Pourquoi (j'aime bien) Kokkos ?
Modern C++, portabilité de performance, ...

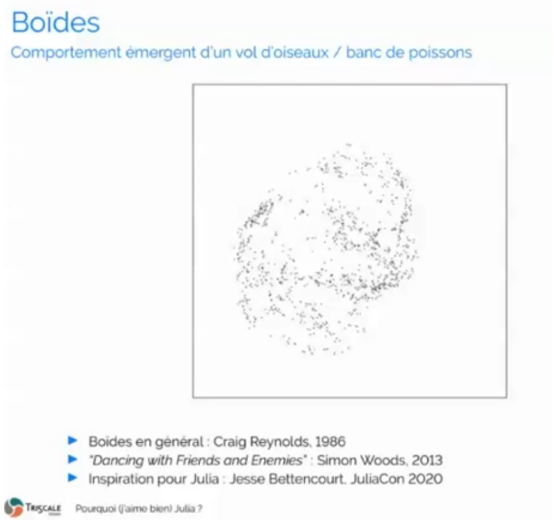
Pierre Kestener

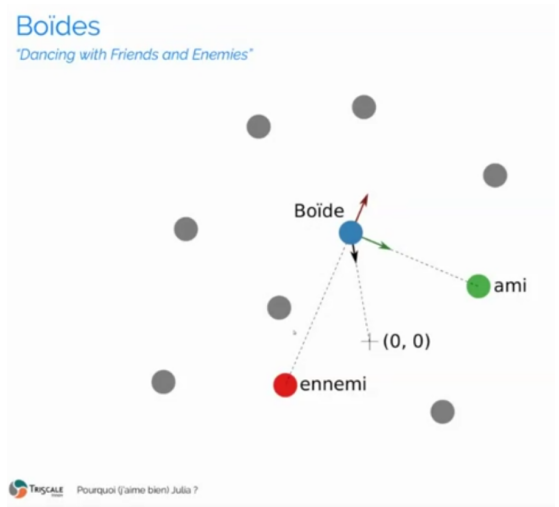
CEA Saclay, DRF, IRFU/DEDIP/LILAS

Café CALCUL - 22 octobre 2021

Plan

- 8 avril 2021 : Café Calcul - Pourquoi Julia ? (F. Févotte)
  le problème des deux languages (script polyvalent + bas niveau performant)
  ⇒ **Julia**
- le problème du modèle de programmation (polyvalent, multi architecture)
  ⇒ **portabilité de performance**





nanoApp : (naively) revisiting boids flight with Kokkos,
https://github.com/pkestene/kboids

(Pre-)Exascale machines - architecture diversity !

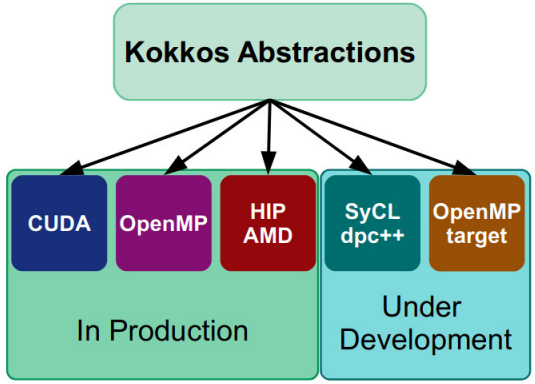
- **US**: Summit , Sierra ⇒ mostly OpenPower (IBM P9 + Nvidia V100), GPU-based architecture, #2 and #3 @top500; exascale machines announced
  - Aurora (Argonne NL, 2022): Intel Xe GPU
  - Frontier (Oak Ridge NL, 2021 ?): AMD EPYC + Radeon Instinct GPU
- **China:**
  - Phytium FT2000/64 ARM chips + Matrix2000 GPDSP accelerators ⇒ #6 @top500, Tianhe-2A, 61 PFlops
  - 260-core Shenwei, **homegrow technology** hardware + software (C++/fortran compiler + OpenACC) ⇒ #4 @top500 , Sunway TaihuLight, 105 PFlops
  - Dhyana, AMD-licenced x86 multicore (300 M$), identical to AMD EPYC
- **Japan: Fugaku**(Fujitsu, ARM, RIKEN) A64FX ARM (**home grown**, started in 2014, **#1 @top500 (Nov. 2020)**, 900 M$), GPU, etc ...
- **Europe** : new organization EuroHPC (2018), EC H2020 budget (~ 500 M€ per year)
  home grown (EPI) ARM and RISC-V architecture, early stage

