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

Pierre Kestener[1]

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

PATC, April 1st - April 2nd, 2019

location: IDRIS, training room

Monday, April 1st, 2019: Kokkos tutorial

- GPU Computing / Cuda refresh
- **Introduction performance portability**
- `ouessant` : a CPU (IBM Power8) + GPU (Nvidia Pascal P100) computing platform : short overview
- **C++ 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)
- **a Kokkos miniapp skeleton project with cmake**

Tuesday, April 2nd, 2019: Kokkos tutorial

- 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: **Illustrate how to use random number generator in kokkos**
  ⇒ RNG 101, parallel compute π with Monte Carlo
- Hands-On 7: CSCS miniApp: **Fisher equation solver**
  ⇒ use Kokkos lambda
- Hands-On 8: CFD miniApp: **Euler solver**
  ⇒ performance measurement for several Kokkos backends (OpenMP, CUDA)

The pedagogical material is located in:
`/pwrwork/workshops/2019-04_patc` **on** `ouessant`

- **About ouessant computing platform**
  - online ouessant user guide :
    `http://www.idris.fr/media/ouessant/ouessant-user_guide.pdf`
  - Use material from IBM/NVidia [1], gives detail on the platform
    See file: `doc/ouessant/Introduction_ouessant_2017.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`
  - **Examples of job submission scripts:**
    `/pwrlocal/ibmccmpl/share/templates/lsf`
    You will need to understand the basics of task affinity on Power architecture (p8aff).
    affinity_power8_lsf_submit.html

[1] Thanks to Nicolas Tallet and P. Vezolle (IBM)

Ouessant : IBM Power8 + Nvidia Pascal P100

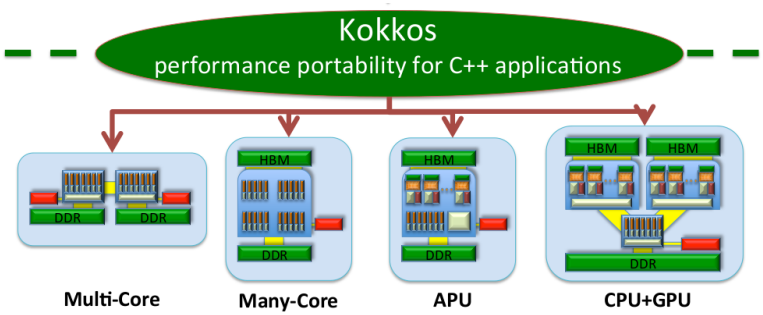
- **About ouessant computing platform**
  - **CPU is IBM Power8 - see output of lstopo**
  - **GPU are K80 (on login nodes) and P100 (on compute nodes)**

- **Official Kokkos documentation**
  - **Wiki on github :** https://github.com/kokkos/kokkos/wiki
  - **Kokkos programming guide** in sources : https://github.com/kokkos/kokkos/blob/master/doc/Kokkos_PG.pdf
  - **Doxygen :** https://trilinos.org/docs/dev/packages/kokkos/doc/html/index.html
- **this PATC training material on Kokkos**
  - Last up-to-date archive for this training is on `ouessant`: /pwrwork/workshops/2019-04_patc/patc_kokkos.tgz
- **C++ reference**
  - http://en.cppreference.com/w/
  - a book on c++, by Peter Gottschling: Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers
  - a book on software engineering best practices : API Design for C++, by Martin Reddy
- **git** : https://githowto.com/

Kokkos: a programming model for perf. portability

- **Kokkos** is a **C++ library** for **node-level parallelism** (i.e. **shared memory**) providing:
  - **parallel algorithmic patterns**
  - **data containers**
- 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** [2], **C. Trott**)

Goal: **ISO/C++ 2020 Standard** subsumes Kokkos abstractions



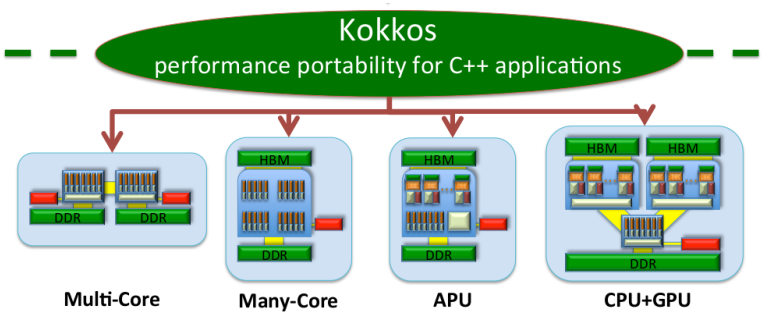
see mdspan proposal https://github.com/kokkos/array_ref

[2] now hired @Nvidia

Kokkos: a programming model for perf. portability

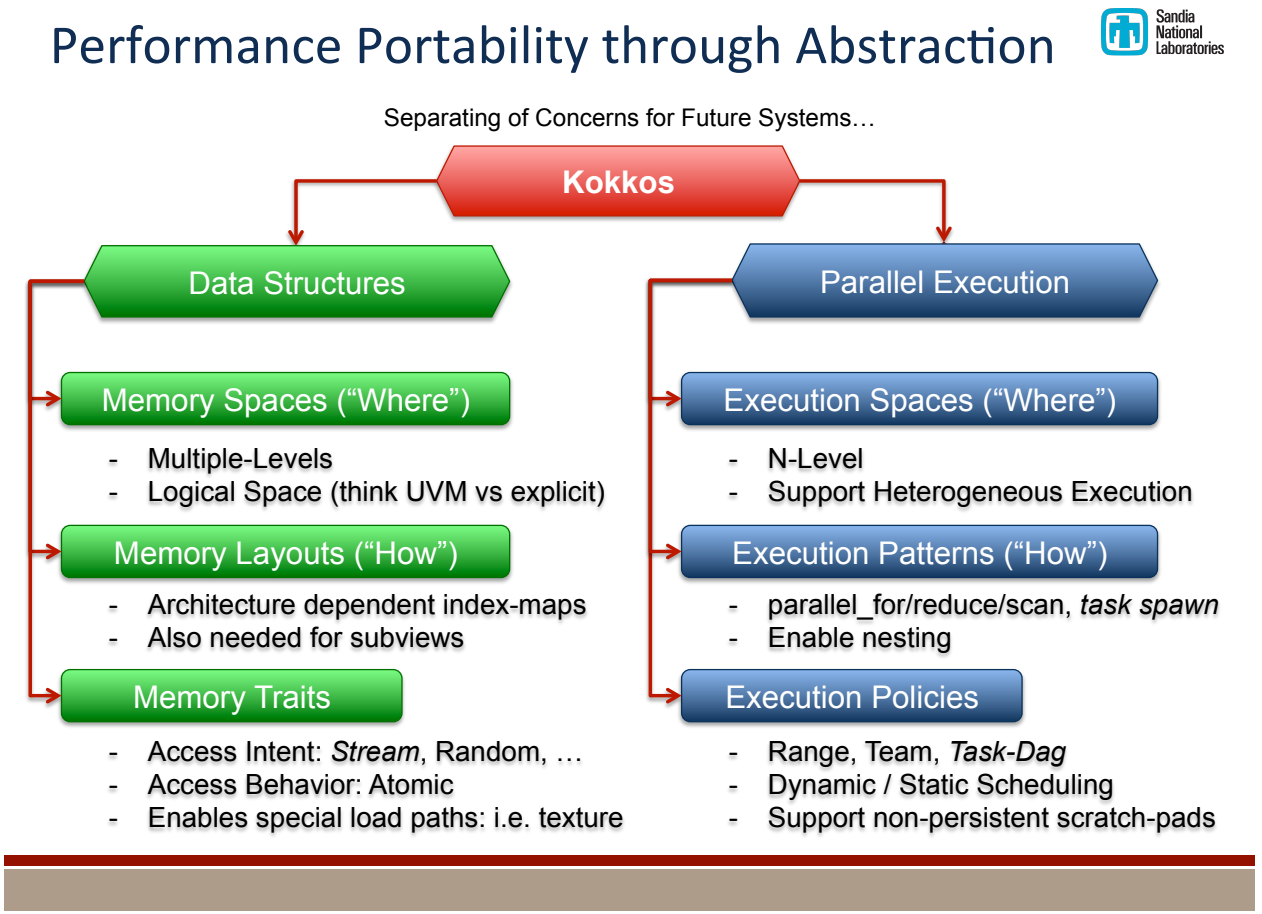
- **Open source**, https://github.com/kokkos/kokkos
- Primarily developped as a base building layer for **generic high-performance parallel linear algebra** in Trilinos
- Also used in
  - LAMMPS (molecular dynamics code),
  - NALU CFD (low-Mach wind),
  - SPARTA/DSMC (rarefied gas flow),
  - Albany (fluid/solid,...)

Goal: **ISO/C++ 2020 Standard** subsumes Kokkos abstractions



see mdspan proposal https://github.com/kokkos/array_ref

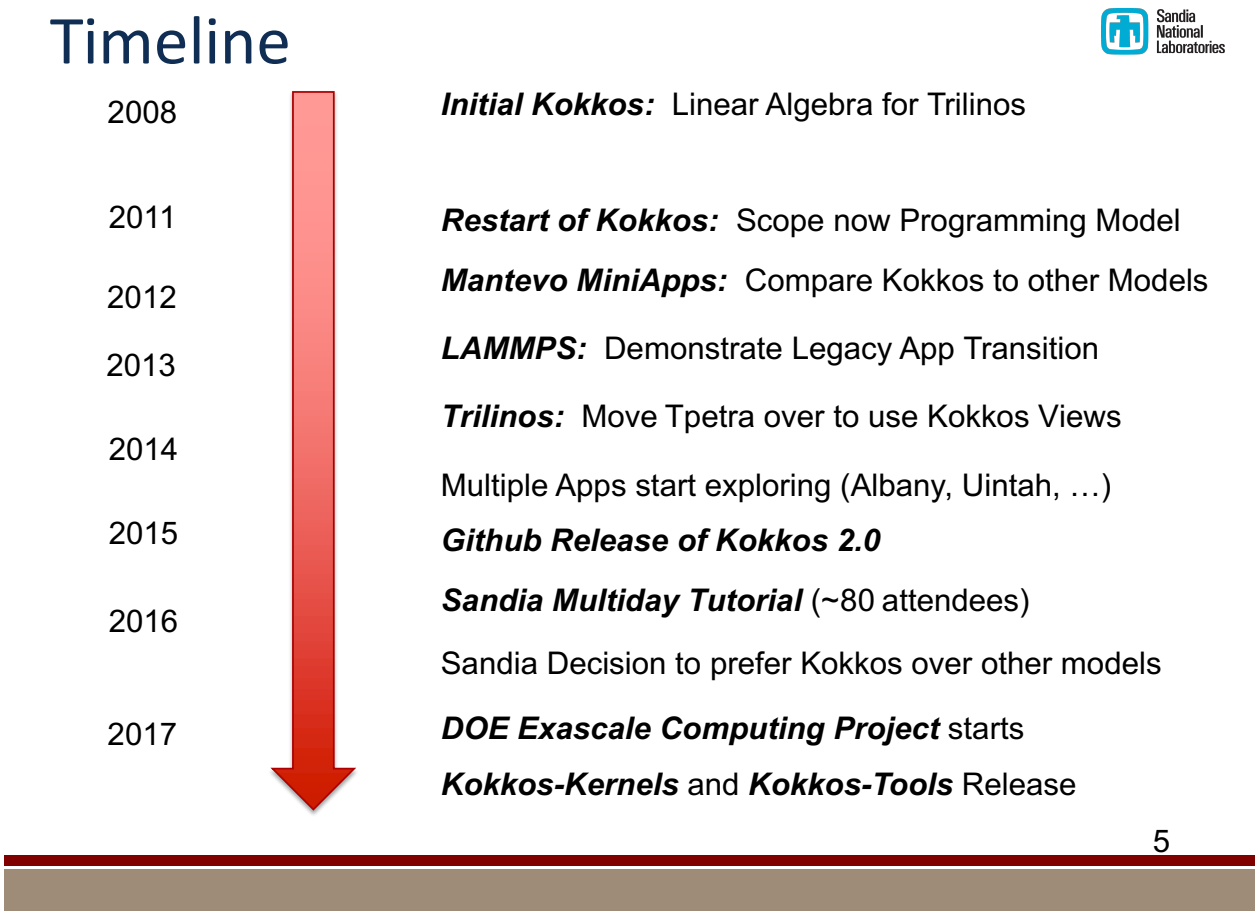
Kokkos: a programming model for performance portability



reference:

