Kokkos, Modern c++ and performance portability

Presentation · January 2017

DOI: 10.13140/RG.2.2.26312.11523

Pierre Kestener
Atomic Energy and Alternative Energies Commission
71 PUBLICATIONS **793** CITATIONS

Kokkos, Modern C++, performance portability, ...

Pierre Kestener[1]

[1]CEA Saclay, DSM, Maison de la Simulation

PATC, January, 16-18th, 2017

Schedule

- **Monday + Tuesday morning:** NVidia OpenACC tutorial
- **Tuesday afternoon + Wednesday:** Kokkos tutorial
  - Introduction performance portability
  - IBM Power8 + Nvidia Pascal P100 platform : short overview
  - **Kokkos: features overview**
  - Hands-on 0: **retrieve Kokkos sources**, how to build, how to run a *helloworld* application, explore different configurations
  - Hands-on 1: cross-checking **Kokkos + hwloc** is OK
  - **Replay some tutorial slides from SC2016 for deeper Kokkos concepts**
  - Hands-On 2: Simple example **SAXPY**
    ⇒ simplest computing kernel in Kokkos
  - Hands-On 3: Simple example **Mandelbrot set**
    ⇒ 1D Kokkos::View + linearized index (+ asynchronous execution)
  - Hands-On 4: Simple examples **Stencil + Finite Difference**
    ⇒ 2D Kokkos::View
  - Hands-On 5: **Laplace exercice**
    ⇒ pure Kokkos versus Kokkos + MPI + hwloc (multiGPU)
  - Hands-On 6: CSCS miniApp: **Fisher equation solver**
    ⇒ use Kokkos lambda
  - Hands-On 7: CFD miniApp: **Euler solver**
    ⇒ performance measurement for several Kokkos backends (OpenMP, CUDA)

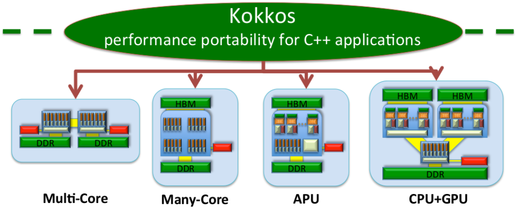
IBM Power8 / Nvidia Pascal P100

- **Kokkos training material archive (last up-to-date version) is on `ouessant`:**
  `/pwrwork/workshops/patc-201701/kokkos/training_kokkos.tar.gz`
- Use material from IBM/NVidia [1], gives detail on the platform
  See file: `doc/ouessant/Introduction.pdf` in archive
- Minimal information about software environment, how to build and run an application, submit a job on machine `ouessant`
  See file:
  `doc/ouessant/Ouessant-Application_User_Guide-16-12-1.pdf`

[1] Thanks to Nicolas Tallet (IBM)

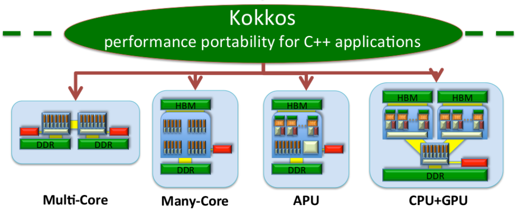
Kokkos: a programming model for performance portability

- **Kokkos** is a **C++ library** with **parallel algorithmic patterns** AND **data containers** for **node-level parallelism**.
- Implementation relies heavily on **meta-programing** to derive native **low-level code (OpenMP, Pthreads, CUDA, ...)** and adapt data structure **memory layout** at compile-time
- Core developers at **SANDIA NL** (**H.C. Edwards, C. Trott**)

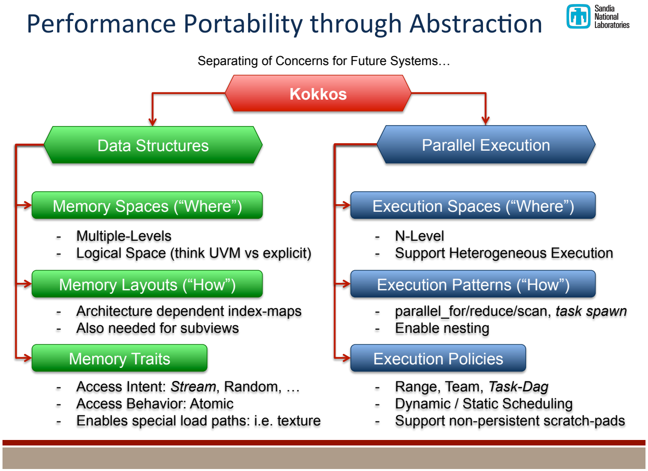


Kokkos: a programming model for performance portability

- **Open source**, `https://github.com/kokkos/kokkos`
- Primarily developped as a base building layer for **generic high-performance parallel linear algebra** in Trilinos
- Also used in molecular dynamics code, e.g. LAMMPS
- Goal: **ISO/C++ 2020 Standard** subsumes Kokkos abstractions



Kokkos: a programming model for performance portability



Performance Portability through Abstraction

Separating of Concerns for Future Systems…

Sandia National Laboratories
Kokkos
Data Structures
Parallel Execution
Memory Spaces (“Where”)
- Multiple-Levels
- Logical Space (think UVM vs explicit)
Memory Layouts (“How”)
- Architecture dependent index-maps
- Also needed for subviews
Memory Traits
- Access Intent: *Stream*, Random, …
- Access Behavior: Atomic
- Enables special load paths: i.e. texture
Execution Spaces (“Where”)
- N-Level
- Support Heterogeneous Execution
Execution Patterns (“How”)
- parallel_for/reduce/scan, *task spawn*
- Enable nesting
Execution Policies
- Range, Team, *Task-Dag*
- Dynamic / Static Scheduling
- Support non-persistent scratch-pads

reference:
https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/Kokkos-Multi-CoE.pdf

Kokkos Concepts (1) - the abstract machine model

- Kokkos defines an abstract machine model for future large shared-memory nodes made of
  - **latency-oriented cores** (contemporary CPU core)
  - **throughput-oriented cores** (GPU, ...)