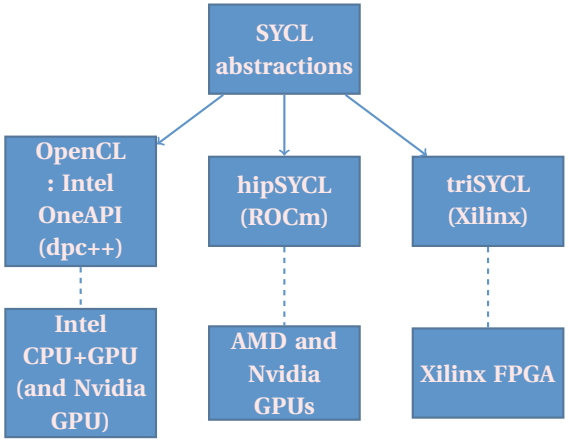
Motivations for performance portability

- What is performance portability ?
  - **(Re)write your code once, (try to) run *efficiently* everywhere**
  - By everywhere, we mean : Multicore Intel/ARM and Nvidia/AMD GPUs
  - **High-level approach:** as much as possible (if possible) hide hardware details to the (physicist / applied math) software developer
  - https://performanceportability.org
  - 1st annual DOE Performance Portability Meeting (2016)
- Is that **possible** ?
  How ?
  Which programming model ?
  Which language ?
  Which compiler ? ⇒ large combinatorics
- for the rest of this talk, i'll focus on the kokkos/C++ library

Parallel programming models landscape

- **Low-level native language:** OpenCL, CUDA, HIP
- **Directive approach (code annotations)** for multicore/GPU, ...:
  - OpenMP 5.1 (Clang, PGI, GNU, ...), OmpSs-2
  - OpenACC 2.7 (PGI, GNU, ...) ⇒ Fortran codes.
- **Other high-level library-based approaches**:
  - Kokkos, RAJA, Alpaka, HPX, GridTools, ArrayFire...
  - SYCL (Khronos Group *standard*), C++ high-level layer on top of OpenCL. Intel OneAPI/DPCPP (Intel CPU/GPU/FPGA, Nvidia GPUs), CodePlay, AMD and Nvidia GPUs, Keryell/Xilinx
  - **C++-17 built-in parallelism for multicore and GPUs**, e.g.:
    - Nvidia's hpc-sdk (May 2020)
    - Intel OneAPI/TBB





additionnal features:

memory management,

data containers, ...

Kokkos (2010), before C++-11 and lambdas

- Before 2010, starts as a refactoring of Trilinos (10.4), abstract concept of `Node` (generic for SerialNode, TBBNode or CUDANode)
  conf paper: A light-weight API for portable Multicore Programming

```
// data-work struct
template <class Node>
class AxpyOp {
  Node::buffer x,y;
  double alpha,beta;
  void execute(int i);
};

template <>
void SomeNode::parallel_for<AxpyOp>(int begin, int end, AxpyOp wd) {
  // node specific implementation
  // if SomeNode == TBB, then call TBB API
  // if SomeNode == CUDA, then call Cuda thrust::for_each
  // ...
}
```

Kokkos: a programming model for perf. portability

- **Kokkos** is a **C++ library** for **node-level parallelism** (i.e. **shared memory**) providing abstractions for **harware-aware**:
  - **parallel algorithmic patterns**
  - **data containers**
- `https://kokkos.org/`
- Implementation relies heavily on **C++ meta-programing** to derive native low-level code (OpenMP, CUDA, HIP, SYCL...) and adapt data structure memory layout at compile-time
- Developped at **Sandia NL** (core, CUDA, OpenMP), **ORNL** (HIP, SYCL), ...