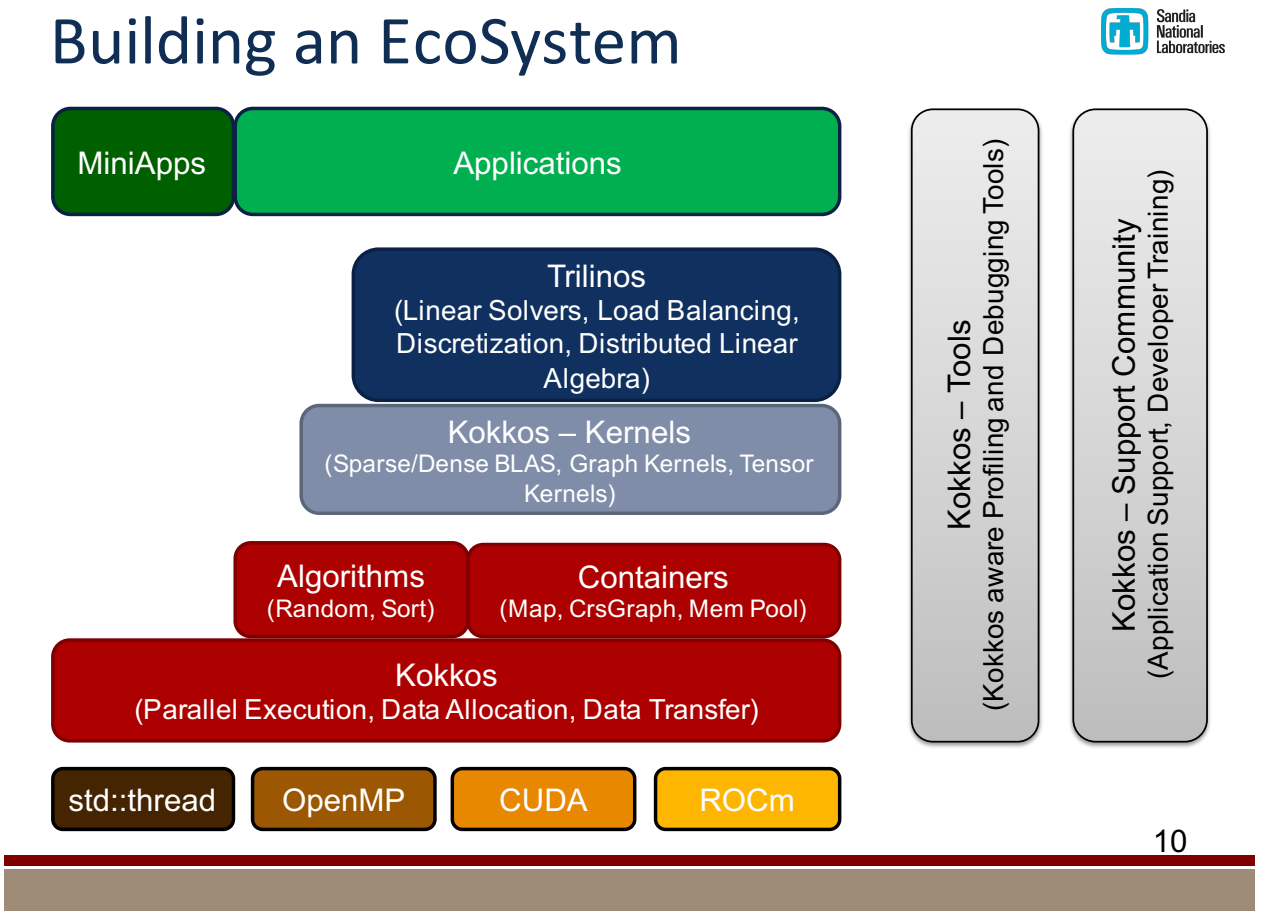
https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/Kokkos-Multi-CoE.pdf

Kokkos: a programming model for performance portability



reference:

https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/Kokkos-Needs-Of-Apps.pdf

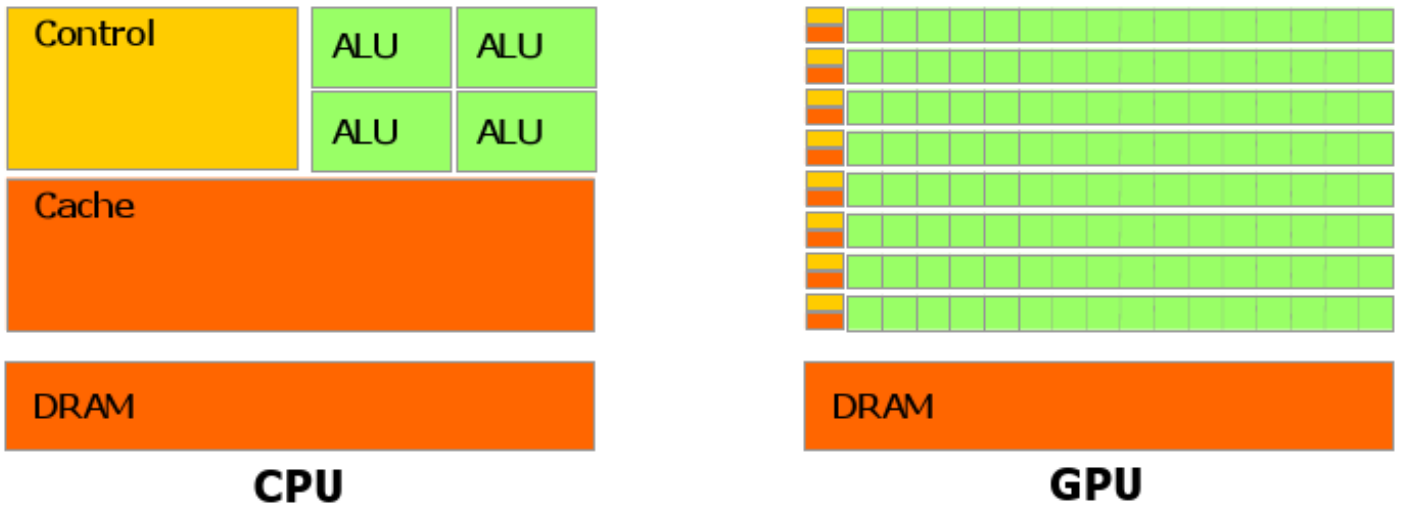


reference:

https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/Kokkos-Needs-Of-Apps.pdf

Main difference between CPU and GPU

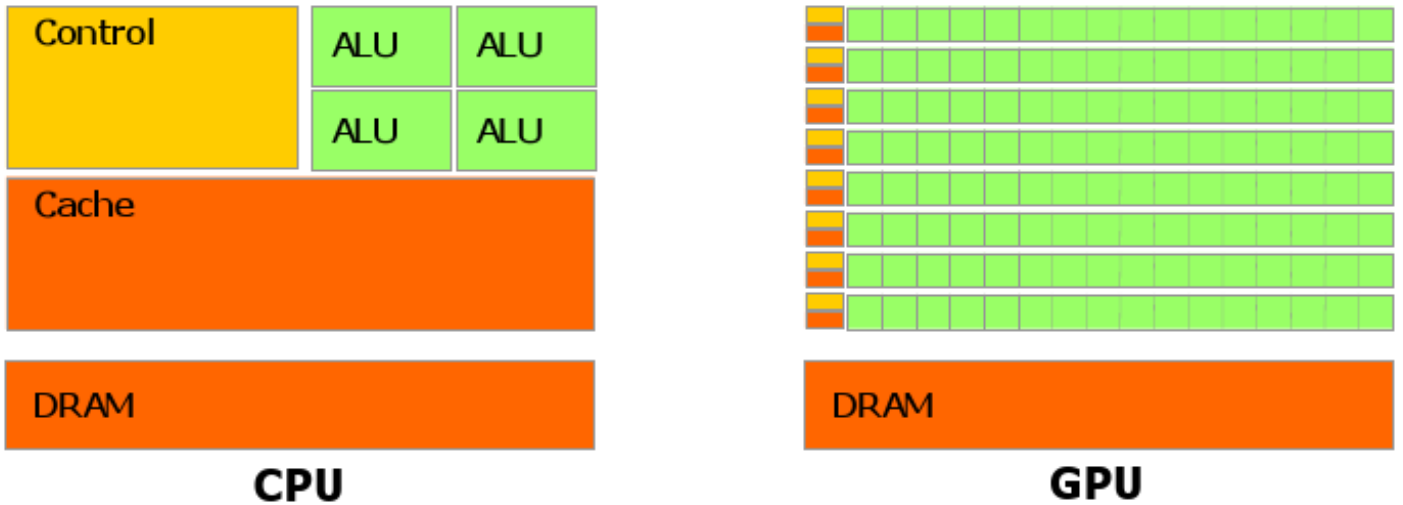
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **Different goals produce different designs:**
  - **CPU** must be good at everything, parallel or not
  - **GPU** assumes work load is highly parallel

