A FIRST INTRODUCTION

CUDA Advanced GPU Courses

June 5, 2024 | Jayesh Badwaik | Jülich Supercomputing Centre

WHAT IS KOKKOS?

- Single Source Implementation using C++
- Descriptive Programming Model
- Aligns with C++ Standard Development on Parallel STL
- **Goal:** Single implementation
  - Compiles and runs on multiple architectures
  - Performant memory access patterns across architectures
  - Leverage architecture-specific features as possible

HISTORY AND SUPPORT

- Established project since 2012
- Used majorly in large number of HPC projects
- Support for most major HPC platforms
- Feedback loop with C++ Standards
  - Parallel STL
  - `std::atomic_ref`
  - `std::mdspan` and `std::mdarray`
- `https://github.com/kokkos`
  - Primary Github Organization
- `https://kokkosteam.slack.com`
  - Slack Channel for Kokkos

OUTLINE

FIRST LOOK

Setup your Environment

- `source $PROJECT/kokkos/setup.sh`
- `rsync -aAX $PROJECT/kokkos/tasks <destination-path>`
- `cd <destination-path>`
- `make t00`

```
$ source $PROJECT/kokkos/setup.sh
$ rsync -aAX $PROJECT/kokkos/tasks <destination-path>
$ cd <destination-path>
$ make t00
# ... some slurm output
Hello, World!
The default execution space is N6Kokkos4CudaE
```


JÜLICH Forschungszentrum

FIRST LOOK

Building Kokkos

- Kokkos –> specific to / optimized for the target architecture
- One GPU, one parallel CPU and one serial CPU backend simultaneously
  - GPU Backends: CUDA, HIP, ...
  - CPU Parallel Backends: OpenMP, Threads
  - CPU Serial Backend: Serial
- Beneficial to know the build of installed Kokkos library

```
cmake -B build -S kokkos \
  -DKokkos_ENABLE_OPENMP=ON -DKokkos_ENABLE_CUDA=On -DKokkos_ENABLE_SERIAL=On \
  -DKokkos_ENABLE_HWLOC=On -DKokkos_ARCH_AMPERE80=On \
  -DCMAKE_CXX_FLAGS="-ccbin g++" \
  -DCMAKE_CXX_COMPILER=$(realpath kokkos/bin/nvcc_wrapper)
```

FIRST LOOK

CMake Configuration

- CMake Preset

```
# In `tasks` directory
cmake --preset training
cmake --build build --target t00
./exrun build/t00
./exrun.nsys build/t00
```

- Make and Run the Target

```
make t00
```

- Run with nsys profiler

```
make t00.nsys
```

FIRST LOOK

Hello, World!

```
$ make t01
```

```cpp
struct functor {
  __host__ __device__
  void operator()(const int i) const {
    Kokkos::printf("Hello from i = %i\n", i);
  }
};

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);

  printf("Hello World on Kokkos execution space %s\n",
  typeid(Kokkos::DefaultExecutionSpace).name());

  Kokkos::parallel_for("HelloWorld", 15, functor());

  Kokkos::finalize();
}
```

```
Hello World
Kokkos execution space N6Kokkos4CudaE
Hello from i = 0
Hello from i = 1
Hello from i = 2
Hello from i = 3
Hello from i = 4
Hello from i = 5
Hello from i = 6
Hello from i = 7
Hello from i = 8
Hello from i = 9
Hello from i = 10
Hello from i = 11
Hello from i = 12
Hello from i = 13
Hello from i = 14
```

FIRST LOOK

Hello, World!

- The Call Invocation

```
Kokkos::parallel_for("HelloWorld", 15, hello_world());
```

Pattern: `parallel_for`, Policy: 15 threads Body: `hello_world`

- The Body (Functor of Function Object)

```
struct hello_world {
  __host__ __device__
  void operator()(const int i) const {
    Kokkos::printf("Hello from i = %i\n", i);
  }
};
```

- **Task 02** : Replace function object with a lambda function

FIRST LOOK

Execution Policies

```
Kokkos::parallel_for("HelloWorld", 15, hello_world());
```

FIRST LOOK

Execution Policies

```
   Kokkos::parallel_for("HelloWorld", 15, hello_world());
```

```
template<class ExecPol, class FType>
parallel_for(const std::string &name, const ExecPol &policy, const FType &functor);
```