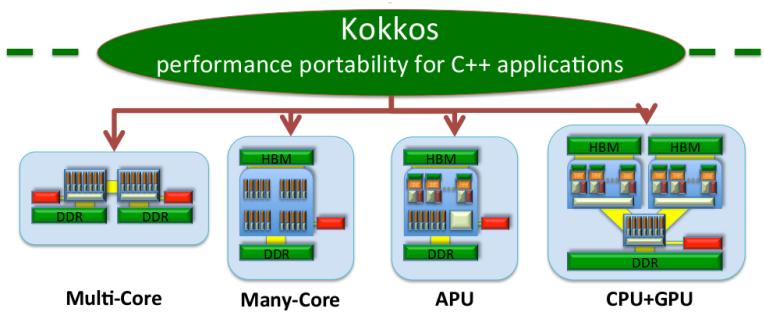
Goal: write one implementation which:

- compiles and **run on multiple archs**,
- obtains **performant memory access pattern** across archs,
- can leverage **arch-specific features** where possible.

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
- Used in, e.g.:
  - LAMMPS (molecular dynamics code),
  - NALU CFD (low-Mach wind flow),
  - SPARTA/DSMC (rarefied gas flow), SPARC (CFD, RANS, LES, hypersonic flow)
  - Albany (fluid/solid,...)
  - Uintah (structured AMR, combustion, radiation)

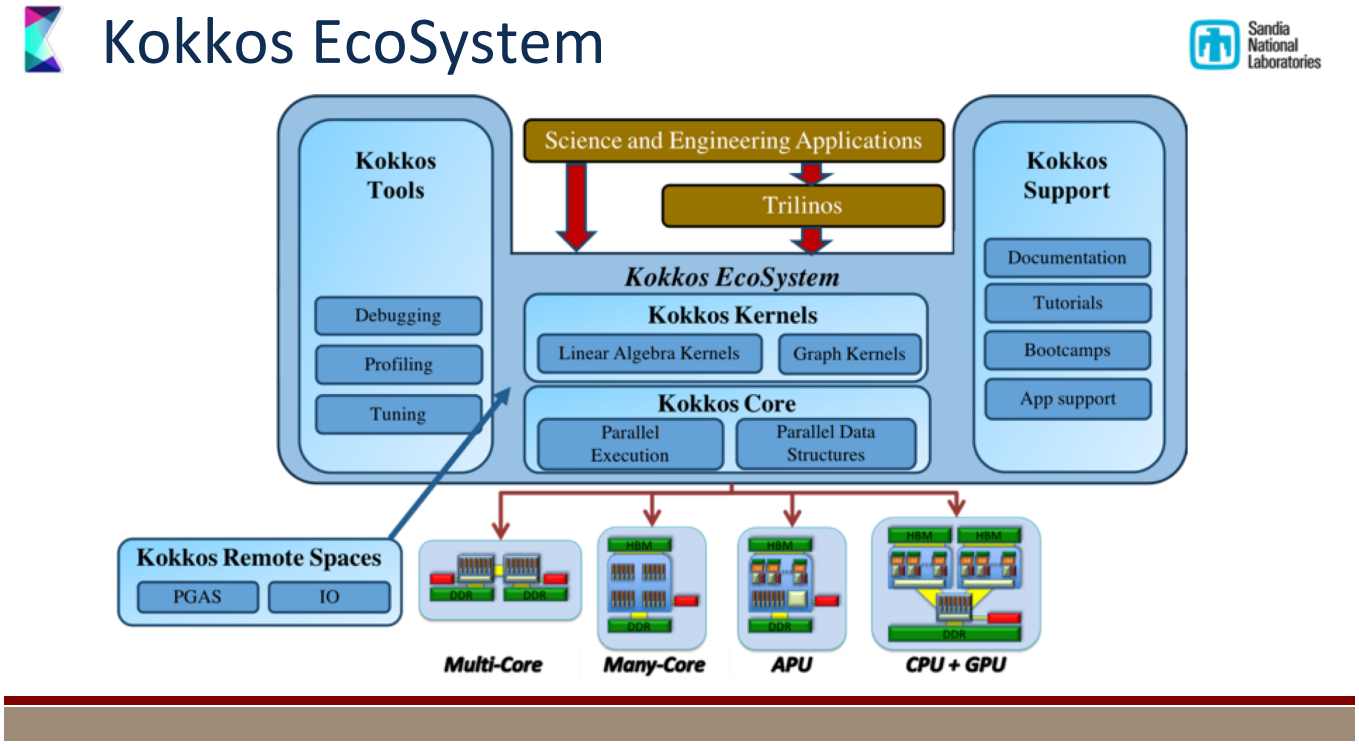
Strong involvement in
ISO/C++ 2020 Standard
Make Kokkos a sliding window of future c++ features