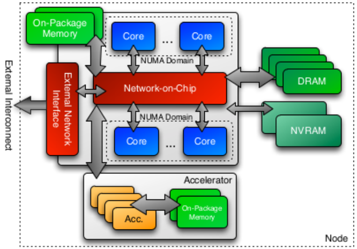


Figure: Conceptual model of a future HPC node. (Kokkos User's Guide).

Kokkos Concepts (2) - What is a device ?

- A **Kokkos device**:
- From a C++ API design point of view, Kokkos defines several c++ class for a device in `core/src`, e.g.
  - Kokkos::Cuda, Kokkos::OpenMP, Kokkos::Pthreads, Kokkos::Serial
  - *device* = execution space + memory space
- Each *Kokkos device* pre-defines some types
- Example **Kokkos device** (not required for a user, only Kokkos developper), e.g.

```
class Cuda {
  public:
  // Tag this class as a kokkos execution space
  typedef Cuda                    execution_space ;

  #if defined( KOKKOS_USE_CUDA_UVM )
  // This execution space's preferred memory space.
  typedef CudaUVMSpace            memory_space ;
  #else
  // This execution space's preferred memory space.
  typedef CudaSpace               memory_space ;
  #endif

  // This execution space preferred device_type
  typedef Kokkos::Device<execution_space,memory_space> device_type;

  // The size_type best suited for this execution space.
  typedef memory_space::size_type  size_type ;

  // This execution space's preferred array layout.
  typedef LayoutLeft              array_layout ;
  ...
} // end class Cuda
```

Kokkos Concepts (3) - execution space, memory space

- **Execution space:** Where should a parallel contruct (`parallel_for`, `parallel_reduce`, ...) be executed
  - Special case: `class HostSpace`, special device (always defined) where execution space is either (Serial, Pthread or OpenMP).
  - Each execution space is equipped with a `fence`: `Kokkos::Cuda::fence()`
- **Memory space:** Where / how data are allocated in memory (HostSpace, CudaSpace, CudaUVMSpace, CudaHostPinnedSpace, HBWSpace, ...)
- **Memory layout** (come back later on that)
- Other concepts:
  - Execution policy: used to modify a parallel thread dispatch
- Multiple execution / memory space can be used in a single application
  See for example in Kokkos sources
  `example/tutorial/Advanced_View/07_Overlapping_DeepCopy`
  Though, take care that currently, Cuda stream are not completely mapped into Kokkos API [2]; meanwhile Cuda streams can be used directly (but looses portability);

[2] Will be implemented in the coming months

Hands-On 0: Build kokkos

- **Kokkos is still experimental, but moving fast**
- **1. Get Kokkos sources, development branch**
  - Practicals on `ouessant:`

    `1. mkdir $HOME/kokkos-tutorial; cd $HOME/kokkos-tutorial`

    some kokkos tutorial examples have a Makefile configured for using that precise location.

    `2. git clone https://github.com/kokkos/kokkos`

    `3. cd kokkos; git checkout develop`
- **2. Build configuration**
  - About build system, several ways to use/build Kokkos
    1. CMake: only when Kokkos is build inside Trilinos,
    2. **Regular standalone utilization (installed Kokkos):** use `generate_makefile.bash`, then `make kokkoslib; make install`
       Then use a *modulefile* to configure the environment
    3. **Embedded Kokkos source files in your application** - mostly usefull in tutorial.
  - We will use 2. and 3.

Hands-On 0: Build kokkos (2)

About Makefile variable for building on multiple architectures

- The following variables are usefull when building some of the tutorial examples :
  - `KOKKOS_PATH`: path to Kokkos source dir
  - `KOKKOS_DEVICES`: define possible execution spaces: CUDA, OpenMP, Pthreads, Serial, ...
  - `KOKKOS_ARCH`: used to customize compiler flags; e.g. Power8, Kepler35, SNB, KNL, ARMv80, ...
- When building for CUDA device, you'll need to use Kokkos' own compiler wrapper: `nvcc_wrapper` (included in Kokkos sources)
- When building Kokkos and aiming at an installed Kokkos, the same information (in a different form) is passed to script `generate_makefile.bash`
  Just type `./generate_makefile.bash -help` at top-level Kokkos sources
- When using Kokkos embedded in your application, these variables must be set on the `make` command line.

Hands-On 0: Build kokkos (3)

- **Example build configurations (for an installed Kokkos)**
  - For `ouessant`, see file `doc/readme_build_kokkos_ouessant` in the provided archive
  - Serial (mostly for testing)
    ```
    ../generate_makefile.bash -with-serial
    -prefix=$HOME/local/kokkos_serial
    ```
  - **OpenMP**
    ```
    ../generate_makefile.bash -with-openmp
    -prefix=$HOME/local/kokkos_openmp_dev
    ```
  - **CUDA (+ OpenMP)**; typical configuration
    ```
    ../generate_makefile.bash -with-cuda -arch=Pascal60
    -prefix=$HOME/local/kokkos_cuda_lambda_openmp
    -with-cuda-options=enable_lambda -with-openmp -with-hwloc=/usr
    ```
- **After installation** (`make kokkoslib; make install;`) the file `Makefile.kokkos` is created, and designed to be reused in your application build system.
- **2 choices for integrating Kokkos in your app:**
  - Use an existing Makefile from Kokkos tutorial, examples, ...
  - Use your own build system: there can be a quite large combinatorics of `DEVICES`, `ARCH`, compilers, compiler options, ...