Main difference between CPU and GPU

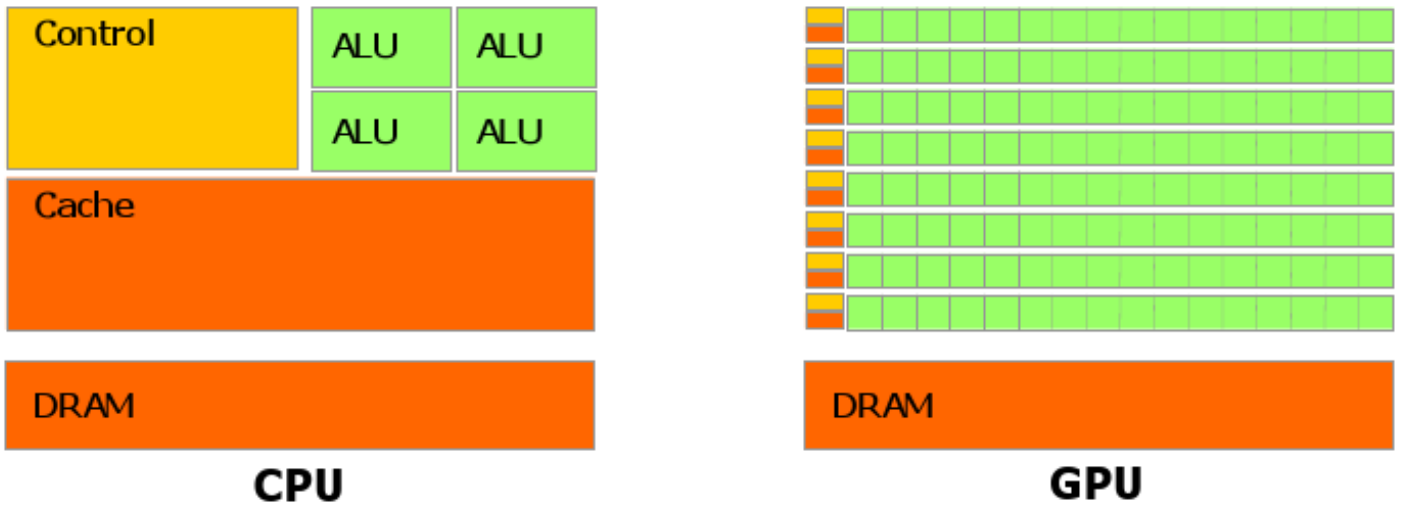
Architecture design differences between manycore GPUs and general purpose multicore CPU ?



- **CPU** design goal : optimize architecture for sequential code performance : minimize latency experienced by **1 thread**
  - sophisticated (i.e. large chip area) control logic for instruction-level parallelism (branch prediction, out-of-order instruction, etc...)
  - CPU have large cache memory to reduce the instruction and data access latency

Main difference between CPU and GPU

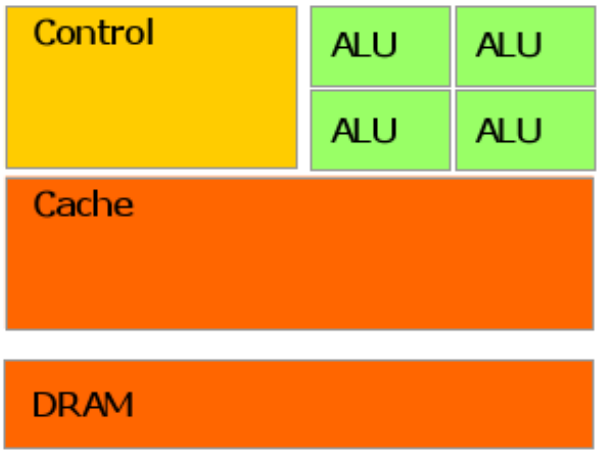
Architecture design differences between manycore GPUs and general purpose multicore CPU ?

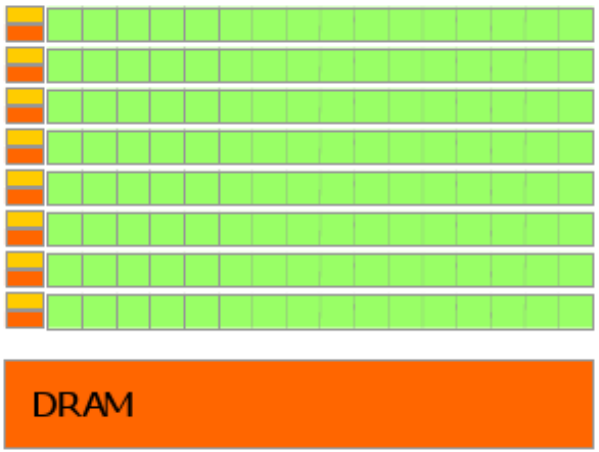


- **GPU** design goal : maximize throughput of **all threads**
  - # threads in flight limited by resources => lots of resources (registers, bandwidth, etc.)
  - multithreading can hide latency => skip the big caches
  - share control logic across many threads

Main difference between CPU and GPU

Architecture design differences between manycore GPUs and general purpose multicore CPU ?





- **CPU** : **1 thread $\Leftrightarrow$ 1 core**

GPU: **# threads $\gg$ # cores**

Memory layouts / index linearization: e.g. 2D

row-major

$n_x(n_y-1)$	$n_x(n_y-1)+1$	$\cdots$	$n_x n_y - 1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$2n_x$	$2n_x+1$	$\cdots$	$3n_x-1$
n_x	n_x+1	$\cdots$	$2n_x-1$
0	1	$\cdots$	n_x-1

index $= i + n_x j$, left layout
fast index on the left

column-major

n_y-1	$2n_y-1$	$\cdots$	$n_x n_y - 1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
2	n_y+2	$\cdots$	$n_y(n_x-1)+2$
1	n_y+1	$\cdots$	$n_y(n_x-1)+1$
0	n_y	$\cdots$	$n_y(n_x-1)$

index $= j + n_y i$, right layout
fast index on the right

Question: Assuming left layout, which loop would you prefer to parallelize first (inner or outer) ?

row-major

$n_x(n_y-1)$	$n_x(n_y-1)+1$	$\cdots$	$n_x n_y - 1$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
$2n_x$	$2n_x+1$	$\cdots$	$3n_x-1$
n_x	n_x+1	$\cdots$	$2n_x-1$
0	1	$\cdots$	n_x-1

index $= i + n_x j$, left layout
fast index on the left

```
for(int j=0; j<ny; ++j)
  for(int i=0; i<nx; ++i)
    data[i+nx*j] += 12;
```

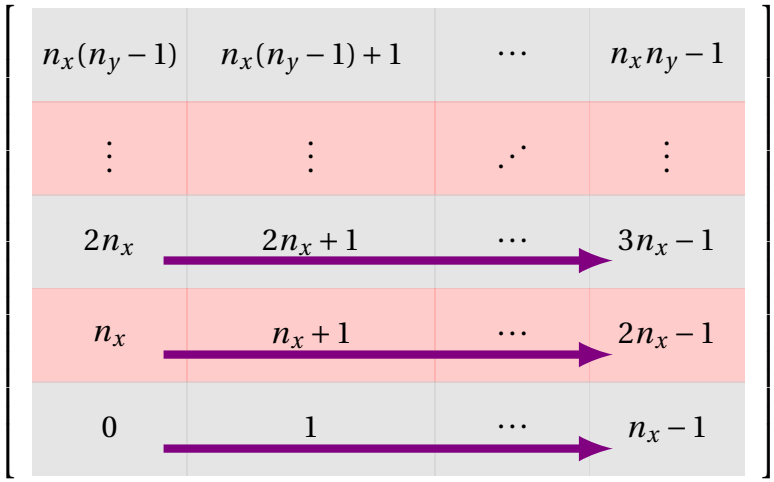
Favor memory locality:

- maximize cache usage for CPU
- maximize memory coalescence on GPU

Different hardware ⇒ different parallelization strategies

Question: Assuming left layout, which loop would you prefer to parallelize first (inner or outer) ?

row-major



index $= i + n_x j$, left layout
fast index on the left

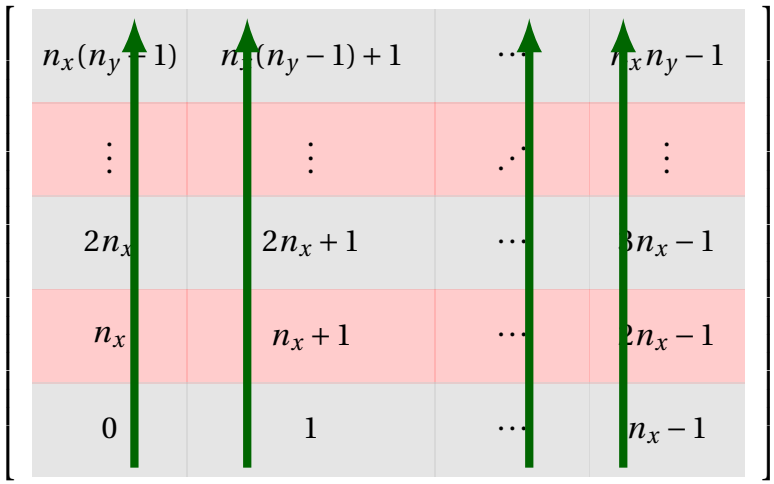
OpenMP // outer loop
each OpenMP thread handles **1 or more row(s)**

```
#pragma omp parallel
{
  #pragma omp for
  for(int j=0; j<ny; ++j)

    // vectorization loop
    // memory contiguity
    for(int i=0; i<nx; ++i)
      data[i+nx*j] += 12;
}
```

Question: Assuming left layout, which loop would you prefer to parallelize first (inner or outer) ?

row-major



index $= i + n_x j$, left layout
fast index on the left

CUDA // inner loop

each CUDA thread handles **1 or more col(s)**

memory coalescence

```
__global__ void compute(int *data)
{
  // adjacent memory cells
  // computed by
  // neighboring threads
  int i = threadIdx.x +
      blockIdx.x*blockDim.x;

  for(int j=0; j<ny; ++j)
    data[i+nx*j] += 12;
}
```

Conclusion

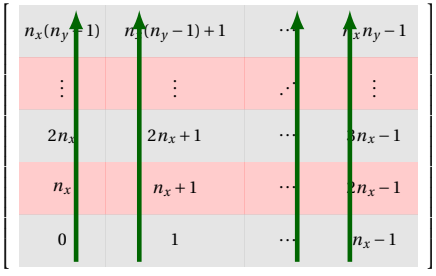
Don't assume layout, let's chose at compile-time !

- **First conclusion:**
  if we keep the same memory layout, **OpenMP** and **CUDA** **disagree** on which loop should be parallelized to optimize for their respective hardware target.
- **How can we make portable code ?**
- Note that swapping memory layout and `for loops` is **involutive**
- **Kokkos answer:** make memory layout abstract (since a good memory layout is hardware dependent), fixed at compile-time
  access $data(i, j)$
  - On **OpenMP** $data(i, j)$ actually means accessing $dataPtr[Ny * i + j]$
  - On **Cuda** $data(i, j)$ actually means accessing $dataPtr[i + Nx * j]$

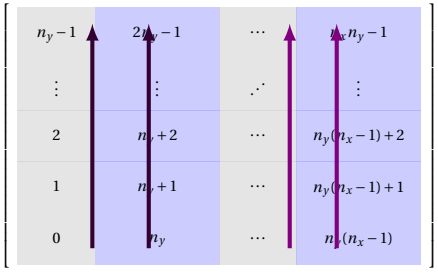
Conclusion

Don't assume layout, let's chose at compile-time !
⇒ **and make it hardware aware.**

left layout / CUDA



right layout / OpenMP



Kokkos parallel version for both CUDA/OpenMP

```
Kokkos::parallel_for(nx,
    KOKKOS_LAMBDA(int i) {
      for (int j=0; j<ny; ++j)
        data(i,j) += 12;
    }
);
```

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, ...)

