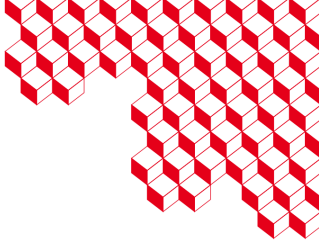


Kokkos/C++ training

Performance portability

Pierre Kestener

December 9-11th, 2024, Cerfacs, Toulouse

Funded by the European Union

Schedule

Monday, December 9th, 2023: Kokkos tutorial

- **Introduction to performance portability**, hardware trends, programing models for heterogeneous computing platforms
- GPU Computing / Cuda refresher
- supercomputer `calypso` : a CPU (Nvidia Grace ARM) + GPU (Nvidia Hopper H100) computing platform : short overview
- Hands-on 00: Get familiar with `calypso` compile environment, use `saxpy` to review CPU/GPU differences
- **C++ Kokkos: features overview**
- Hands-on 01: **retrieve Kokkos sources**, how to build with `Make` or `cmake`, explore different hardware/compiler configurations
- **Kokkos abstract concepts overview**: parallel programing patterns, data container
- Hands-on 02: **build a Kokkos app**, how to build with `Make` or `cmake`, how to integrate `kokkos` in an existing application

Schedule

Tuesday, December 10th, 2023: Kokkos tutorial

- **Kokkos abstract concepts overview (continue)**: parallel programing patterns, data container
- Hands-On 03: Revisit simple example **SAXPY**
  $\Rightarrow$ simplest computing kernel in Kokkos, performance measurement
- Hands-On 04: Simple example **Mandelbrot set**
  $\Rightarrow$ 1D Kokkos::View + linearized index (+ asynchronous execution)
- **a Kokkos miniapp skeleton project with cmake**
- Hands-On 05: Simple examples **Stencil + Finite Difference**
  $\Rightarrow$ 2D Kokkos::View
- Hands-On 05-b: **Laplace exercise**
  $\Rightarrow$ pure Kokkos versus Kokkos + MPI + hwloc (multiGPU)
- Hands-On 06: **Illustrate how to use random number generator in kokkos**
  $\Rightarrow$ RNG 101, parallel compute π with Monte Carlo
- Hands-On 07: $\Rightarrow$ use Kokkos lambda and hierarchical parallelism (Team Policy)

Schedule

Wednesday, December 11th, 2023: Kokkos tutorial

- Hands-On 08:: Kokkos and simd
- Hands-On 09:: Kokkos ecosystem python bindings and code generation: pykokkos
- Hands-On 10:: Kokkos ecosystem FLCL (Fortran Language Compatibility Layer)
- Hands-On 11:: Kokkos ecosystem for linear algebra : kokkos-kernels
- introduction to `Kokkos::tools` (integration with profiling tools, e.g. Nvidia `nsys`)
- using Kokkos and MPI. Using a simple MPI+Kokkos CFD miniApp: **Euler solver**
  $\Rightarrow$ performance measurement for several Kokkos backends (OpenMP, CUDA)

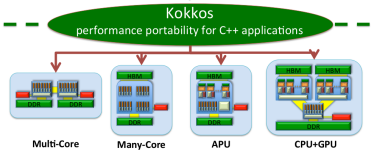
1. Introduction - Kokkos concepts

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 **c++ meta-programing** to derive native low-level code (OpenMP, CUDA, HIP, OpenAcc, Sycl, ...) and adapt data structure memory layout at compile-time
- Core developers at **Sandia NL** (lead by **C. Trott**) and at **ORNL** (lead by **D. Lebrun-Grandie**)

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

- (C++ 2023) mdspan
  reference implementation
  https://github.com/kokkos/mdspan
- (C++ 2026) std::linalg
  reference implementation
  https://github.com/kokkos/stdBLAS