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

https://arxiv.org/abs/2010.06474

Kokkos: a programming model for perf. portability



- Kokkos-kernels (many dense/sparse BLAS problems, ...), simd-math, Cabana (for particle-based codes)
- Fortran compatibility layer (REX code XGC-Cabana, Plasma physics, Gyrocinetics, particle-in-cell)
- pykokkos-base (pybind11-based API mapping + memory, numpy/cupy interoperability), pykokkos (decorator + python to C++ translation)
- Task-DAG parallelism (CPU / GPU)

source: C. Trott, DOE Performance Portability Meeting, April 2019

Kokkos - Documentation

- Kokkos video lectures + slides : `https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Seri`
- Kokkos tutorial : `https://github.com/kokkos/kokkos-tutorials`
- Kokkos source code itself, reading unit tests code is also very helpful

Illustrating portability with Kokkos

```
for(int j=0; j<ny; ++j)
  for(int i=0; i<nx; ++i)
    data[i+nx*j] += 42;
```

Question: Assuming 2d data with left layout, but only 1 loop to parallelize, which one would you prefer to parallelize (inner or outer) ?

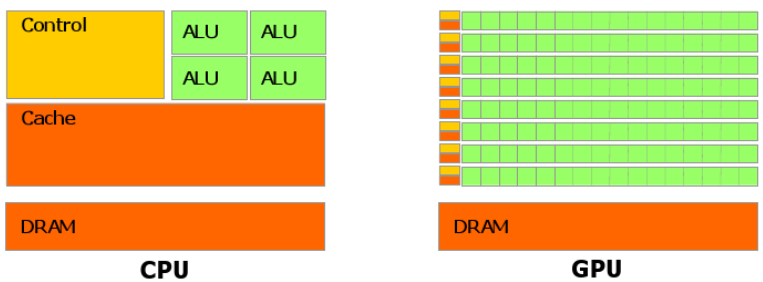
Answer:
Optimize memory access pattern !

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

left-layout = 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

Different hardware ⇒
Different parallelization strategies

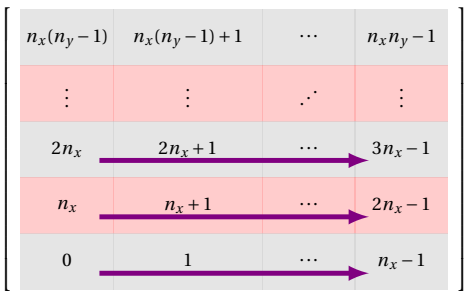


Illustrating portability with Kokkos

Question: Assuming 2d data, left layout, which loop would you prefer to parallelize (inner or outer) ?

OpenMP // outer loop

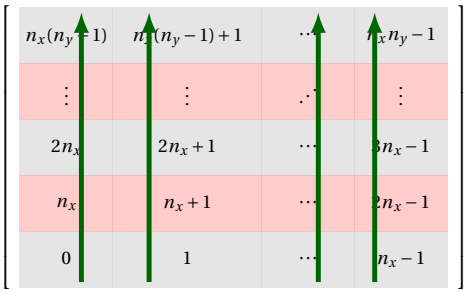
```
#pragma omp parallel
{
  #pragma omp for
  for(int j=0; j<ny; ++j)
     #pragma omp simd ivdep
    for(int i=0; i<nx; ++i)
      data[i+nx*j] += 42;
}
```