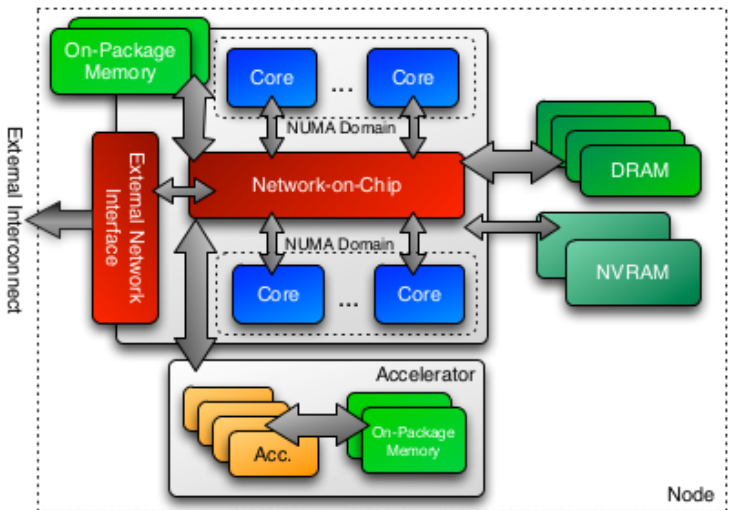


Figure: Conceptual model of a current/future **HPC node**. (Kokkos User's Guide).

Kokkos Concepts (2) - What is a device ?

- Kokkos defines several **c++ class** for representing 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** (we will 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; meanwhile Cuda streams can be used directly (but looses some portability); [2]

[2] kokkos/issues/1723

Hands-On 0: Build kokkos (1)

0. Kokkos is still experimental, but moving fast: use git sources

1. Get Kokkos sources, development branch - don't try to build yet !

- 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. How to build and use

1. **As a regular library (standalone Makefile, installed library):**
   - **not recommended** for production level (see below), **OK for testing and building examples**
   - use `generate_makefile.bash`, then `make kokkoslib; make install`
     Then use a *modulefile* to configure the environment
     Kokkos examples (inside source) can be built that way, as well as Kokkos-tutorials
2. **Embedded Kokkos source files in your application**
   - Why ?
   - ⇒ Kokkos by design has **many different configurations possible** (hardware adaptability, heavily relies on C++ metaprograming - compile timing)
   - ⇒ best practice advice : better compile kokkos as part as the target application (same flags, same compiler, etc...)
   - ⇒ **recommended use**: **standalone cmake + kokkos sources embedded in your application** (we'll see a skeleton app)
3. There exists another cmake-based build sytem, but relies on a third-party tools TriBITS. Right now this can only be used used when Kokkos is build inside Trilinos (heterogeneous distributed sparse and dense linear algebra package).

About standalone Makefile and environment variables settings 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, ROCm, ...
- When building for **CUDA device**, you'll need to use Kokkos' own compiler wrapper: **nvcc_wrapper** (included in Kokkos sources), not just `nvcc`
- 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.

- **Example build configurations (only 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.

Hands-On 0: Build kokkos (5)

- **What is really important:** use **consitently** the same flags for building **kokkos** as well as for building the final application
- **In summary:** **two choices for integrating Kokkos in your application**

1. Use / adapt an existing Makefile from Kokkos tutorial, examples, ...
2. Use your own build system (**cmake recommended**): there can be a quite large combinatorics of `DEVICES`, `ARCH`, compilers, compiler options, ...

- 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**: ROCm > Cuda > OpenMP > Threads > 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` library is available and activated, it provides 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_ENABLE_CUDA )
Kokkos::HostSpace::execution_space::initialize(teams*num_threads);
Kokkos::Cuda::SelectDevice select_device(device);
Kokkos::Cuda::initialize(select_device);
#elif defined( KOKKOS_ENABLE_OPENMP )
Kokkos::OpenMP::initialize(teams*num_threads);
#elif defined( KOKKOS_ENABLE_PTHREAD )
Kokkos::Threads::initialize(teams*num_threads);
#endif
```

- **Prefer using command line argument / env variables**, e.g.:
  `./my_kokkos_app --help; ./my_kokkos_app --kokkos-threads=10`
- **Not really needed** if you run on a supercomputer where the job scheduler will set OpenMP env variables or GPU/task mapping for you.

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
// you may need to add flag : --ndevices=X on the command line
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 1a : Cuda device_query - job submission

Purpose: just make sure you are able to launch a job on Ouessant

- We will use a cuda sample
- In your home on `ouessant`:
  - `cp -a /usr/local/cuda-9.2/samples ./samples-9.2`
  - `cd samples-9.2/1_Utilities/deviceQuery`
  - `module load at/10.0 cuda/9.2`
  - `make`
  - You have an executable named `deviceQuery`
  - You can run it on a **Ouessant login node:** `./deviceQuery`
  - You can run it on a **Ouessant compute node** using the script `submit_ouessant.sh` launched like this:
    `bsub < submit_ouessant_gpu.sh`
    The submission script is located in the training archive (`code/handson/1a/submit_ouessant.sh`)
  - What differences can you see between the two executions ?

Hands-On 1b : Kokkos query_device with hwloc

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 1b : Kokkos query_device with hwloc

Purpose:

- just cross-checking Kokkos/Hwloc is working OK
- On login nodes only for now

TO DO:

- Kokkos sources will be built by the application Makefile
- `cd $HOME/kokkos-tutorial/kokkos/example/query_device`
- open `query_device.cpp` to have a look; no computations, it just prints hardware information
- **Take some time to have a look at the 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 1b : Kokkos query_device with hwloc

1. **Serial build (default, with hwloc):**
   `make KOKKOS_USE_TPLS="hwloc"`
   How many NUMA / Cores / Hyperthreads on power8 CPU ?
   Cross check: what is the current SMT mode on a `ouessant` login node ?
   - `ppc64_cpu --smt`
   - `ppc64_cpu --info`
2. environment: [3] `module load at/10.0 cuda/9.2`
3. **OpenMP build (with hwloc):**
   `make KOKKOS_USE_TPLS="hwloc" KOKKOS_DEVICES=OpenMP` (off course, exact same information obtained as with serial execution)
4. **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

[3] module at/10.0 provides gnu (advanced) toolchain, version 6.3.1 (at/11.0 not compatible with cuda 9.2)

Question: 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. Rebuild and run `./query_device.host --help`
  4. run `./query_device.host --kokkos-threads=12` (alternatively, you can use regular OpenMP environment variables)
  5. change

```
Kokkos::print_configuration( msg );
```

- Rebuild 1 without HWLOC:

```
make KOKKOS_DEVICES=OpenMP
```

- Rebuild 2 with HWLOC:

```
make KOKKOS_DEVICES=OpenMP KOKKOS_USE_TPLS="hwloc"
```

- processor affinity is important to performance; you can/must configure OpenMP environment.

`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 ? $data(i,j)$! *a la Fortran*
- **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 + metadata(array shape, layout, ...)
  - assignment is fast (shallow copy + increment ref counter) [4]
- `Kokkos::View` are designed to be pass by value to a function (**no hard copy**).

[4] 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
- **shallow copy** is default behavior

```
Kokkos::View<int *> a("a",10);
Kokkos::View<int *> b("b",10);
a = b; // a now points to b (ref counter incremented by 1)
// a destructor (memory deallocation) only actually happen
// when ref counter reaches zero.
```

- **Deep copy** must by explicit:

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

  - **Usefull when copying data from a memory space to another**
    e.g. from HostSpace to CudaSpace replacing `cudaMemcpy`
    ⇒ one API for all targets
  - 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 of a 1D array of doubles:
  `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";
```

Other interesting types

- static size `Kokkos::Array` (equivalent to `std::array`)
- can be used inside a Kokkos kernel
- example

```
using vec = Kokkos::Array<double,3>;
```

Interoperability with a legacy C++ API (pointer based)

- `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. Only if you must (e.g. pass data to CuBLAS, ...).
  No more reference counting. Kokkos::View's are reference-counted

Incrementally porting a code to Kokkos

- Use **unmanaged Kokkos::Views**, before using **regular Kokkos::Views**
- Unmanaged view are not reference counted

```
// legacy code allocate memory this way ...
const size_t N0 = ...;
double* x_raw = malloc (N0 * sizeof (double));
{
  // ... but you want to access it with Kokkos.
  //
  // malloc() returns host memory, so we use the host memory space HostSpace.
  // Unmanaged Views have no label because labels work with the reference counting system.
  Kokkos::View<double*, Kokkos::HostSpace, Kokkos::MemoryTraits<Kokkos::Unmanaged> >
  x_view (x_raw, N0);

  functionThatTakesKokkosView (x_view);

  // It's safest for unmanaged Views to fall out of scope before freeing their memory.
}
free (x_raw);
```

What is a functor class ?

Functor = Function object, can be called like a function.

- a simple computation

```
void do_a_for_loop(std::vector<double>& data) {
  for (int i=0; i<data.size; ++i) {
    data[i] += 12;
  }
}
```

What is a functor class ?

Functor = Function object, can be called like a function.

- same with a function pointer

```
void doSomething(double &value) {
  value += 12;
}

// use a function pointeur
void do_a_for_loop(std::vector<double>& data, void f(double&)) {

  for (int i=0; i<data.size; ++i) {
    f(data[i]);
  }
}
```

What is a functor class ?

Functor = Function object, can be called like a function.

- same with a **function object (functor)**

```
class DoSomething {
  // a functor can have parameters, members, execution context, ...
  // can be copied, passed to function, to threads, ...
  DomeSometing(double param) : param(param) {}

  void operator() (double &value) {
    value += param;
  }
  private:
  double param;
}

// use a function pointeur
void do_a_for_loop(std::vector<double>& data, DoSomething f) {
  for (int i=0; i<data.size; ++i) {
    f(data[i]);
  }
}
```

What is a functor class ?

Functor = Function object, can be called like a function.