FIRST LOOK

Execution Policies

```
Kokkos::parallel_for("HelloWorld", 15, hello_world());
```

```
template<class ExecPol, class FType>
parallel_for(const std::string &name, const ExecPol &policy, const FType &functor);
```

- Execution Policies –> Control how the code runs
  - IntegerType –> `RangePolicy(0, n)`
  - RangePolicy –> 1D Range
  - MDRangePolicy –> Multidimensional Range
- Execution Spaces –> Control where the code runs
- Memory Space –> Control where the data resides (Views)

FIRST LOOK

Execution Spaces

A homogeneous set of cores and an execution mechanism

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


Kokkos::parallel_for("Hello", 15,
  [=] ( const int64_t Index ){
  // kernel code
  }
);
```

- Host code –> Host process
- Parallel Code –> Specified Execution Space
  - DefaultExecutionSpace (set at compilation)
  - Execution Space in Policy

FIRST LOOK

Changing the Execution Space

```
double a[15] = {...};
struct logistic_map {
KOKKOS_INLINE_FUNCTION
  void operator()(const int i) const {
   double value = a[i];
   for (std::size_t i=0; i < 1'000'000; ++i){
     value = 4 * value * (1 - value);
   }
  }
};

Kokkos::parallel_for("LogisticMap", 15, logistic_map());
```

FIRST LOOK

Changing the Execution Space

```
Kokkos::parallel_for("LogisticMap", 15, logistic_map());
```

FIRST LOOK

Changing the Execution Space

```
Kokkos::parallel_for("LogisticMap", 15, logistic_map());

Kokkos::parallel_for("LogisticMap", RangePolicy<>(0,15), logistic_map());
```

FIRST LOOK

Changing the Execution Space

```
Kokkos::parallel_for("LogisticMap", 15, logistic_map());

Kokkos::parallel_for("LogisticMap", RangePolicy<>(0,15), logistic_map());

Kokkos::parallel_for(
 "LogisticMap", RangePolicy<Kokkos::DefaultExecutionSpace>(0,15), logistic_map());
```

FIRST LOOK

Changing the Execution Space

```
Kokkos::parallel_for("LogisticMap", 15, logistic_map());

Kokkos::parallel_for("LogisticMap", RangePolicy<>(0,15), logistic_map());

Kokkos::parallel_for(
 "LogisticMap", RangePolicy<Kokkos::DefaultExecutionSpace>(0,15), logistic_map());
```

Execution Spaces

- `DefaultExecutionSpace`
- `Serial`
- `OpenMP`
- `Cuda`

Task 03 : Run the code on Serial and OpenMP Execution Spaces

SIMPLE REDUCE

The Problem

$$S = \sum_{i=1}^{n} i^2 \qquad (1)$$

OpenMP Version

```
std::atomic_int sum(0);
#pragma omp parallel for
for (int i=0; i<n; i++){
  sum += i * i;
}
```

STL Serial Version

```
auto view = std::ranges::views::iota(0, n);
std::reduce(std::begin(view), std::end(view), 0,
  [](int a, int b){ return a + b*b; });
```

SIMPLE REDUCE

Kokkos Version

- The Function Object

```
struct squaresum {
  using value_type = int;
  KOKKOS_INLINE_FUNCTION
  void operator()(const int i, int& lsum) const {
    lsum += i * i;
  }
};
```

- `Kokkos::parallel_reduce(n, squaresum(), sum);`
- `lsum` : **thread-local** variable used internally by Kokkos
- **Task 04** : Try different execution spaces for the reduce operation
  - `Kokkos::Serial`
  - `Kokkos::OpenMP`
  - `Kokkos::Cuda`

SUMMARY

- Pattern, Policy, Body Paradigm
- `Kokkos::parallel_for`
- Execution Spaces
- `Kokkos::parallel_reduce`

VIEWS

Motivation

- A lightweight copyable C++ templated class
- Datatype for multidimensional array
- No allocations except when explicitly specified
- Automatic deallocation by reference counting