CUDA // inner loop

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

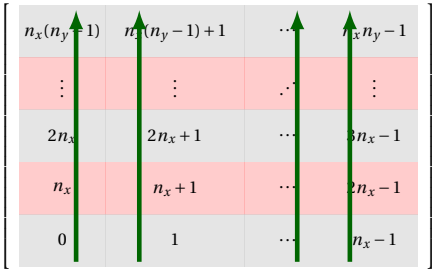
  for(int j=0; j<ny; ++j)
    data[i+nx*j] += 42;
}
```



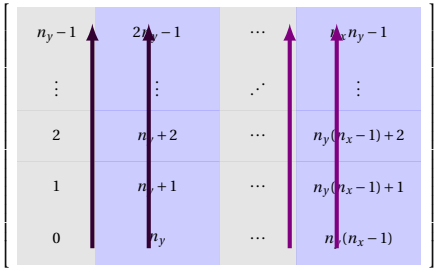
Illustrating portability with Kokkos

Let's chose memory layout at compile-time
Make it hardware aware.

left layout / CUDA



right layout / OpenMP



Kokkos single parallel version (CUDA+OpenMP)

Kokkos/CUDA defaults to **left-layout**

Kokkos/OpenMP defaults to **right-layout**

```
Kokkos::parallel_for(nx,
    KOKKOS_LAMBDA(int i) {
      for (int j=0; j<ny; ++j)
        data(i,j) += 42;
    }
);
```

Kokkos Concepts (1) - the abstract machine model

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

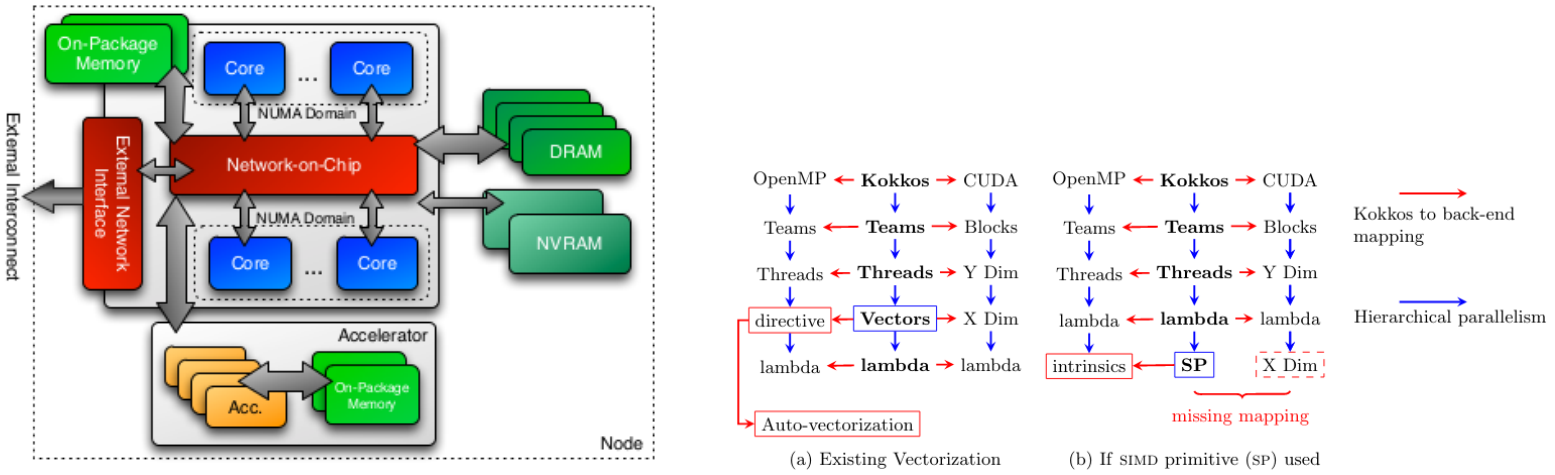


Figure: (left) Conceptual model of a current/future **HPC node**. (Kokkos User's Guide). (right) Abstractions mapping.

reference : A portable SIMD primitive in Kokkos for heterogeneous architectures

Kokkos Concepts (2) - What is a device ?

- Kokkos defines several **c++ class** for representing a device in `core/src`, e.g.
  - `Kokkos::Cuda`, `Kokkos::HIP`, `Kokkos::SYCL`, `Kokkos::OpenMPTarget` `Kokkos::OpenMP`, `Kokkos::Threads`, `Kokkos::Serial`
  - **device = execution space + memory space**
- Each *Kokkos device* pre-defines some types
- Example **Kokkos exec space** (not required for a user, only Kokkos developper), e.g.

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

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

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

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

  // This execution space's preferred array layout.
  using array_layout = LayoutLeft;
  ...
} // 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`
  Cuda stream can be used Kokkos; they must be created before `Kokkos::Cuda` exec space