Kokkos - Documentation

- PDF documentation in kokkos source tree : `doc/Kokkos_PG.pdf` (programming guide)
- Doxygen can only be built from inside Trilinos source tree
  Version of the day can be browsed at
  `https://trilinos.org/docs/dev/packages/kokkos/doc/html/index.html`
- Kokkos source code itself, reading unit tests code is also helpful

Additionnal resources:

- Tutorial slides and codes:
  `https://github.com/kokkos/kokkos-tutorials`

Kokkos - initialize / finalize

- `Kokkos::initialize / finalize`

```
#include <Kokkos_Macros.hpp>
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  // default: initialize the host exec space
  // What exactly gets initialized depends on how kokkos
  // was built, i.e. which options was passed to
  // generate_makefile.bash
  Kokkos::initialize();
  ...
  Kokkos::finalize();
}
```

- **What's happening inside `Kokkos::initialize`**
  - Defines `Default Device / DefaultExecutionSpace Default memory space` (as specified when kokkos itself was built, by order of priority: Cuda > OpenMP > Pthreads > Serial)
    e.g. if `--with-cuda` was not pass to `generate_makefile.bash`, but `--with-openmp` was, then `DefaultExecutionSpace` is OpenMP
  - You can activate several execution spaces (recommended)
  - all this information provided at compile time will internally be used inside Kokkos sources as default (hidden) template parameters

Kokkos - initialize / finalize

- `Kokkos::initialize / finalize` (most of the time OK)

```
#include <Kokkos_Macros.hpp>
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  // default: initialize the host exec space
  // What exactly gets initialized depends on how kokkos
  // was built, i.e. which options was passed to
  // generate_makefile.bash
  Kokkos::initialize();
  ...
  Kokkos::finalize();
}
```

- **Fine control of initialization:**
  - `Kokkos::initialize(argc, argv);`
    User can change/fix e.g. number OpenMP threads on the application's command line
  - This is regular initialization. If available `hwloc` is used to provide default hardware locality:
    - For OpenMP exec space: number of threads (default is all CPU cores)
      NB: usual environment variables (e.g. `OMP_NUM_THREADS`, `GOMP_CPU_AFFINITY` can (of course) also be used
    - Mapping between GPUs and MPI task

Kokkos - initialize / finalize

- **Advanced initialization** with **OpenMP + CUDA**
  Needed/usefull to be able to execution computation on both HOST / GPU

```
#if defined( KOKKOS_HAVE_CUDA )
Kokkos::HostSpace::execution_space::initialize(teams*num_threads);
Kokkos::Cuda::SelectDevice select_device(device);
Kokkos::Cuda::initialize(select_device);
#elif defined( KOKKOS_HAVE_OPENMP )
Kokkos::OpenMP::initialize(teams*num_threads);
#elif defined( KOKKOS_HAVE_PTHREAD )
Kokkos::Threads::initialize(teams*num_threads);
#endif
```

Kokkos - initialize / finalize with MPI

- **Advanced initialization** with **MPI + Kokkos/CUDA version 1 : implicit mapping**
  Don't do anything special, let Kokkos through hwloc chose the GPU

```
// Just checking how Kokkos+hwloc performed
// the MPI rank - GPU mapping
int cudaDeviceId;
cudaGetDevice(&cudaDeviceId);
std::cout << "I'm MPI task #" << rank << " pinned to GPU #" << cudaDeviceId << "\n";
```

- **Advanced initialization** with **MPI + Kokkos/CUDA version 2 : explicit mapping** (we will come back into that with example code)

```
// MPI initialized above

// probe the number of CUDA device (i.e. GPUs)
const int ngpu = Kokkos::Cuda::detect_device_count();

// provide a mapping 1 MPI task <-> 1 GPU
const int cuda_device_rank = pre_mpi_local_rank % ngpu ;

// each MPI task initialize the selected device id
Kokkos::Cuda::initialize(
Kokkos::Cuda::SelectDevice( cuda_device_rank ) );
```

- In any case, cross-check this information with the job scheduler, e.g. `mpirun --report-bindings`

Hands-On 1 : query_device

Purpose: just cross-checking Kokkos/Hwloc is working OK

- We will first re-use material from Kokkos github repository.
- On your home, on `ouessant`:
  1. `mkdir kokkos-tutorial; cd kokkos-tutorial`
  2. `git clone https://github.com/kokkos/kokkos.git`
     # **Don't try to build kokkos here (for now)**

Hands-On 1 : query_device

Purpose: just cross-checking Kokkos/Hwloc is working OK

- Kokkos sources will be built by the application Makefile
- `cd $HOME/kokkos-tutorial/kokkos/example/query_device`
- open `query_device.cpp`; no computations, it just prints hardware information
- 1. **Default serial build (with hwloc):** `make KOKKOS_USE_TPLS="hwloc"`
     How many NUMA / Cores / Hyperthreads on power8 CPU ?
     What is the current SMT mode on a ouessant login node ? (use command `ppc64_cpu --smt` or `ppc64_cpu -info`)
  2. **OpenMP build (with hwloc):** `make KOKKOS_USE_TPLS="hwloc" KOKKOS_DEVICES=OpenMP` (off course, exact same information obtained)
  3. **CUDA/OpenMP build (with hwloc):** `make KOKKOS_USE_TPLS="hwloc" KOKKOS_DEVICES=Cuda,OpenMP`; rerun and you should get information about the CPU+GPU configuration
- **Take some time to have a look at Makefile.**
  Note that latter when using an installed kokkos library, we won't need to set architecture or device related variables on the command line .