- same with a **lambda** : lambda = shorhand for a functor, context is captured from the surrounding code.

```
// use a function pointeur
template<class ALambda>
void do_a_for_loop(std::vector<double>& data, ALambda f) {
  for (int i=0; i<data.size; ++i) {
    f(data[i]);
  }
}

double param = 12;
auto domesometing = [=](double& value) {value += param; };

// do some computation
do_a_for_loop(data, dosomething);
```

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

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.

```
// Note: here we use the simplest way to specify an execution policy
// i.e. the first parameter (100)
Kokkos::parallel_for (100, KOKKOS_LAMBDA (const int i) {
  data(i) = 2*i;
});

// is equivalent to the following serial code
for(int i = 0; i<100; ++i) {
  data[i] = 2*i;
}
```

`KOKKOS_LAMBDA` is a preprocessor macro specifying the **capture close**

- by default `KOKKOS_LAMBDA` is aliased to `[=]` to capture variables of surrounding scope **by value**
- `KOKKOS_LAMBDA` has a special definition is CUDA is enabled

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.

```
// Note: here we use the simplest way to specify an execution policy
// i.e. the first parameter (100)
Kokkos::parallel_for (100, KOKKOS_LAMBDA (const int i) {
  data(i) = 2*i;
});

// is equivalent to the following serial code
for(int i = 0; i<100; ++i) {
  data[i] = 2*i;
}
```

Using lambda's means 2 things in 1:

- define the computation body (lambda func)
- launch computation.

How to specify a compute kernel in Kokkos ?

2. **Use a C++ functor class.**

A functor is a class containing a function to execute in parallel, usually it is an `operator ()`

```
class FunctorType {
  public:
  // constructor : pass data
  FunctorType(Kokkos::View<...> data);

  KOKKOS_INLINE_FUNCTION
  void operator() ( const int i ) const
  { data(i) = 2*i; };
};
...
Kokkos::View<int *> some_data("some_data",100);
FunctorType func(some_data); // create a functor instance
Kokkos::parallel_for (100, func); // launch computation
```

- `KOKKOS_INLINE_FUNCTION` is a macro with different meaning depending on target (e.g. it contains `__device__` for cuda)

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

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 kernels

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`:

```
// - local_sum is a temporary variable to transfer intermediate result
//   between threads (or block of threads)
// - sum contains the final reduced result
Kokkos::parallel_reduce (100,
   KOKKOS_LAMBDA (const int i, int &local_sum) {
     local_sum += data(i);
   },
   sum);
```

- Important note: using `parallel_reduce` with lambda is only really usefull if the reduce operation $'+'$
- If the reduce operation is something else, you need to specify:
  - how the local result is initialized (default 0)
  - how the different intermediate results are *joined*

About Kokkos::parallel_reduce with a 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 {...}
}
```

This is useful when the reduced variable is complex (e.g. a multi-field structure)

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);
```

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
- Lambda interface changed:

  ```
  KOKKOS_LAMBDA (const team_member& thread) { ...};
  ```

  team_member is a structure (aliased to `Kokkos::TeamPolicy<>::member_type`)


  ```

Parallel dispatch - execution policy

- **Team policy:** for **hierarchical parallelism**
- `team_member` is a structure equipped with
  - `league_size()` : return number of teams (of threads)
  - `league_rank()` : return team id (of current thread)
  - `team_size()` : return number of threads (per team)
  - `team_rank()` : return thread id (of current thread)
- Can I synchronize threads ?
  Yes, but only threads inside a team (same semantics as in CUDA with `__syncthreads();`)
  ⇒ `team_barrier()`

Team policy: for **hierarchical parallelism**

```
// with the team policy you need to map a thread to an iteration id
KOKKOS_INLINE_FUNCTION
void operator() ( const team_member & thread) {
  // example of data/iteration mapping (similar to CUDA)
  int i = thread.team_rank() +
          thread.league_rank() * thread.team_size();
  data(i) = ... ;
}
```

this very similar to CUDA:

```
// inside a CUDA kernel, using built-in variables
// threadIdx and blocIdx
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

Team policy: for **nested parallelism**

```
// within a parallel functor with team policy
// you can call another parallel_for / reduce / ...
KOKKOS_INLINE_FUNCTION
void operator() ( const team_member & thread) {
  // do something (all threads of all teams participate)
  do_something();

  // then parallelize a loop over all threads of a team
  // each team is executing a loop of 200 iterations
  // the 200 iterations are splitted over the thread of current tea
  // the total number of iterations is 200 * number of teams
  Kokkos::parallel_for(Kokkos::TeamThreadRange(thread,200),
            KOKKOS_LAMBDA (const int& i) {
               ...;
  });
}
```

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
  - Kokkos Task DAG capabilities
  - Example application: Task Parallel Incomplete Cholesky Factorization using 2D Partitioned-Block Layout

SIMD / Vectorization

The following reference give details / best practices to obtain carefully written kernels for portable SIMD vectorization:
http://www.sci.utah.edu/publications/Sun2016a/ESPM2Dan-sunderland.pdf

- `Kokkos::subview` ⇒ allow to extract a *view*

```
// assume data is a 3d Kokkos::View
// slice is a 1d sub view : column at (i,j)
auto slice = subview(data, i, j,  ALL());
```

This is usefull for SIMD, auto vectorization, it helps the compiler understand we are accessing memory with a stride 1 (assuming layout right, which the default for OpenMP device).

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

Proposed activity (get the sources):

1. First, make sure you cloned kokkos sources inside `$HOME/kokkos-tutorial`:
   ```
   mkdir -p $HOME/kokkos-tutorial; cd $HOME/kokkos-tutorial
   git clone https://github.com/kokkos/kokkos.git
   ```
2. From the provided material `cd patc_kokkos/code/handson/2/saxpy`

Proposed activity:

- **Saxpy serial (reference executable on Power8)**
  - `cd handson/2/saxpy/Serial`
  - `make KOKKOS_ARCH=Power8`
  - Alternatively, we could have modified `Makefile` and changed `SNB` (Sandy Bridge) into `Power8`
- **Saxpy regular OpenMP (on Power8)**
  - `cd handson/2/saxpy/OpenMP`
  - Rebuild: `make KOKKOS_ARCH=Power8`; and observe performance by change vector size

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

- **Saxpy Kokkos OpenMP (on Power8)** [5]
  - `cd handson/2/saxpy/Kokkos-Lambda`
  - Add the following lines in `saxpy.cpp` right after Kokkos initialization

```
std::ostringstream msg;
if ( Kokkos::hwloc::available() ) {
  msg << "hwloc( NUMA[" << Kokkos::hwloc::get_available_numa_count()
  << "] x CORE["    << Kokkos::hwloc::get_available_cores_per_numa()
  << "] x HT["      << Kokkos::hwloc::get_available_threads_per_core()
  << "] )"
  << std::endl ;
}

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

  - `make KOKKOS_ARCH=Power8`
  - Make sure all available CPU cores were used (1 × 160 × 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 !

[5] 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.

- **Saxpy CUDA (on Power8 + Nvidia K80/P100)**
  - `cd handson/2/saxpy/Kokkos-Lambda`
  - `module load at/10.0 cuda/9.2`
- 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.

Kokkos - cmake integration (1)

- Why Cmake ?
  - cmake is supported by kokkos
  - easy to integrate and configure (versus e.g. old autotools, versus regular Makefile): need to handle the architecture flags combinatorics
- User application top-level cmake can be as small as 7 lines

```
cmake_minimum_required(VERSION 3.3)
project(myproject CXX)

# C++11 is for Kokkos
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_EXTENSIONS OFF)

# first buid kokkos
add_subdirectory(external/kokkos)

# pass Kokkos include directories to our target application
include_directories(${Kokkos_INCLUDE_DIRS_RET})

# build the user sources
add_subdirectory(src)
```

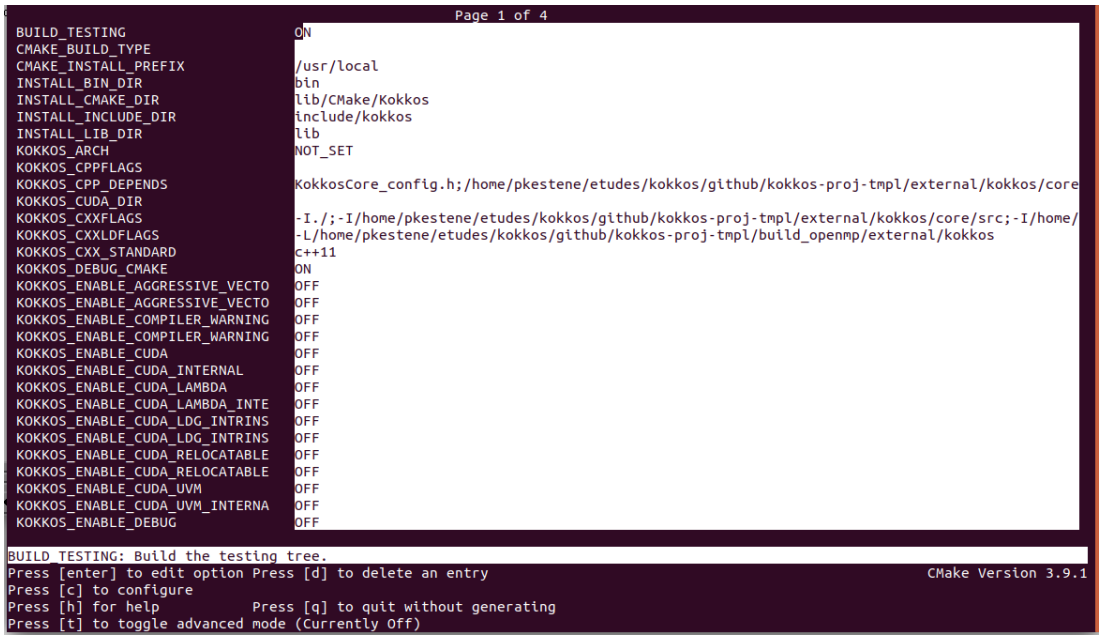
List of important kokkos-related **cmake variables**

- `KOKKOS_ENABLE_OPENMP`, `KOKKOS_ENABLE_CUDA`,... ⇒ which execution space are enabled (multiple possible)
- `KOKKOS_ARCH` (bold values are relevant for `ouessant`), will trigger relevant arch flags (complete list avail. from `Makefile.kokkos`)

```
# Intel:        KNC,KNL,SNB,HSW,BDW,SKX
# NVIDIA:       Kepler,Kepler30,Kepler32,Kepler35,Kepler37,Maxwell,
                Maxwell50,Maxwell52,Maxwell53,Pascal60,Pascal61,
                Volta70,Volta72,Turing75
# ARM:          ARMv80,ARMv81,ARMv8-ThunderX,ARMv8-TX2
# IBM:          BGQ,Power7,Power8,Power9
# AMD-GPUS:     Kaveri,Carrizo,Fiji,Vega
# AMD-CPUS:     AMDAVX,Ryzen,Epyc
```

Kokkos - cmake integration (3)

- curse gui interface: `ccmake`



- command line interface : `cmake mkdir build_openmp; cd build_openmp; ccmake -DKOKKOS_ENABLE_OPENMP ..`
- How to build ? for OpenMP / CUDA ?

Activity: Use the template cmake / kokkos project

- **Clone the template project:**

```
git clone --recursive https://github.com/pkestene/kokkos-proj-tmpl.git
```

- Build the sample application (saxpy): use `ccmake` interface to setup the Kokkos OpenMP target; then try to setup the CUDA target (for arch Kepler37)