```
View <double *, ... > x (...);
View <double *, ... > y (...);
... populate x , y ...
parallel_for ( " DAXPY " ,N , [=] ( const int64_t i ) {
// Views x and y are captured by value ( copy )
y ( i ) = a * x ( i ) + y ( i );
});
```

VIEWS

Design

- Multi-dimensional array of 0 or more dimensions
- Number of dimensions (rank) fixed at compile time
- Rectangular arrays only
- Dimension size itself can either be compile-time or runtime
- Access via `(...)` operator

```
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
// Access
data(i,j,k) = 0.0;
```

VIEWS

Life Cycle

- Allocations only happen when explicitly specified
  - **no hidden allocations**
- Copy and assignment are **shallow**

```
View < double *[5] > a ( "a" , N );
View < double *[5] > b ( "b" , K );
a = b;
View < double ** > c ( b );
a (0 ,2) = 1;
b (0 ,2) = 2;
c (0 ,2) = 3;
print_value ( a (0 ,2) );
```

VIEWS

Properties

- View's size –> `extent(dim)`
- Static extents -> `static_extent(dim)`
- Raw pointer via `data()` function
- Label via `label()` function
- Deep Copy via `deep_copy()`

```
View < double *[5] > a ( " A " , N0 );
assert (a.extent(0)==N0);
assert (a.extent(1)==5);
static_assert (a.static_extent(1) == 5);
assert (a.data() != nullptr);
assert (a.label() == "A" );
```

VIEWS

Simple View Example

```
using view_type = Kokkos::View<double*[3]>;

struct InitView {
  view_type a;

  InitView(view_type a_) : a(a_) {}

  KOKKOS_INLINE_FUNCTION
  void operator()(const int i) const {
    a(i, 0) = 1.0 * i;
    a(i, 1) = 1.0 * i * i;
    a(i, 2) = 1.0 * i * i * i;
  }
};

Kokkos::parallel_for(N, InitView(a));
```

- **Task 05** : Compute parallel reduction of the elements of the view

MEMORY SPACES

Reading from a File

- Access data from both CPU and GPU

```
View < double * > data ( " data " , size );
for ( int64_t i = 0; i < size ; ++ i ) {
  data ( i ) = ... read from file ...
}
double sum = 0;
Kokkos::parallel_reduce("Label", RangePolicy<ExecSpace>(0, size),
[=](const int64_t index , double &updateval ) {
  valueToUpdate += data ( index );
  }
},
sum );
```

MEMORY SPACES

In Disguise Till Now

```
int main(int argc, char* argv[]) {
 Kokkos::initialize(argc, argv);
 {
  const int N = 10;

  Kokkos::View<double * [3]> a("A", N);

  Kokkos::parallel_for(N, InitView(a));
 }
 Kokkos::finalize();
}
```

MEMORY SPACES

Motivation

```
int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  {
    const int N = 10;

    Kokkos::View<double * [3], DefaultMemorySpace> a("A", N);

    Kokkos::parallel_for(RangePolicy<DefaultExecutionSpace>(0,N), InitView(a));
  }
  Kokkos::finalize();
}
```

MEMORY SPACES

Design

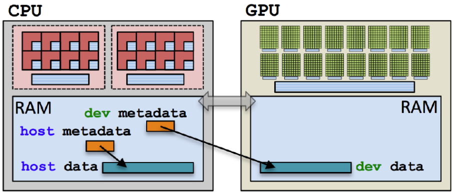
- `View<double***,Memory Space> data(...);`
- Available memory spaces: HostSpace, CudaSpace, CudaUVMSpace
- Default memory space associated with the execution space

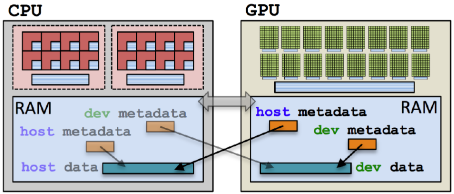
```
// Equivalent :
View < double* > a ( " A " ,N );
View < double*, DefaultExecutionSpace::memory_space>b ( " B " ,N );
```

MEMORY SPACES

Host Space vs Cuda Space