Hands-On 1 : query_device

Purpose: just cross-checking Kokkos/Hwloc is working OK

- **What happens if hwloc is not activated ?**
- Edit file `query_device.cpp` and do the following modification:
  1. Add `Kokkos::initialize(argc, argv);` after `MPI_Init`
  2. Add `Kokkos::finalize();` before `MPI_Finalize`
  3. change

```
#if defined( KOKKOS_HAVE_CUDA )
  Kokkos::Cuda::print_configuration( msg );
#else
  Kokkos::OpenMP::print_configuration( msg );
#endif
```

- Rebuild 1 without HWLOC: `make KOKKOS_DEVICES=OpenMP`

```
Kokkos::OpenMP KOKKOS_HAVE_OPENMP thread_pool_topology[ 1 x 80 x 1
```

- Rebuild 2 with HWLOC: `make KOKKOS_DEVICES=OpenMP KOKKOS_USE_TPLS="hwloc"`

```
hwloc( NUMA[2] x CORE[10] x HT[4] )
Kokkos::OpenMP KOKKOS_HAVE_OPENMP hwloc[2x10x4] hwloc_binding_enabl
```

- As already said: processor affinity is crucial to performance

Kokkos data Container (1)

`Kokkos::View<...>` is **multidimensionnal data container** with **hardware adapted memory layout**

- `Kokkos::View<double **> data("data",NX,NY);` : 2D array with sizes known at runtime
- `Kokkos::View<double *[3]> data("data",NX);` : 2D array with first size known at runtime (NX), and second known at compile time (3).
- How do I access data ? $data(i,j)$!
- Which memory space ? By default, the default device memory space !
  Want to enforce in which memory space lives the view ? `Kokkos::View<..., Device>`: if a second template parameter is given, Kokkos expects a `Device` (e.g. `Kokkos::OpenMP`, `Kokkos::Cuda`, ...)
- `Kokkos::View` are **small**, designed as reference to allocated memory buffer
  - View = pointer to data + array shape
  - assignment is fast (shallow copy + increment ref counter) [3]
- `Kokkos::View` are designed to be pass by value to a function.

[3]NB: same behaviour as in python for example

Kokkos data Container (2)

- Concept of **memory layout:**
- **Memory layout is crucial for performance:**
  - **LayoutLeft**: $data(i, j, k)$ uses linearized index as $i + NX * j + NX * NY * k$ (column-major order)
  - **LayoutRight**: $data(i, j, k)$ uses linearized index as $k + NZ * j + NZ * NY * i$ (raw-major order)
- `Kokkos::View<int**, Kokkos::OpenMP>` defaults with LayoutRight; a single thread access contiguous entries of the array. Better for cache and avoid sharing cache lines between threads.
- `Kokkos::View<int**, Kokkos::Cuda>` defaults LayoutLeft so that consecutive threads in the same warp access consecutive entries in memory; try to ensure memory coalescence constraint
- You can if you like, still enforce memory layout yourself (or just use 1D Views, and compute index yourself);
  We will see the 2 possibilities with the miniApp on the Fisher equation

Kokkos data Container (3)

- `Kokkos::View<...>` are reference-counted
- By default do a **shallow copy**

```
Kokkos::View<int *>("a",10);
Kokkos::View<int *>("b",10);
a = b; // a now points to b (ref counter incremented by 1)
```

- **Deep copy** must by explicit:

```
Kokkos::deep_copy(dest,src);
```

  - **Usefull when copying data from one memory space to another**
    e.g. from HostSpace to CudaSpace
  - When `dest` and `src` are in the same memory space, it does nothing ! (usefull for portability, see example in miniapps later)

Kokkos data Container (4)

- A verbose **Kokkos::View** declaration example:

```
Kokkos::View<double*,Kokkos::LayoutLeft,Kokkos::CudaSpace> a;
```

  - **What ?** a data type
  - **How ?** a memory layout
  - **Where ?** a memory space
  - the last two template parameters are optionnal (have default values)
  - There is actually a 4th template parameter for Memory traits (e.g. atomic access)
- `Kokkos::DualView<...>` : usefull when porting an application incrementally, adata container on two different memory space. see `tutorial/Advanced_Views/04_dualviews/dual_view.cpp`
- `Kokkos::UnorderedMap<...>`
- Can also define **subview (array slicing, no deep copy)**. See exercice about Mandelbrot set.

Kokkos data Container (5)

- **What types of data may a View contain ?**
  C++ Plain Old Data (POD), i.e. basically compatible with C language:
  - Can be allocated with `std::malloc`
  - Can be copied with `std::memmove`
- POD in C++11:
  - a trivial type (no virtual member functions, no virtual base class)
  - a standard layout type
- C++11: How to check if a given class `A` is POD ?

```
#include <type_traits>

class A { ... }
std::cout << "is class A POD ? " << std::is_pod<A>::value << "\n";
```

Kokkos data Container (6)

Interoperability

- **With a legacy API** `void legacyFunction(int * data, int size)`
  how to retrieve a raw pointer from a `Kokkos::View<int *> data`:
  `int *raw_ptr = data.ptr_on_device()`
  This is not recommended. No more reference counting.

Kokkos compute Kernels - parallel dispatch (1)

- **3 types of parallel dispatch**
  - `Kokkos::parallel_for`
  - `Kokkos::parallel_reduce`
  - `Kokkos::parallel_scan`
- A dispatch needs as input
  - **an execution policy:** e.g. a range (can simply be an integer), team of threads, ...
  - **a body:** specified as a lambda function or a functor
- Very important: launching a kernel (thread dispatching) is by default asynchronous

Kokkos compute Kernels - parallel dispatch (2)

How to specify a compute kernel in Kokkos ?