```
mkdir build_openmp; cd build_openmp; ccmake ..
# set KOKKOS_ENABLE_OPENMP to ON
make
```

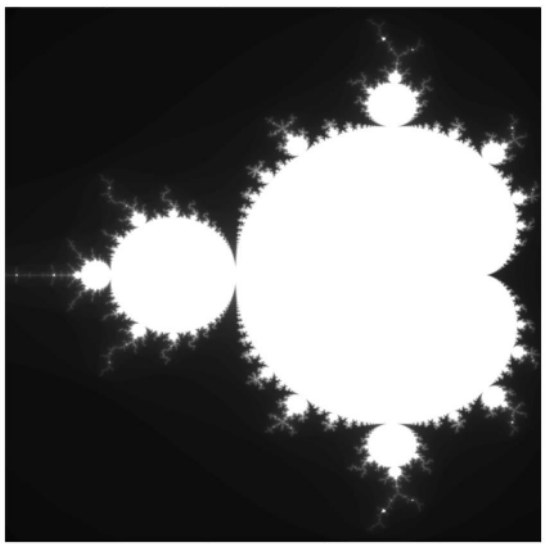
- Build the sample application (saxpy): repeat as above to setup the Kokkos CUDA target (for arch Kepler37)

```
# don't forget to set environment variable CXX
# export CXX="full path to nvcc_wrapper"
mkdir build_cuda_kepler37; cd build_openmp; ccmake ..
# set KOKKOS_ENABLE_CUDA to ON; set KOKKOS_ARCH
make
```

- Try to add another executable; e.g. copy of the tutorial `01_hello_world`

- **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/handson/3/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);
}
```



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

- See kokkos basic version from `code/handson/3/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

- Use code from directory `code/handson/3`; it is designed to work with cmake
- Build the `kokkos_basic` version
- **OpenMP**
  - `mkdir build_openmp; cd build_openmp`
  - `cmake -DKOKKOS_ENABLE_OPENMP=ON ..; make`
- **Cuda**
  - `mkdir build_cuda; cd build_cuda_kepler37`
  - `export CXX=/full/path/to/nvcc_wrapper`
  - `cmake -DKOKKOS_ENABLE_CUDA=ON -DKOKKOS_ARCH=Kepler37 ..`
  - you also add `-DKOKKOS_ENABLE_CUDA_LAMBDA=ON`; you can also build again for architecture Pascal60
  - `make`
- **Compare performance** for a large Mandelbrot set 8192×8192 : OpenMP versus Cuda

- **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.**

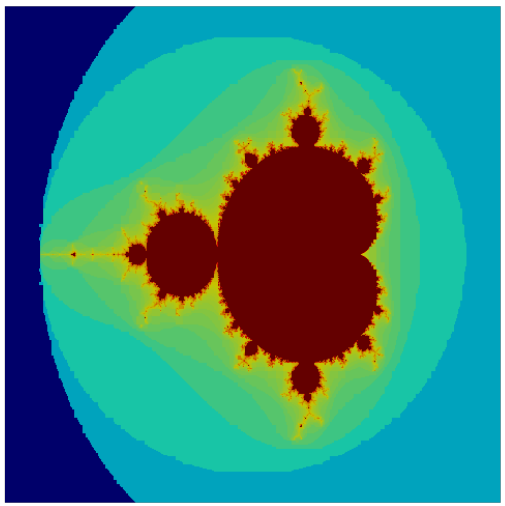
Hands-On 3 : Mandelbrot set

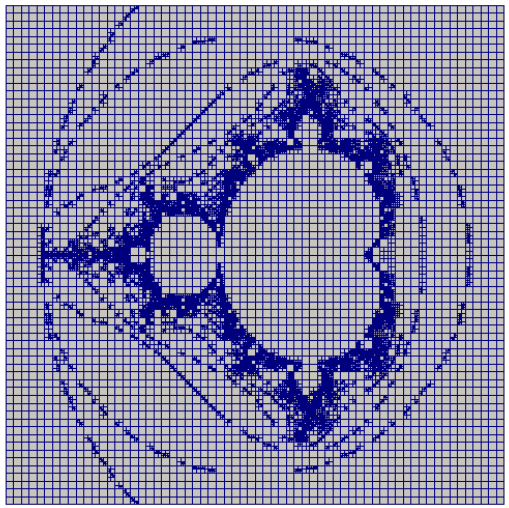
- Pipelined version of Mandelbrot is not currently fully functional; it requires a small patch applied to `Kokkos` for `cudaStreams`;
  see https://github.com/kokkos/kokkos/issues/532
- Understand what is pipelined version of Mandelbrot see:
  http://on-demand.gputechconf.com/gtc/2015/webinar/openacc-course/advanced-openacc-techniques.pdf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

Hands-On 3 bis : Mandelbrot set on adaptive grid

This is an advanced example.

- make use of `Kokkos::UnorderedMap` container to manage the list of cells
- full code is available here:
  https://github.com/pkestene/AMR_mandelbrot.git





- **Purpose:**
  - Illustrate the use of 2D/3D `Kokkos::View`
  - Illustrate the use of alternative execution policies: `Kokkos::Experimental::MDRangePolicy`, `Kokkos::TeamPolicy`,...
- **Stencil kernel:**

```
for (int k=0; k<nz; ++k)
  for (int j=0; j<ny; ++j)
    for (int i=0; i<nx; ++i) {
      y(i,j,k) = -5*x(i,j,k) +
        ( x(i-1,j  ,k  ) + x(i+1,j  ,k  ) +
          x(i  ,j-1,k  ) + x(i  ,j+1,k  ) +
          x(i  ,j  ,k-1) + x(i  ,j  ,k+1) );
    }
```

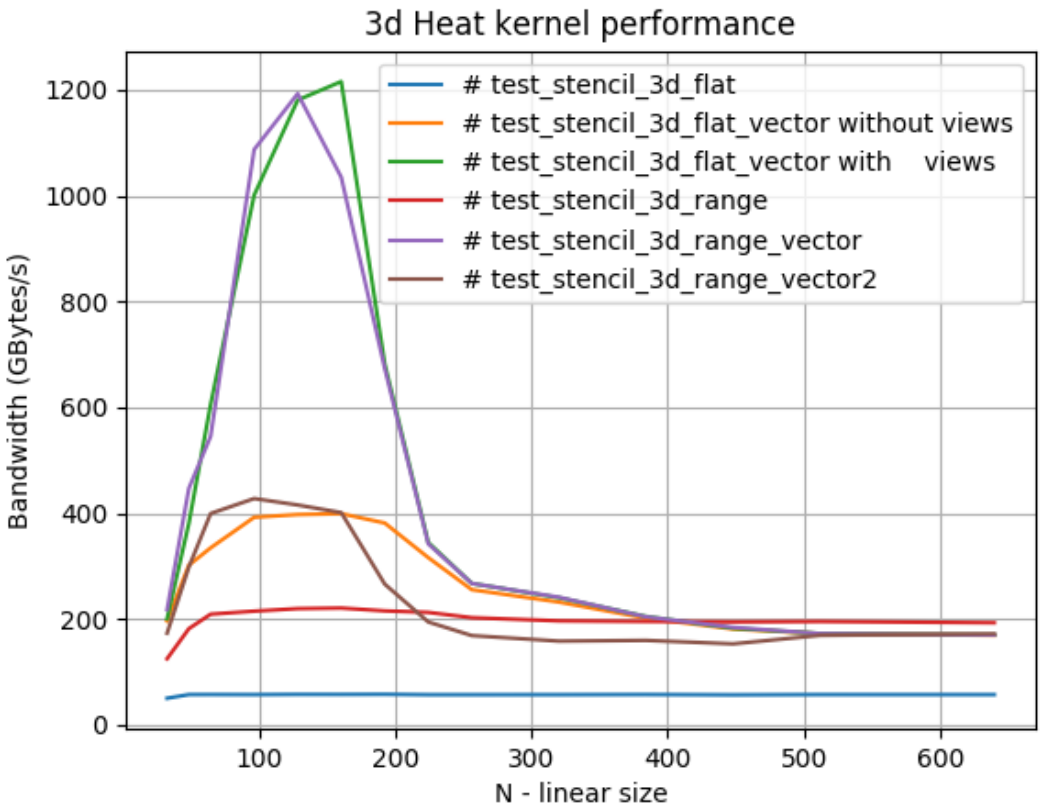
- exercise located in `code/handson/4/stencil`

Work to do:

- follow `readme`
- Once the kernels are done, you can compare the performance for different configurations
  - the difference between kernel implementations, for different sizes
  - run on CPU versus run on GPU
  - interpretation of memory bandwidth measurements

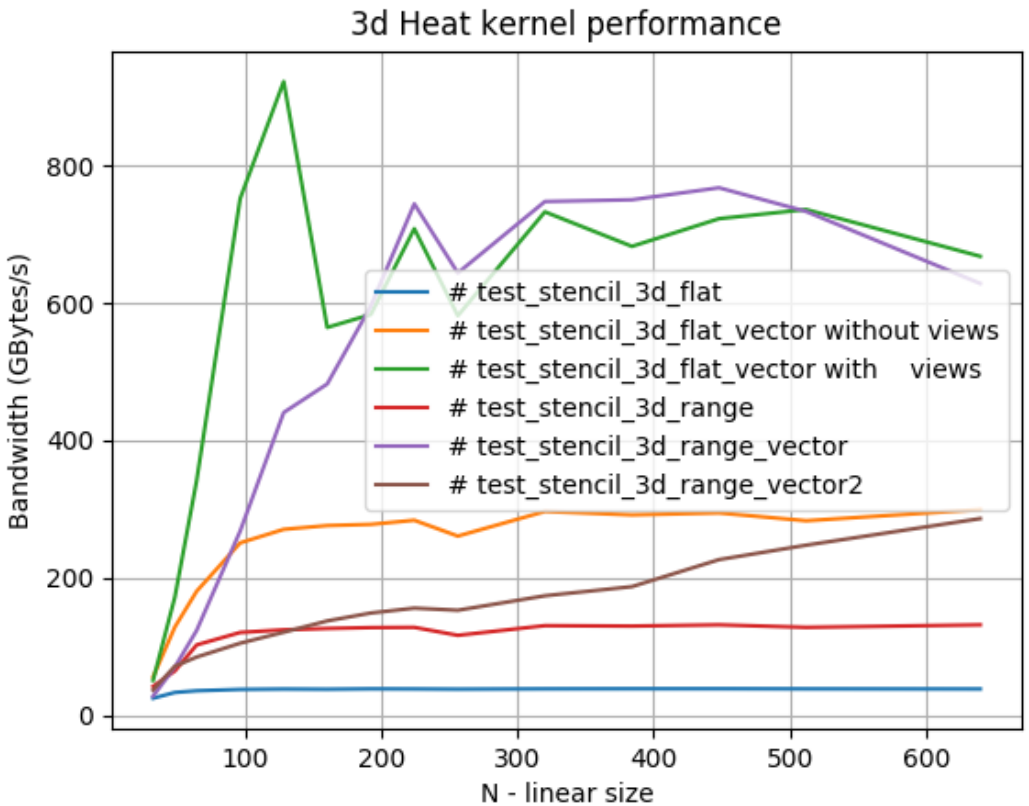
Example of performance obtained on different architectures (can be reproduced using `https://github.com/pkestene/kokkos-proj-tmpl/`):
How to estimate hardware peak bandwidth ? See additional slides at the end