How to Build kokkos (1)

- **Very large combinatorics of compile options / compiler / target architecture !**
- `Kokkos` is (used to be) mostly header-only;
  examples can be build using a **standalone Makefile** (provided `Kokkos` is cloned in your home directory) or **cmake**.

1. Build with standalone Makefile, play with examples:
   - `1. mkdir $HOME/Kokkos; cd $HOME/Kokkos`
     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`
     `4. cd example/tutorial/01_hello_world`

How to Build kokkos (2)

2. **Build/install with cmake:**
   - Note on using vs integrating `Kokkos` in your own application :
     Don't try to add `Kokkos` source as a git submodule to your project (unless for quick demo ;) ⇒ deprecated
3. **Build/install Kokkos with spack:**
   - ⇒ Kokkos by design has **many different configurations possible** (hardware adaptability, heavily relies on C++ metaprograming - compile timing)
   - For each different configuration, you will have a *modulefile* to configure the environment
   - see kokkos+spack
4. Side note: 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).

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 cmake
  Kokkos::initialize(argc, argv);
  ...
  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 > HIP > SYCL > OpenMPTarget > OpenMP > Threads > HPX > Serial
    see header Kokkos_Macros.hpp
  - 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 cmake
  Kokkos::initialize(argc, argv);
  ...
  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 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 ? $data(i,j)$! *à 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) [1]
- `Kokkos::View` are designed to be pass by value to a function (**no hard copy**).

[1] 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 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.