1. **Use Lambda functions.**
   NB: a lambda in c++11 is an unnamed function object capable of capturing variables in scope.
   ```
   Kokkos::parallel_for (100, KOKKOS_LAMBDA (const int i) {
     data(i) = 2*i;
   });
   ```
   Here we do 2 things in 1 step: define the computation body (lambda func) and launch computation.
2. **Use a C++ functor class.**
   A functor is a class containing a function to execute in parallel.
   ```
   class FunctorType {
     public:
     KOKKOS_INLINE_FUNCTION
     void operator() ( const int i ) const ;
   };
   ...
   FunctorType func;
   Kokkos::parallel_for (100, func);
   ```

Note: 100 here is the simplest way to specify an execution policy

Kokkos compute Kernels - parallel dispatch (3)

Notes on macros defined in `core/src/Kokkos_Macros.hpp`

- `KOKKOS_LAMBA` is a macro which provides a compiler-portable way of specifying a lambda function with **capture-by-value closure**.
  - `KOKKOS_LAMBA` must be used at the most outer parallel loop; inside a lambda one can call another lambda
- `KOKKOS_INLINE_FUNCTION void operator() (...) const;`
  this macro helps providing the necessary compiler specific *decorators*, e.g. `__device__` for Cuda to make sure the body can be turns into a Cuda kernel.
  - macro `KOKKOS_INLINE_FUNCTION` must be applied to any function call inside a parallel loop

Kokkos compute Kernels - parallel dispatch (4)

Lambda or Functor: which one to use in Kokkos ? Both !

1. **Use Lambda functions.**
   - easy way for small compute kernels
   - For GPU, requires Cuda 7.5 (8.0 is current and latest CUDA version)
2. **Use a C++ functor class.**
   - More flexible, allow to design more complex kernel

Kokkos compute Kernels - parallel dispatch (5)

About Kokkos::parallel_reduce with lambda

- As for `parallel_for`, loop body can be specified as a lambda, or a functor; here is the lambda way when reduce operation is `sum`:

```
Kokkos::parallel_reduce (100, KOKKOS_LAMBDA (const int i, int &local_sum)
  local_sum += data(i);
}, sum);
```

Kokkos compute Kernels - parallel dispatch (6)

About Kokkos::parallel_reduce functor

- Kokkos supplies a default `init` / `join` operator which is `operator+`
- If the reduce operator is not trivial (i.e. not a sum) ⇒ you need to define methods `init` and `join`

```
class ReduceFunctor {
  public:
  // declare a constructor ...
  KOKKOS_INLINE_FUNCTION void
  operator() (const int i, data_t &update) const {...}

  // How to join/combine intermediate reduce from different threads
  KOKKOS_INLINE_FUNCTION void
  join(volatile data_t &dst, const volatile data_t &src) const {...}

  // how each thread initializes its reduce result
  KOKKOS_INLINE_FUNCTION void
  init(const volatile data_t &dst) const {...}
}
```

Kokkos compute Kernels - parallel dispatch (7)

Parallel dispatch - execution policy

- Remember that an execution policy specifies **how** a parallel dispatch is done by the device
- **Range policy:** from...to
  no prescription of order of execution nor concurrency; allows to adapt to the actual hardware; e.g. a GPU has some level of hardware parallelism (Streaming Multiprocessor) and some levels of concurrency (warps and block of threads).
- **Multidimensional range:** still experimental (as of January 2017), mapping a higher than 1D range of iteration.

```
// create the MDrangePolicy object
using namespace Kokkos::Experimental;
using range_type = MDRangePolicy< Rank<2>, Kokkos::IndexType<int> >;
range_type range( {0,0}, {N0,N1} );

// use a special multidimensional parallel for launcher
md_parallel_for(range, functor);
```

Kokkos compute Kernels - parallel dispatch (8)

Parallel dispatch - execution policy

- **Team policy:** for **hierarchical parallelism**
    - threads team
    - threads inside a team
    - vector lanes

-
  ```
  // Using default execution space and launching
  // a league with league_size teams with team_size threads each
  Kokkos::TeamPolicy <>
  policy( league_size , team_size );
  ```

  equivalent to launching in CUDA a 1D grid of 1D blocks of threads.

  Team scratch pad memory $\Longleftrightarrow$ CUDA shared memory

Kokkos compute Kernels - parallel dispatch (9)

Hierarchical parallelism (advanced)

- OpenMP: League of Teams of Threads
- Cuda: Grid of Blocks of Threads
- Experimental features: task parallelism
  see slides by C. Edwards at GTC2016 2016-04-GTC-Kokkos-Task.pdf

Hands-On 2 : SAXPY

Purpose: The simplest computing kernel in Kokkos, importance of hwloc

- There 5 differents versions
- **1. Serial : no Kokkos)**
- **2. OpenMP : no Kokkos)**
- 3. Kokkos-Lambda-CPU : Kokkos with lambda for threads dispatch
- **4. Kokkos-Lambda : Kokkos with lambda for threads dispatch and data buffer (Kokkos::View)**
- 5. Kokkos-Functor-CPU : Kokkos with functor for threads dispatch only

- **Saxpy serial (reference executable on Power8)**
  - `cd $HOME/kokkos-tutorial/kokkos-tutorials/1-Day-Tutorial/Exercises/01_AXPY/Serial`
  - `make KOKKOS_ARCH=Power8`
  - Alternatively, we could have modify `Makefile` and change `SNB` into `Power8`
- **Saxpy regular OpenMP (on Power8)**
  - `cd $HOME/kokkos-tutorial/kokkos-tutorials/1-Day-Tutorial/Exercises/01_AXPY/OpenMP`
  - Rebuild: `make KOKKOS_ARCH=Power8`; and observe performance

see also slides from SC2016, page 42(74).