- On Intel skylake (1 socket, 24 cores)



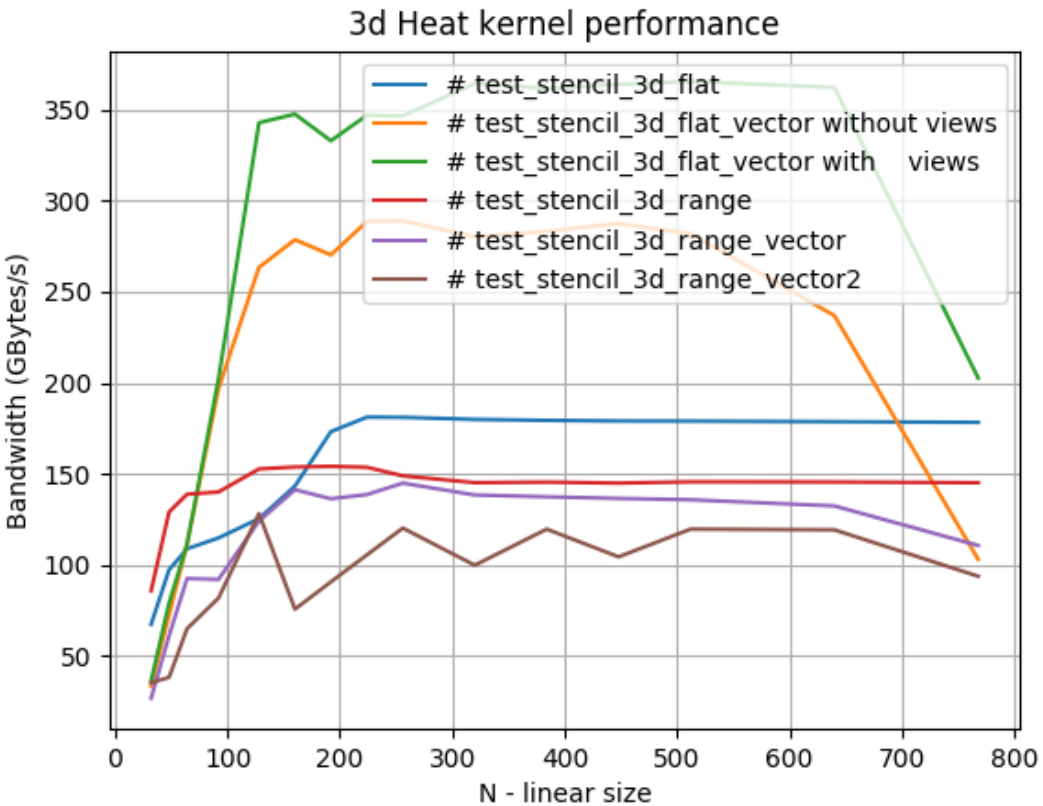
Example of performance obtained on different architectures (can be reproduced using `https://github.com/pkestene/kokkos-proj-tmpl/`):
How to estimate hardware peak bandwidth ? See additional slides at the end

- On Intel KNL (256 threads)



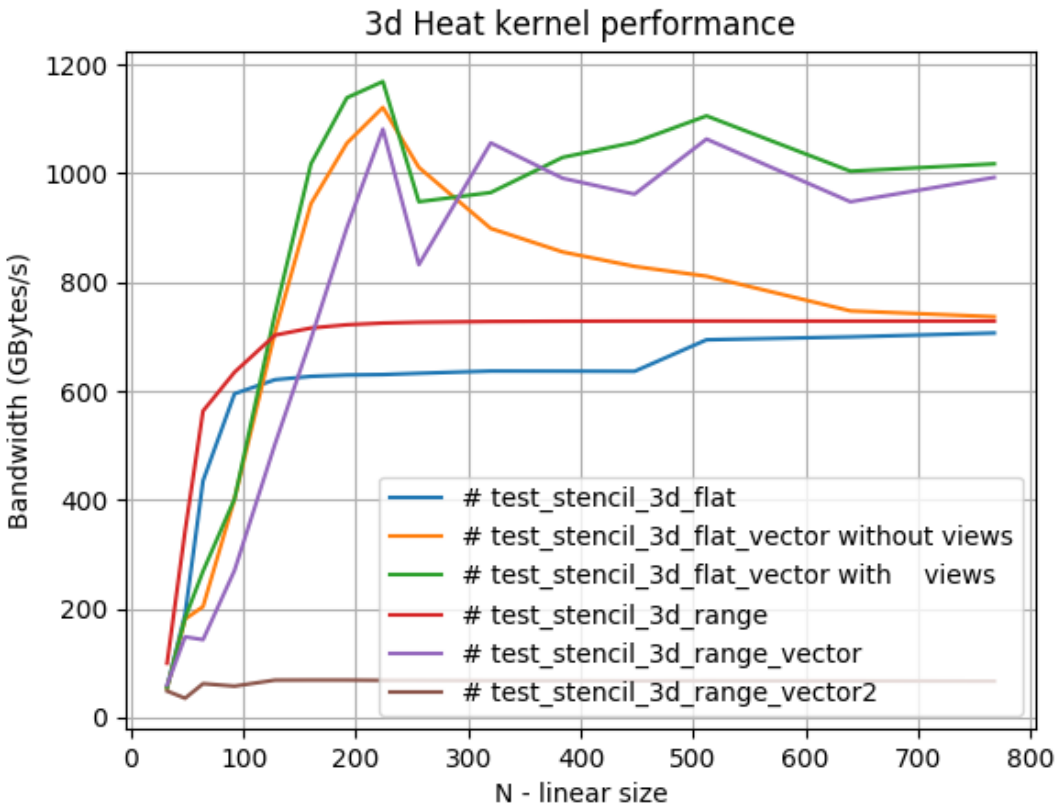
Example of performance obtained on different architectures (can be reproduced using `https://github.com/pkestene/kokkos-proj-tmpl/`):
How to estimate hardware peak bandwidth ? See additional slides at the end

- On Nvidia K80



Example of performance obtained on different architectures (can be reproduced using `https://github.com/pkestene/kokkos-proj-tmpl/`):
How to estimate hardware peak bandwidth ? See additional slides at the end

- On Nvidia P100



MPI + Kokkos on Ouessant (1)

- Perform **distributed computing** on a cluster of **Power8 nodes (4 GPU/node)**
- **How to build an MPI application when `KOKKOS_DEVICE` is Cuda ?**
  - Solution 1: Use `mpicxx` and pass env variable `OMPI_CXX=nvcc_wrapper` [6]
  - 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 `Kokkos::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";
```

[6] Use MPICH_CXX if your MPI implementation is MPICH.

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 computet1                       # queue name
#BSUB -a p8aff(10,8,1,balance)           # 10 OpenMP thread/task, SMT=8
#BSUB -R 'span[ptile=2]'                # tile : number of MPI task/node
#BSUB -W 00:05                           # maximum runtime

module load gcc/4.8/ompi/2.1

# number of OpenMP thread per MPI task
# env variable OMP_NUM_THREADS is set by LSF

# report bindings for cross-checking
mpirun --report-bindings ./test_mpi_kokkos.omp
```

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

```
#!/bin/bash
#BSUB -x
#BSUB -n 8                                  # number of MPI tasks
#BSUB -J test_mpi_kokkos_cuda               # Job name
#BSUB -o test_mpi_kokkos_cuda.%J.out        # stdout filename
#BSUB -q computet1                          # queue name
#BSUB -a p8aff(5,1,1,balance)               # 5 threads/task, 2 tasks/socke
#BSUB -gpu "num=4:mode=exclusive_process:mps=no:j_exclusive=yes"
#BSUB -W 00:05                              # max runtime

module load gcc/4.8/ompi/2.1 cuda/9.2

# Each mpi tasks are binded to a different GPU
# tell kokkos to use up to 4 GPUs/node
mpirun --report-bindings ./test_mpi_kokkos.cuda --ndevices=4
```

- 1 MPI task = 1 GPU
- This script actually requests 2 nodes, i.e. $2 \times 4 = 8$ GPUs
- `p8aff(5,1,1)`: just to be sure that each Power8 will receive 2 MPI tasks

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
  - `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

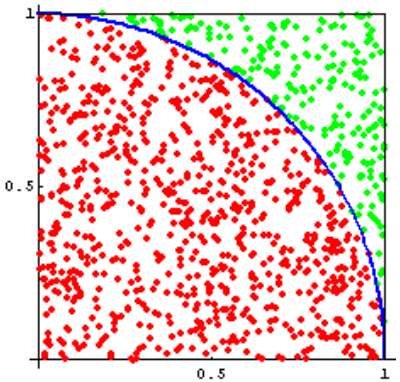
Slightly adapted/refactored from Nvidia's OpenACC exercise:
nvidia-advanced-openacc-course-sources
We will use code from `code/handson/5/laplace_kokkos`, 4 different versions of the 2D Laplace solver:

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

Please note that this exercise uses the old Makefile (no cmake)
⇒ you need to set env variable KOKKOS_PATH to the kokkos source directory.

Hands-On 6 : Random number generator with Kokkos

- Activity: **Estimating Pi via Monte Carlo**
- purpose: learn to use the **random number generator** features (built-in Kokkos)
- Draw random points in $[0,1]^2$ and compute the fraction of points inside the unit circle:



- These generators are based on Vigna, Sebastiano (2014). *An experimental exploration of Marsaglia's xorshift generators, scrambled.* http://arxiv.org/abs/1402.6246
- Use code in `code/handson/6/compute_pi`; read `readme` file; fill the holes
- Which compute pattern will you use ? `parallel_for`, `parallel_reduce`, `parallel_scan` ?