```
// 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

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.

```
// 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.

Kokkos compute Kernels - parallel dispatch (3)

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)

Kokkos compute Kernels - parallel dispatch (4)

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 (5)

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

Kokkos compute Kernels - parallel dispatch (6)

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*

Kokkos compute Kernels - parallel dispatch (7)

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)

Kokkos compute Kernels - parallel dispatch (8)

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 (9)

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`)
  ```

Kokkos compute Kernels - parallel dispatch (10)

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()`

Kokkos compute Kernels - parallel dispatch (11)

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;
```

Kokkos compute Kernels - parallel dispatch (12)

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) {
               ...;
  });
}
```

Kokkos compute Kernels - parallel dispatch (13)

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

Kokkos compute Kernels - Advanced items

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).

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 3.16)
project(myproject CXX)

# C++11 is for Kokkos
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_EXTENSIONS OFF)

# first buid kokkos
# not recommended, but ok for demo
add_subdirectory(external/kokkos)

# build the user sources
add_executable(my_exe PRIVATE main.cpp)
target_link_library(my_exe PRIVATE Kokkos::kokkos)
```

Kokkos - cmake integration (2)

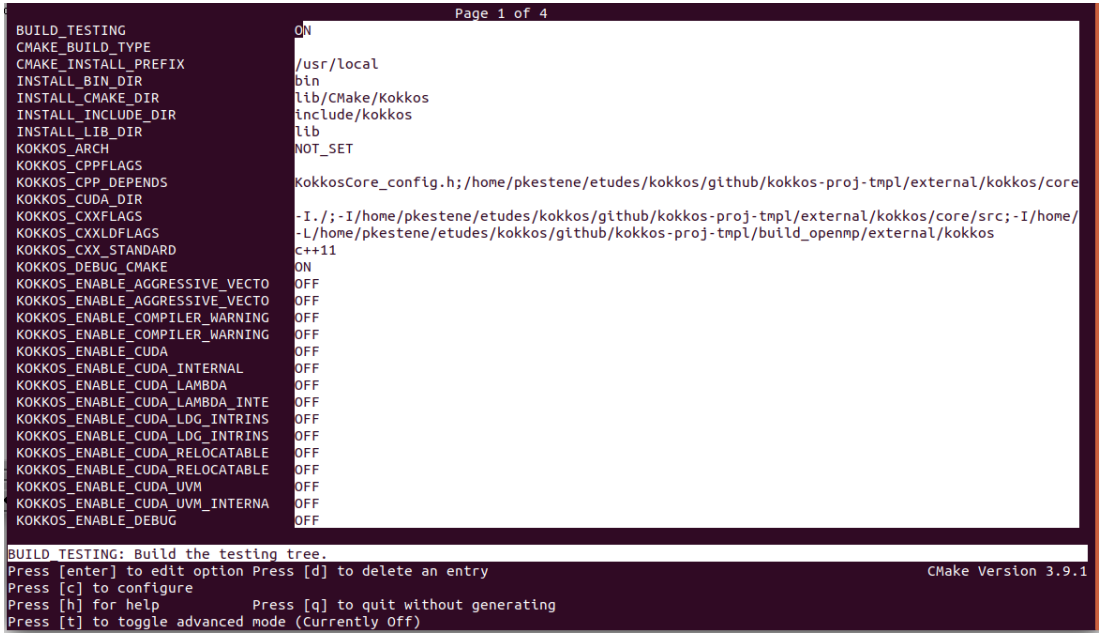
List of important kokkos-related **cmake variables**

- `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,Ampere80
# 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 ?

A template starter project

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 Turing75)

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

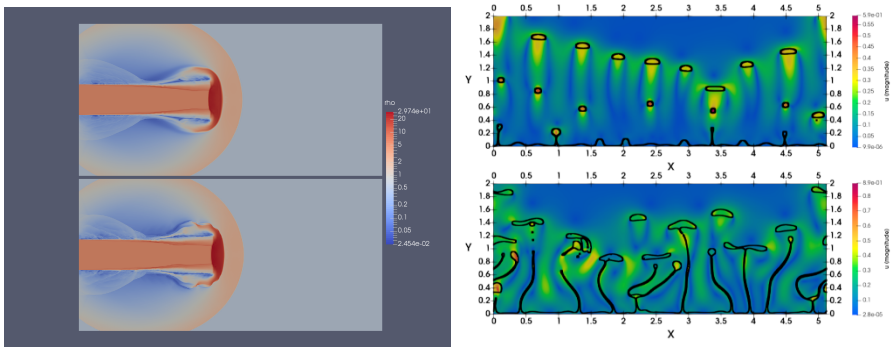
- Build the sample application (saxpy): repeat as above to setup the Kokkos CUDA target (for arch Kepler37)