Hands-On 2 : SAXPY

- **Saxpy Kokkos OpenMP (on Power8)** [4]
  - `cd $HOME/kokkos-tutorial/kokkos-tutorials/1-Day-Tutorial/Exercises/01_AXPY/Kokkos-Lambda`
  - Add 3 lines in `saxpy.cpp` right after Kokkos initialization

```
std::ostringstream msg;
Kokkos::OpenMP::print_configuration( msg );
std::cout << msg.str();
```

  - `make KOKKOS_ARCH=Power8`
  - Make sure all available CPU cores were used ($1 \times 160 \times 1$)
  - Change the number of OpenMP threads created by kokkos, e.g. : `./saxpy.host -threads=20`
  - Add again `KOKKOS_USE_TPLS="hwloc"` on the command line
    Rebuild and rerun, you should see that application uses **all the available numa domains**, and a strongly increased bandwidth usage !

[4]Make sure to use a very large data array; Power8 has very large cache memory. If you don't, this example will not measure memory bandwith. Maximum bandwidth is 230 GB/s on a 2 socket P8. You should measure around 170 GB/s.

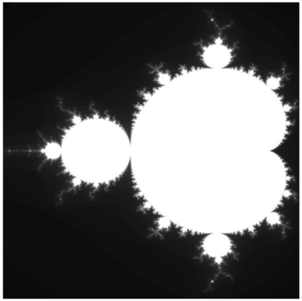
Hands-On 2 : SAXPY

- **Saxpy CUDA (on Power8 + Nvidia K80/P100)**
  - `cd $HOME/kokkos-tutorial/kokkos-tutorials/1-Day-Tutorial/Exercises/01_AXPY/Kokkos-Lambda`
  - `module load cuda/8.0`
- Rebuild for K80, run on ouessant (front node):
  ```
  make KOKKOS_DEVICES="Cuda,OpenMP"
  KOKKOS_ARCH="Kepler37,Power8" KOKKOS_USE_TPLS="hwloc"
  ```
- Rebuild for P100, run on compute node using `submit_ouessant.sh` (should see a strong difference):
  ```
  make KOKKOS_DEVICES="Cuda,OpenMP"
  KOKKOS_ARCH="Pascal60,Power8" KOKKOS_USE_TPLS="hwloc"
  ```
  Please note that **maximun bandwith is 732 GB/s for Pascal P100**, you can retrieve this number by examining `deviceQuery` example in CUDA/SDK.

Hands-On 3 : Mandelbrot set

- **Illustrate Functor class + 1D `Kokkos::View` + linearized index**
- the original serial code use 1D `std::vector<unsigned char>` data with linearized index, i.e. $index = i + Nx * j$
- See serial code from `code/exercises/mandelbrot_kokkos/serial` (also read `main.cpp`)

```
for(int index=0; index<WIDTH*HEIGHT; ++index) {
  int i,j;
  index2coord(index,i,j,WIDTH,HEIGHT);
  image[index]=mandelbrot(i,j);
}
```



Hands-On 3 : Mandelbrot set

Proposed activity:
refactor this computing loop into a C++ Kokkos functor class

- See kokkos basic version from `code/exercises/mandelbrot_kokkos/kokkos_basic` (already a bit refactored to ease the job)

1. we added a file `kokkos_shared.h`: `std::vector` replaced by a `Kokkos::View`
2. **TODO:** fill TODOs in `mandelbrot.h` containing the definition of the c++ mandelbrot kokkos functor.
   Notice: the global constants have disappeared, they are now part of the functor context.
3. **TODO:** refactor `main.cpp` (change the TODO)
   - Modify data allocation (from `std::vector` to `Kokkos::View`); we have now arrays: `image` and `imageHost` (mirror)
   - Copy back results from device to host.

Hands-On 3 : Mandelbrot set

- The provided `Makefile` is designed to be used with kokkos environment from a modulefile
- Build the `kokkos_basic` version
- **OpenMP**
  - `module use /pwrwork/workshops/patc-201701/kokkos/modulefiles`
  - `module load kokkos/openmp_gnu485_dev`
  - `make`
- **Cuda**
  - `module use /pwrwork/workshops/patc-201701/kokkos/modulefiles`
  - `module load cuda/8.0 kokkos/cuda80_gnu485_dev_k80`
  - `make`
- **Compare performance** for a large Mandelbrot set 8192×8192 : OpenMP versus Cuda

Hands-On 3 : Mandelbrot set

- **Additionnal:** revisit this simple example using a **multidimensional range policy** to launch the Mandelbrot functor:

```
Kokkos::Experimental::MDRangePolicy< Kokkos::Experimental::Rank<2>
                                     Kokkos::IndexType<int> >;
```

- **TODO:** fill TODOs in `mandelbrot.h` and `main.cpp` in directory `mandelbrot_kokkos/kokkos_mdrange`
- **This way avoids the use of linearized indexes.**

Hands-On 3 : Mandelbrot set

- Pipelined version of Mandelbrot is not currently fully functional; it requires a small patch applied to `Kokkos` for `cudaStreams`; see `https://github.com/kokkos/kokkos/issues/532`
- Understand what is pipelined version of Mandelbrot see: `http://on-demand.gputechconf.com/gtc/2015/webinar/openacc-course/advanced-openacc-techniques.pdf` It basically consists in overlapping GPU computations with CPU/GPU memory transfert.
- See explanations given during training

Hands-On 4 : Finite Difference / Stencil

- Illustrate the use of 2D `Kokkos::View`

MPI + Kokkos on Ouessant (1)

- Perform **distributed computing** on a cluster of **Power8 nodes (4 GPU/node)**
- **How to build application when `KOKKOS_DEVICE` is Cuda ?**
  - Solution 1: Use `mpicxx` and pass env variable `OMPI_CXX=nvcc_wrapper` [5]
  - Solution 2: Use `nvcc_wrapper` as the compiler, but modify `CXX_FLAGS` / `LDFLAGS` to add MPI specific flags.
- **How to make sure everything is ok regarding hardware affinity ?**
  Cross-check at all possible level ! (so many ways to go wrong)
  - Use `mpirun --report-bindings` to cross-check afterwards how the job scheduler mapped the MPI task to core/host.
  - Use either `Kokkos::OpenMP::print_configuration` / `Kokkos::Cuda::print_configuration`
  - **Check MPI task - GPU binding is what you expect it to be in the application.**

```
int cudaDeviceId;
cudaGetDevice(&cudaDeviceId);
std::cout << "I'm MPI task #" << rank << " pinned to GPU #" << cudaDeviceId << "\n";
```

[5] Use MPICH_CXX is your MPI implementation is MPICH.

