::make_pair(some_int, some_double);

int first;
double second;
Kokkos::tie(first, second) = my_pair;
```

Useful extras

Kokkos supports a basic timer interface

```
Kokkos::Timer timer();
…work to time…
double elapsed_seconds = Timer.seconds();

Timer.reset();
…some more work…
elapsed_seconds = Timer.seconds();
```

Kokkos

A select set of basic Kokkos features have been displayed in these slides. For more details please see the Kokkos *Programming Guide* or take a look at the actual *source code*

Hands On: Base

- *Implement a Kokkos version of the Mandelbrot set using the escape time algorithm*
- *Incrementally add Distance Estimator, Color, and Smoothed color to the application. Ensure correct behavior at each step.*
- *Add anti-aliasing(at least 4x4 subsampling)*
- *Generate benchmarks against the GPU and CPU (Serial/OpenMP backends)*
- *Determine the FLOP/s used by your application*

Hands On: Optional

- *Add ability to create an animation by producing a series of images zooming in on the specified center point*
  - *Hint: Use the ImageMagick"convert" command to create an animated gif from the individual frames*
- *Add MPI to perform weak scaling of animation frames*
- *Benchmark Kokkos::complex against hand rolled complex(splitting real and complex components into scalars)*
- *Investigate compiler flags that may improve performance*
- *Investigate strong scaling(multiple MPI ranks per image)*

Thank you!

Questions?

Contact the OLCF at:

help@olcf.ornl.gov

(865) 241 - 6536

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.