```
# don't forget to set environment variable CXX
# export CXX="full path to nvcc_wrapper"
mkdir build_cuda_turing75; cd build_openmp; ccmake ..
# set KOKKOS_ENABLE_CUDA to ON; set KOKKOS_ARCH
make
```

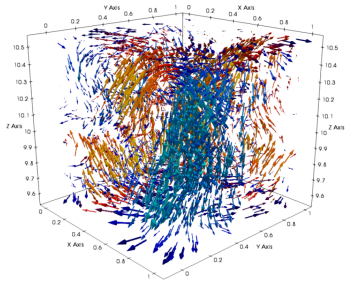
- Try to add another executable; e.g. copy of the tutorial `01_hello_world`

Scientific software devel. at MDLS using Kokkos

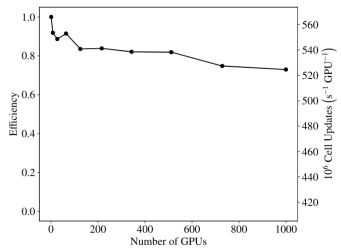
- **since 2016:** R&D using MPI+Kokkos
  mini-app EulerKokkos ⇒ ARK (**high/low Mach flow with well-balanced gravity**) by **T. Padioleau**, **P. Tremblin** (DRF/MDLS), **S. Kokh** (DES/STMF), ARK-RT (Radiative transfert)
- **2018:** LBM_Saclay with **A. Cartalade** and **A. Genty** (DES/STMF)
  PhD: **W. Verdier, T. Boutin** : Two-phase flow (Navier-Stokes + phase fields models) with phase change using **Lattice Boltzmann methods** for studying demixing process in glasses, dissolution in porous media.
- **2017-2018:** ppkMHD MPI+Kokkos implementation of high order **spectral difference method** (SDM)



(left) ppkMHD (MPI+Kokkos: high Mach (M=27) jet
(right) LBM_Saclay (MPI+Kokkos): film boiling (2019, Verdier, STMF)



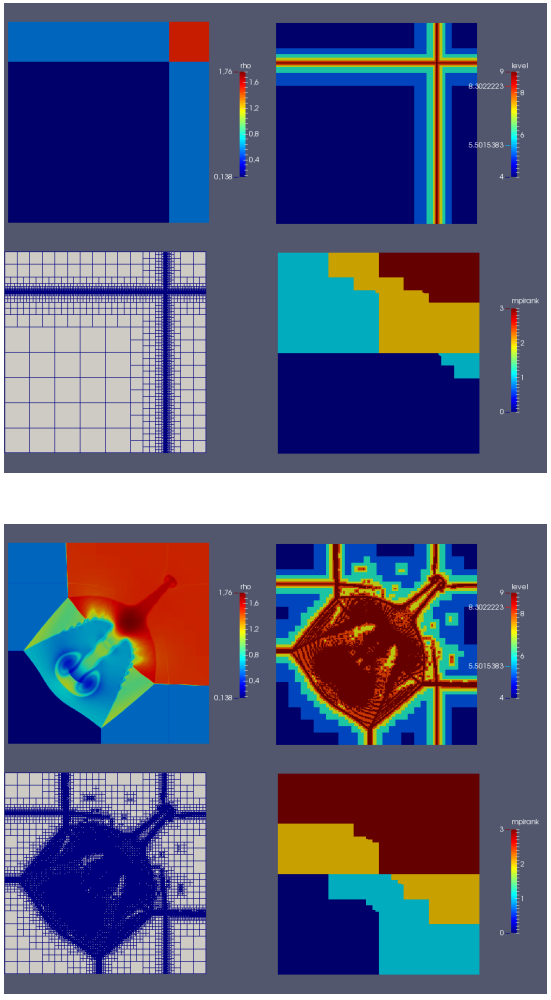
ARK Rayleigh-Benard instability (2019, Padioleau, MDLS)



ARK GPU weak-scaling, 4960^3 (2019, Daley-Yates, MDLS, Jean Zay/IDRIS)

Prototyping adaptive mesh refinement with Kokkos

- At CEA/DRF/IRFU (**A. Durocher, M. Delorme**) : **Dyablo = A C++ software platform for octree-based AMR CFD applications** using external libraries:
  - (PABLO) for distributed mesh management and AMR algorithms
  - Kokkos for shared mem. parallelism (CPU/GPU)
- Mini-app focused on single-node AMR running entirely on device (GPU) written in Kokkos for LBM applications
  - **E. Stavropoulos Vasilakis** (CEA/DES/STMF), started **02/2021**, Lattice Boltzmann numerical scheme implementation



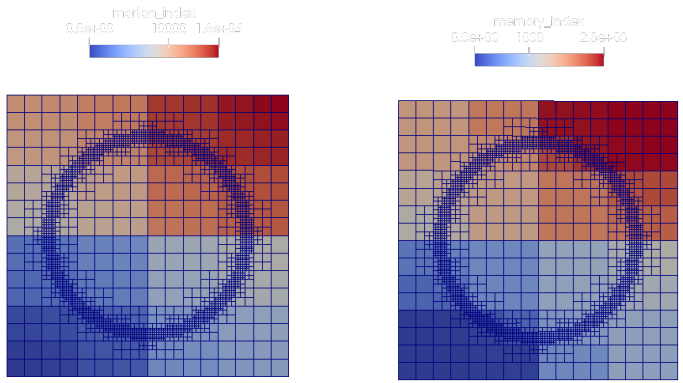
code CanoP (p4est)

Kokkos + AMR = khamr

- **Sand box / testing ideas of generic multi-architecture AMR**
- testing sorting algorithmes with Kokkos, bucket sort, ...
- testing Morton index and data containers (encode/decode, use level/tree id)

```
KOKKOS_INLINE_FUNCTION
uint64_t morton_key(const int ix, const int iy, const int iz)
{
  uint64_t key = 0;
  key |= splitBy3<3>(ix) | splitBy3<3>(iy) << 1 | splitBy3<3>(iz) << 2;
  return key;
} // morton_key - 3d
```

- testing parallel hash tables



Kokkos for Cuda users

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) );
  // ...
}
```

Kokkos for Cuda users

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 { ... };
}
```

Kokkos for Cuda users

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`)