MPI + Kokkos on Ouessant (2)

Simple job script for using **MPI+Kokkos/OpenMP**

```
#!/bin/bash
#BSUB -x
#BSUB -J test_mpi_kokkos_openmp                 # Job name
#BSUB -n 4                                       # total number of MPI task
#BSUB -o test_mpi_kokkos_openmp.%J.out          # stdout filename
#BSUB -q compute                                 # queue name
#BSUB -R "affinity[core(10):cpubind=core]"       # affinity
#BSUB -R 'span[ptile=2]'                         # tile : number of MPI task/nod
#BSUB -W 00:05                                   # maximum runtime

module load gcc/4.8/ompi/1.10

# number of OpenMP thread per MPI task
OMP_NUM_THREADS=20

# report bindings for cross-checking
mpirun --report-bindings -n ${LSB_DJOB_NUMPROC} ./test_mpi_kokkos.omp
```

MPI + Kokkos on Ouessant (3)

Simple job script for using **MPI + Kokkos/Cuda**

```
#!/bin/bash
#BSUB -x
#BSUB -J test_mpi_kokkos_cuda              # Job name
#BSUB -n 8                                # number of MPI tasks
#BSUB -o test_mpi_kokkos_cuda.%J.out       # stdout filename
#BSUB -q compute                          # queue name
#BSUB -R "affinity[core(5):cpubind=core]"  # nb cores per MPI task
#BSUB -R "select[ngpus>0] rusage [ngpus_shared=1]"  # activate GPU usage
#BSUB -W 00:05                            # max runtime

module load gcc/4.8/ompi/1.10 cuda/8.0

CUDA_VISIBLE_DEVICES=0,1,2,3
GPUS_PER_NODES=4

# Each mpi tasks are binded to a different GPU
mpirun --report-bindings -n ${LSB_DJOB_NUMPROC} ./test_mpi_kokkos.cuda --ndevices=$GPUS_PER_NODES
```

- This script requests 2 nodes, i.e. $2 \times 4 = 8$ GPUs
- `core(5)`: just to be sure that each Power8 will receive 2 MPI tasks

MPI + Kokkos on Ouessant (4) - Hands-On

About LSF (job scheduler)

- Use code in `code/exercices/mpi_kokkos`; This application just reports bindings
- **Try to build this application against an installed version of Kokkos**, i.e. either OpenMP / Cuda
  - `module use /pwrwork/workshops/patc-201701/kokkos/modulefiles`
  - OpenMP: `module load kokkos/cuda80_gnu485_dev_k80`
  - Cuda: `module load cuda/8.0 kokkos/cuda80_gnu485_dev_k80`
  - `make`
  - This will build either `test_mpi_kokkos.omp` or `test_mpi_kokkos.cuda`
- Open and read `submit_ouessant_cpu.sh` / `submit_ouessant_gpu.sh`
- **Submit a job, read the output and check everything is what is expected**
- LSF commands to know:
  - **submit:** `bsub < submit_ouessant_cpu.sh`
  - **info/status:** `bjobs`
  - **cancel/kill:** `bkill`

Hands-On 5 : Laplace solver with KOKKOS + MPI

Slightly adapted/refactored from Nvidia's OpenACC exercise:
nvidia-advanced-openacc-course-sources
We will use code from `code/exercises/laplace_kokkos`, 4 different versions of the 2D Laplace solver:

- serial (no kokkos)
- **kokkos with 1D view** (linearized index)
- kokkos_v2 with 2D views
- **kokkos_mpi with MPI+CUDA and hwloc**

Hands-On 6 - Reaction-Diffusion Fisher equation (1)

- **SETUP**: we will use git to download this miniApp code designed at CSCS for HPC teaching purpose.
  - `cd $HOME/patc_kokkos/code/miniapps/SummerSchool2016`
  - `git clone https://github.com/pkestene/SummerSchool2016.git`
  - `cd Summerschool2016; git checkout kokkos`
- **This material contains multiple versions** of a **Reaction-Diffusion PDE solver (Fisher equation)**. We will contribute 2 Kokkos versions of this solver.

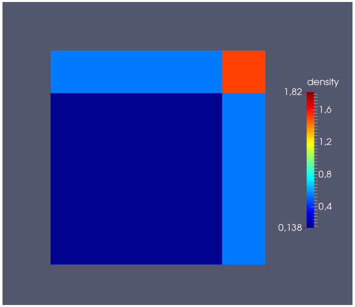
$$\frac{\partial s}{\partial t} = D\left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right) + Rs(1-s) = 0$$

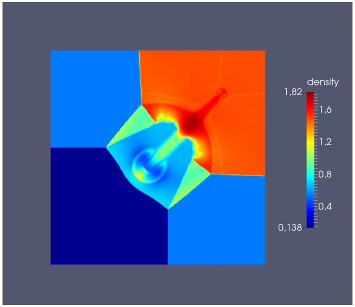
Hands-On 6 - Reaction-Diffusion Fisher equation (2)