```
View<double**, HostSpace> host(...);
View<double**, CudaSpace> dev(...);
```



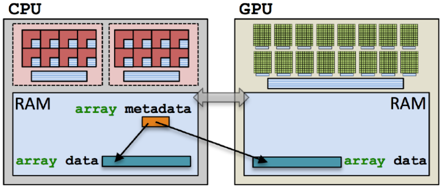


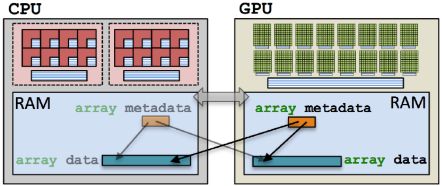
MEMORY SPACES

UVM Space

```
View<double**, UVM Space> array(...);
```

- Runtime Handles Data Copy
- Potential Performance Hit



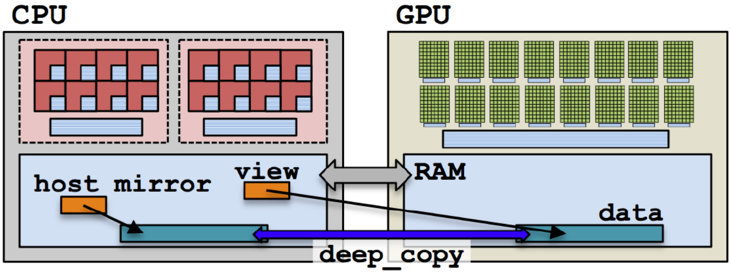


MEMORY SPACES

Mirroring

```
using view_type = Kokkos::View<double**, Space>;
view_type view (...);
view_type::HostMirror hostView_1 = Kokkos::create_mirror_view(view);
view_type::HostMirror hostView_2 = Kokkos::create_mirror(view); //always copy
```

No automatic synchronization



MEMORY SPACES

Mirror Pattern

- Create a `view`'s array in target memory space

```
using view_type = Kokkos :: View < double * , Space >;
view_type view (...);
```

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

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

- Populate `hostView` on the host
- Deep Copy the hostView back to the view

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

- Launch Kernel
- Deep Copy Back

MEMORY SPACES

`mirror` vs `mirror_view`

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

- `create_mirror_view` only allocates data if the destination space cannot access `view`'s data, otherwise it is a shallow copy
- `create_mirror` always allocates data
- Reminder: no hidden deep copy `deep_copy`

Task 06

- Initialize a view in Host Space
- Create a mirror view in Cuda Space
- Run the kernel in Cuda Space
- Copy the data back to Host Space

MEMORY SPACES

CudaUVMSpace

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

Task07 : Run the UVM Example

- Add a second invocation to test performance on already synced data

VIEWS

SubViews

- Call a kernel on a subset of the view
- `Kokkos::subview` –> slice of a view
- No extra memory allocation or copying

```
Kokkos::View<double***> v("v", 10, 10, 10);
auto i0 = Kokkos::pair(0, 5);
auto slice = Kokkos::subview(v, i0, Kokkos::ALL(), Kokkos::ALL());
```

- Use `auto` for the type of the subview
- Return type is implementation-defined for performance reasons

Task08 : Create a subview of a view and run a kernel on the subview

SUMMARY

Views and Spaces

- A lightweight copyable C++ templated class
- Datatype for multidimensional array
- No allocations except when explicitly specified
- Automatic deallocation by reference counting
- MemorySpaces control where the data resides
- By default, the memory space is associated with the default execution space
- Mirror and Mirror View for creating copies of views
- `deep_copy` for synchronization
- Subviews for slicing views

PROFILING

Regions

- Similar to NVTX Ranges

```
Kokkos::Profiling::ProfilingSection section("label");
section.start();
...
section.stop();
```

- RAII-like Behavior

```
void do_work_v2() {
  Kokkos::Profiling::ScopedRegion region("label");
  // <code>
  if (cond) return;
  // <more code>
}
```

MULTIDIMENSIONAL LOOPS

Consider nested loops:

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

Current Kokkos knowledge :

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

- Parallelization only of outer loop
- Only $N0 \times N1 \times N2$ iterations might be worth parallelizing

MULTIDIMENSIONAL LOOPS

MDRangePolicy

```
parallel_for("mdloop", Kokkos::MDRangePolicy<Rank<3>>({0, 0, 0} ,{ N0, N1, N2}),
KOKKOS_LAMBDA ( int64_t i , int64_t j , int64_t k ) {
  some_function(i, j, k);
});
```

- Dimensionality of the loop –> `Rank<3>`
- Only rectangular iteration spaces
- Provide Begin and End of the iteration space