Random number generator in Kokkos: the big picture

- Kokkos defines tuple of types (RNG state, RNG pool)
  e.g. (`Random_XorShift64`, `Random_XorShift64_Pool`)
- Kokkos defines several type of random generator, the main object is a **random number generator pool** of *RNG states*, e.g. `Kokkos::Random_XorShift64_Pool`
  - this is a template class, which takes a Kokkos device as template parameter (Kokkos::OpenMP, Kokkos::Cuda, ...)
  - the pool constructor takes an integral seed to initialize, (option) the number of states in pool
  - it is basically an array of *RNG states*
- A random generator pool defines a subtype to store a given random generator **internal state**: so that inside a functor, one would find:
  `using rng_state_t = GeneratorPool::generator_type`
- rule of thumb:
  One pool ⇔ one functor
  One rng_state ⇔ (use by) one thread

RNG pool interface

get / release a RNG state from the pool in a kokkos thread

```
template<class DeviceType = Kokkos::DefaultExecutionSpace>
class Random_XorShift64_Pool {
  private:
  int num_states_;
  // ...
  public:
  Random_XorShift64_Pool(uint64_t seed) {...}

  KOKKOS_INLINE_FUNCTION
  Random_XorShift64<DeviceType> get_state() const {...}

  KOKKOS_INLINE_FUNCTION
  void free_state(const Random_XorShift64<DeviceType>& state) const
};
```

RNG state interface

```
template<class DeviceType>
class Random_XorShift64 {
  private:
  // state variables...
  public:

  // multiple inline methods to return a rand number
  KOKKOS_INLINE_FUNCTION
  int rand() {
    return ... ;
  }
};
```

`struct rand` interface

- A wrap-up / helper class to draw random numbers with uniform law (static method `draw`):

```
template<class Generator,Scalar>
struct rand {
  //Returns a value between zero and max()
  KOKKOS_INLINE_FUNCTION
  static Scalar draw(Generator& gen);
};
```

- **How to use RNG in a user application ?**
  - the driving code create a **RNG pool**, and pass it to a functor constructor.
  - Inside a kokkos kernel functor, a thread must retrieve a **RNG state** from the pool, **draw some random numbers**, release the **RNG state**.

Exercise:

- open file `src/compute_pi.cpp`; **try to fill the holes (at location of TODO)**
- Explore the OpenMP and Cuda version efficiency

Hands-On 7 - Reaction-Diffusion Fisher equation (1)

- **SETUP**: we will use git to download this miniApp code designed at CSCS for HPC teaching purpose, and modified for testing Kokkos.

```
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 used e.g. in popualtion dynamics)**. We will contribute two Kokkos versions of this solver.

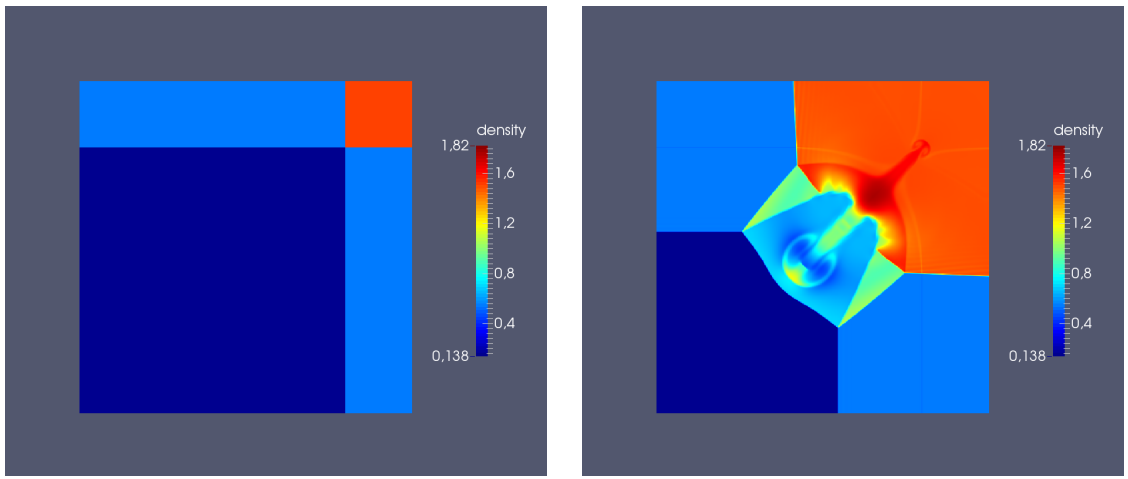
$$\frac{\partial s}{\partial t} = D\left(\frac{\partial^2 s}{\partial x^2} + \frac{\partial^2 s}{\partial y^2}\right) + Rs(1-s) = 0$$

Hands-On 7 - 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 `KOKKOS_PATH` env variable; use the following command line as a starting point
   - `make KOKKOS_DEVICES=OpenMP`
   - `make KOKKOS_DEVICES=Cuda,OpenMP KOKKOS_ARCH=Kepler37`
   - example run: `./fisher.openmp 128 128 100 0.01`
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 8 : Euler equation solver

- Original serial code:
  `$HOME/patc_kokkos/code/miniapps/euler2d_serial`
- See additionnal slides in source directory about CFD numerics
- **Activity 1: Porting code to kokkos:** the serial version has been partially ported to kokkos; fill the TODOs to complete.
- **Activity 2: Build / run / mesure performance** of the kokkos solution (directory `euler2d_kokkos_solution`). Try to plot the OpenMP weak scaling on Power8.
- **How much faster is the GPU version (Pascal P100) versus the Power8 ?**



I strongly recommend starting using cmake to integrate kokkos in your application. But just in case, you're still interested:

- 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

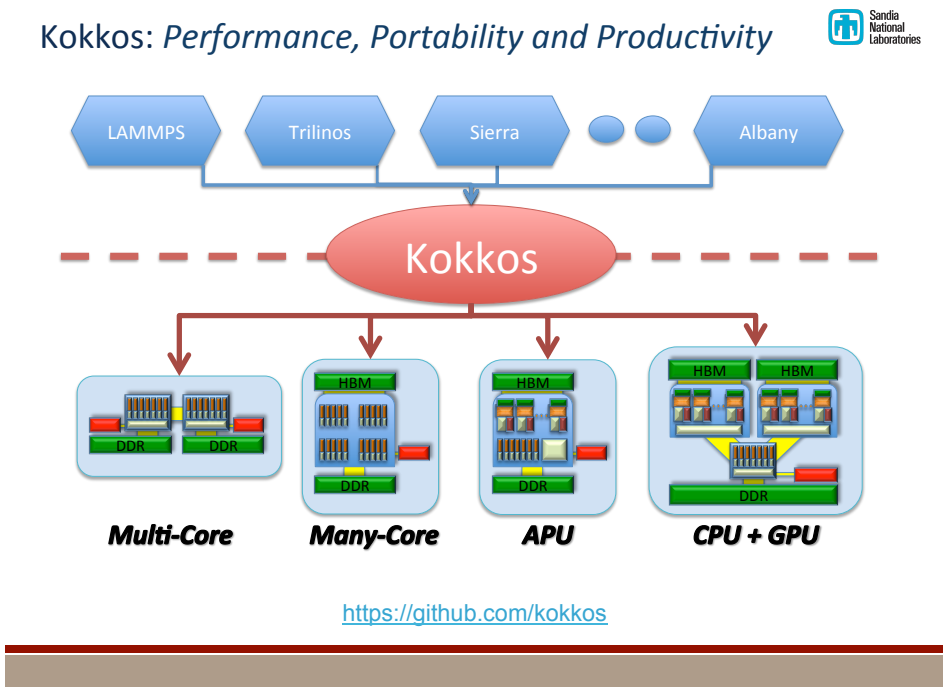
# e.g. load Kokkos for GPU
module load kokkos/cuda80_gnu485_dev_k80
```

- **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



- **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.

How to determine the **peak** hardware memory bandwidth of your compute platform ?

- **Multicore CPU** (e.g. Intel Skylake):
  - Memory type ? e.g. DDR4-2666
  - Number of channels ? e.g. 6
  - Max BW = # NbOfChannel × Frequency(GHz) × BusWidth/8 (Bytes) × # NbOfSockets
  - e.g. on TGCC/IRENE, BW = 6 × 2.6 × 64/8 × 2 = 256 GBytes per node
- **Manycore CPU** (e.g. Intel KNL):
  - depends on HBM configuration (CACHE, FLAT, HYBRID)
  - e.g. KNL on TGCC/IRENE configured in CACHE mode, BW ⩾ 400 GBytes/s
- **NVIDIA GPU** (e.g. Pascal P100):
  - Use sample application `deviceQuery` to retrieve hardware spec.
  - # Memory Clock rate: 715 Mhz
  - # Memory Bus Width: 4096-bit
  - BW = 732.1 Gbytes/s
- **NVIDIA GPU** (e.g. Pascal V100):
  - BW = 898.0 Gbytes/s

What about **achievable** memory bandwidth ?

- Use the stream benchmark. e.g. https://github.com/UoB-HPC/BabelStream
  - Copy : $C[i] = A[i]$
  - Trias : $A[i] = B[i] + scalar * C[i]$
- On **TGCC/Irene/Skylake** (2 sockets per node), one can measure:
  - copy: 190 GBytes/s (74 % of peak)
  - triad: 160 GBytes/s (63 % of peak)
- On **TGCC/Irene/KNL** (1 socket), one can measure:
  - copy: 260 GBytes/s (65 % of peak)
  - triad: 330 GBytes/s (80 % of peak)
- On **NVIDIA P100**:
  - copy: 530 GBytes/s (72 % of peak)
  - triad: 550 GBytes/s (75% of peak)
- On **NVIDIA V100**:
  - copy: 650 GBytes/s (72 % of peak)
  - triad: 860 GBytes/s (95% of peak)

From a pure software engineering point of view, how does **Kokkos** manage to turn **a pur C++ functor** into a **CUDA kernel** ?

1. entry point of parallel computation is through `parallel_for` (function call, templated by execution policy, functor, ...)

```
// parallel_for is defined in
// core/src/Kokkos_Parallel.hpp : line 200
template< class FunctorType >
inline
void parallel_for( const size_t        work_count
                 , const FunctorType& functor
                 , const std::string& str = ""
                 )
{
  // ...
  Impl::ParallelFor< FunctorType , policy >
  closure( functor , policy(0,work_count) );
  // ...
}
```

2. `closure` is an instance of the **driver** class `Kokkos::Impl::ParallelFor`; the precise object type created is off course Kokkos-backend dependent
3. If CUDA backend is activated, the instantiated class `Kokkos::Impl::ParallelFor` is defined in `Cuda/Kokkos_Cuda_Parallel.hpp`; there are multiple specialization for the different execution policies (Range, multi-dimensional range, team policy, ...); e.g. for range

```
template< class FunctorType , class ... Traits >
class ParallelFor< FunctorType
                 , Kokkos::RangePolicy< Traits ... >
                 , Kokkos::Cuda
                 >
{
  // this is where for a given iteration id, the functor is called
  // kind of generic cuda kernel work definition
  inline   __device__  void operator()(void) const { ... };

  // this is where the actual CUDA kernel run time config
  // is setup : block and grid dimension
  // then create a CudaParallelLaunch object
  inline void execute() const { ... };
}
```

4. when `closure.execute()` is called, an object `CudaParallellaunch` is created
5. struct `CudaParallellaunch` contains only a constructor, which only purpose is to actually launch the CUDA kernel (using the <<< ... >>> syntax)
6. Copy closure (driver instance) to GPU memory (either constant, local or global) using Cuda API (e.g `cudaMemcpyToSymbolAsync` to copy constant memory space)
7. finally the actual generated cuda kernel, using one of the static functions defined (e.g. `cuda_parallel_launch_constant_memory`)