1. **Explore/Read slides about the Fisher solver:** `$HOME/patc_kokkos/code/miniapps/SummerSchool2016/miniapp/kokkos/serial/miniapp.pdf`
   - Explore the serial version of the Fisher solver.
2. These **Kokkos exercises** are routed to use the **modulefiles**:
   - `module use /pwrwork/workshops/patc-201701/kokkos/modulefiles`
   - `module load kokkos/openmp_gnu485_dev`
   - `make`
3. **Kokkos version 1** / Exercice with `KOKKOS_LAMBDA` / Already pre-filled, some TODOs
   - Open and read file `miniapp/kokkos/cxx/readme.txt`
   - Fill the TODO with Kokkos LAMBDA kernels
4. **Kokkos version 2** : already done
   - The main difference between version 1 and 2 is how the c++ class `DataWareHouse` is designed
   - Just build and compare performance with version 1, with Kokkos device OpenMP(Power8) and then Cuda

Hands-On 7 : Euler equation solver

- Code location:
  `$HOME/patc_kokkos/code/miniapps/euler2d_kokkos_functor/`
- Build / run / mesure performance of a 2D Euler equation solver OpenMP/Cuda.
- See additionnal slides in source directory





Using an installed Kokkos

- As you will surely **use multiple versions** of Kokkos (OpenMP, Cuda, ...), with/without Lambda, UVM, different compilers, etc ... it will be very usefull to use some **modulefiles**.
- A **module environment** is not a tool specific to a super-computer, it can be used on a **Desktop/Laptop** to configure an execution environment. e.g. `sudo apt-get install environment-modules` (Debian/Ubuntu)
- What is a modulefiles ? A simple way to set env variables to ease the use of a given software package.
- You will find some examples modulefiles for Kokkos in `/pwrwork/workshops/patc-201701/kokkos/modulefiles/kokkos` you can easily adapt to your own platform.

Using an installed Kokkos (2)

- A simple modulefiles for Kokkos should at minimum set variable `KOKKOS_PATH` pointing to the installed directory (the one which contains `Makefile.kokkos`
- **How to use Kokkos modulefiles on your own machine ?** Just use the following:

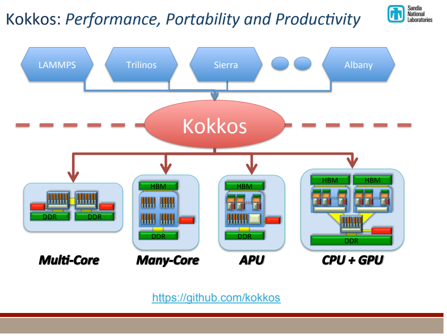
```
# Assuming you placed the module file in
# /somewhere_on_your_machince/modulefiles
module use /somewhere_on_your_machince/modulefiles

# e.g. load Kokkos for GPU
module load kokkos/cuda80_gnu485_dev_k80
```

- **How to use Kokkos modulefiles on ouessant ?** Just use the following:

```
# Assuming you placed the module file in
# /somewhere_on_your_machince/modulefiles
module use /pwrwork/workshops/patc-201701/kokkos/modulefiles
# e.g. load Kokkos for GPU
module load kokkos/cuda80_gnu485_dev_k80
```

About Kokkos in Trilinos

- kokkos is originally a subpackage of trilinos (application framework for solving problems requiring parallel large distributed linear algebra solvers).
- Kokkos is the performance portable layer, to allow running Trilinos as efficiently as possible on multiple architectures.
- **Kokkos can be build independently from Trilinos** and used in other applications


Kokkos: *Performance, Portability and Productivity*
Sandia National Laboratories
LAMMPS
Trilinos
Sierra
Albany
Kokkos
HBM
DDR
Multi-Core
Many-Core
APU
CPU + GPU
https://github.com/kokkos

About Kokkos in Trilinos

- **Don't do the following on `ouessant`, your home is too small**, just keep the spirit to try on your own machine
- **Build a minimal featured Trilinos with Kokkos for GPU activated** : Tpetra + kokkos + Cuda
  1. **Example config plateform:** Ubuntu 16.04 + openmpi + cuda 8.0
     compiler is gcc 5.4.0
  2. **Get Trilinos sources:**
     `git clone https://github.com/trilinos/Trilinos.git; cd Trilinos; git checkout develop`
  3. **CMake configuration script:** Use the provided configuration file `configure_tpetra_kokkos_cuda_nvcc_wrapper.sh` located in the provided archive (`doc/trilinos`)
     this script needs slights changes (var `OMPI_CXX` and install prefix)
     this script must be run in a build directory (not directly in trilinos sources).
     this config will build kokkos with unit tests and examples.
  4. **Build:** `make -j; make install`
  5. **Build a sample project.**

Trilinos/Tpetra example project

- Directory `doc/trilinos/tpetra_example` contains a minimal example application for trilinos/tpetra. You just need to set env variable `TRILINOS_PATH` to install directory.

Kokkos profiling interface (1)

- Kokkos provides by default a profiling interface through a **plugin mechanism**
- **Usage: profiling / monitoring / instrumenting**
- From an application point of view, there is nothing to do, just provide a plugin (shared library), e.g.

```
# define path to the plugin
export KOKKOS_PROFILE_LIBRARY=/somewhere/kp_kernel_logger.so
# run as usal Kokkos application
```

- Examples of Kokkos profile plugins can be found at https://github.com/kokkos/kokkos-tools

Kokkos profiling interface (2)

- A Kokkos profile plugin must provide implementation for callback routines
  - `kokkosp_init_library`
  - `kokkosp_finalize_library`
- A Kokkos profile interface can provide implementation for callback routines specific to a type a parallel construct, e.g. `Kokkos::parallel_for`
  - `kokkosp_begin_parallel_for`
  - `kokkosp_end_parallel_for`

  which are called every time application enters / exits this construct.
- see file `core/src/impl/Kokkos_Profiling_Interface.cpp` for a detailed list of possible callbacks.