```
parallel_reduce("mdloop", MDRangePolicy<Rank<3>>({0, 0, 0} ,{ N0, N1, N2}),
KOKKOS_LAMBDA ( int64_t i , int64_t j , int64_t k, double& lsum) {
  lsum += some_function(i, j, k);
});
```

MULTIDIMENSIONAL LOOPS

Tiling

```
parallel_for("mdloop",
  Kokkos::MDRangePolicy<Rank<3>>({0, 0, 0} ,{ N0, N1, N2}, {T0, T1, T2}),
KOKKOS_LAMBDA ( int64_t i , int64_t j , int64_t k ) {
  some_function(i, j, k);
});
```

- Tiling strategy as third argument
- For GPUs, a tile is handled by a single thread block
- Too large tile sizes will fail!

HANDS-ON EXERCISE

RangePolicy vs MDRangePolicy

Start with `h02.cpp`

- Create a View in the UVM Space
- Initialize the data on the device with the formula using multidimensional loops

$$a(i,j,k) = i * j * k \tag{2}$$

- Initialize the data on the host with the formula using multidimensional loops

$$a(i,j,k) = i + j + k \tag{3}$$

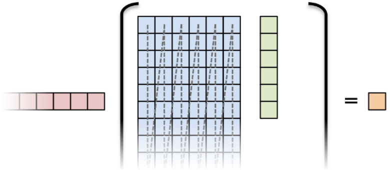
ADVANCED VIEWS

Layouts

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

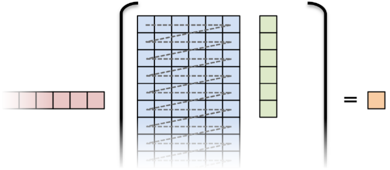
LayoutLeft

in 2D, “column-major”



LayoutRight

in 2D, “row-major”



ADVANCED VIEWS

Unmanaged Views

- Interacting with already allocated memory

```
std::vector<double> input = <somelib>::get_3d_data("file");
```

- Views can simply wrap existing allocations as unmanaged views

```
Kokkos::View<double**, LayoutRight, Kokkos::HostSpace>
 view(input.data(), input.size(), 3);

Kokkos::View<double*[3], LayoutRight, Kokkos::HostSpace>
 view(input.data(), input.size());
```

- Pointer to raw data as first argument
- All runtime dimensions as next arguments
- Layout and MemorySpace should match
- No label can be provided

JÜLICH Forschungszentrum

ADVANCED VIEWS

Unmanaged Views

- Copying to dvice

```
using device_space = Kokkos::Cuda::memory_space;
auto a = Kokkos::create_mirror_view(device_space(), view);
Kokkos::deep_copy(a, view);
```

- Shortcut

```
auto a = Kokkos::create_mirror_view(Kokkos::Cuda::memory_space(), view);
```

- Wrap existing allocation
  - No reference counting
  - No deallocation after going out of scope
  - No checks for memory spaces

ADVANCED VIEWS

Unmanaged Views

Shape Punning

```
double* boundary_data;
cudaMalloc(&data, N *M * sizeof(double));
View<double**> use_in_kernel(data, N, M);
View<double**> halo_transmission(data, nboundary, M * N / nboundary);
```

ADVANCED VIEWS

Dual Views

- Help transition codes to Kokkos
- When converting CPU code to GPU code
  - Generally, no holistic view of data transfers
  - Pre-emptively moving data to GPU is expensive
  - Removing pre-emptive moves is bug-prone
  - Changing code in one part might invalidate a data transfer
- DualView bundle
  - Two views -> one for host, one for device
  - Mark data as modified after being written to
  - Mark data as ”needed to be retrieved” before read from
  - Kokkos will handle the actual data movement as needed

ADVANCED VIEWS

Dual Views

- Data members for the two views
  - `DualView::t_host host_view`
  - `DualView::t_dev dev_view`
- Retrieve data members
  - `t_host view_host();`
  - `t_dev view_dev();`
- Mark view as modified
  - `void modify_host();`
  - `void modify_device();`

Thank You