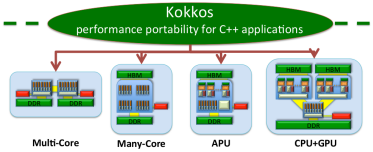


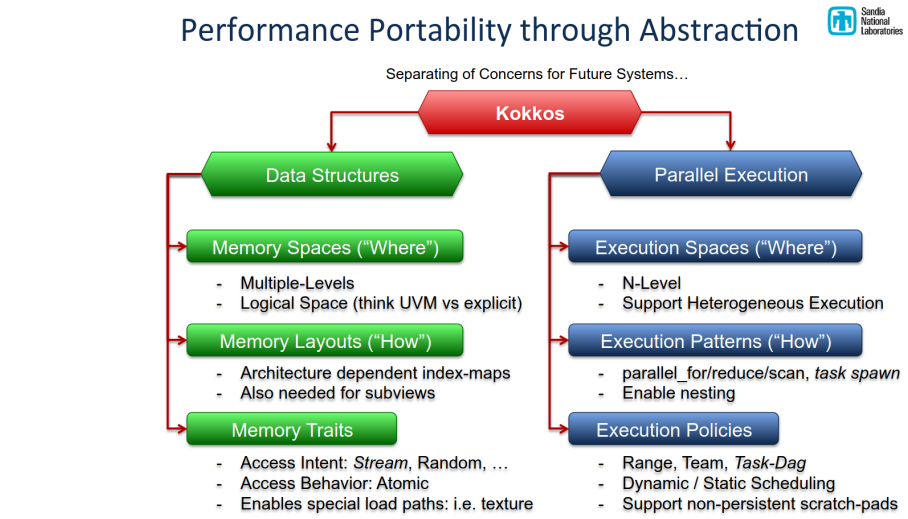
Kokkos: a programming model for perf. portability

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

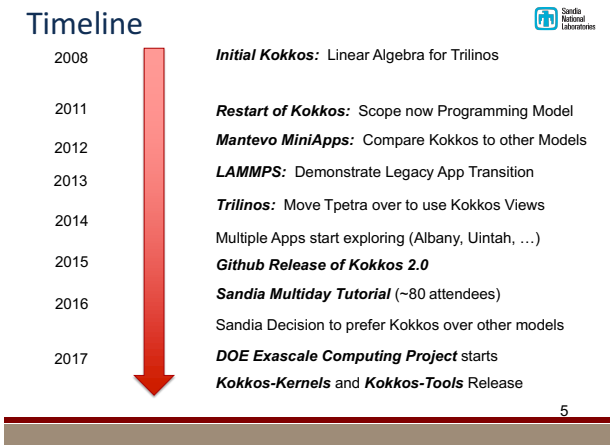
Goal: **Make ISO/C++ Standard subsumes Kokkos abstractions**

- (C++ 2023) mdspan
  reference implementation
  https://github.com/kokkos/mdspan
- (C++ 2026) std::linalg
  reference implementation
  https://github.com/kokkos/stdBLAS

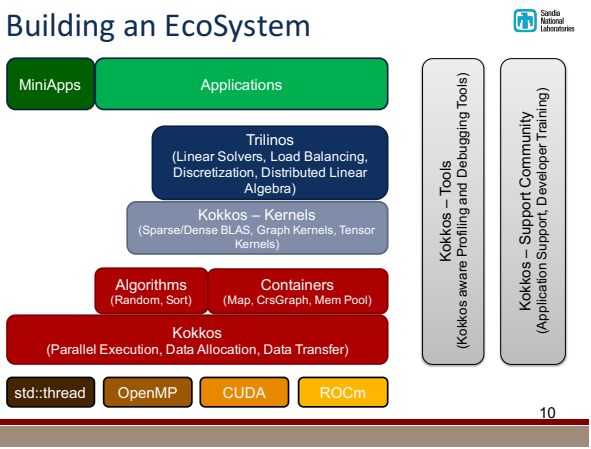




reference: https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/Kokkos-Multi-CoE.pdf



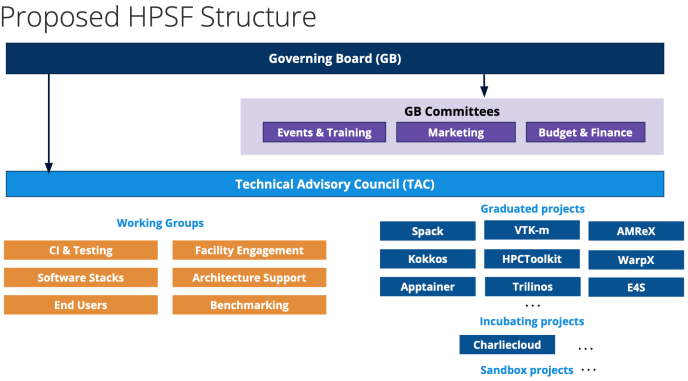
reference: https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/Kokkos-Needs-Of-Apps.pdf



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

High Performance Sorftware Foundation

- Announced at SC23 (Nov. 2023)
- https://hpsfoundation.github.io/



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) ?

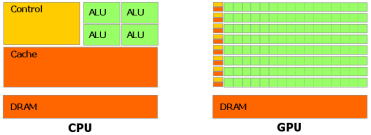
left-layout = row-major

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

Answer:
Optimize memory access pattern !

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

Different hardware ⇒
Different parallelization strategies

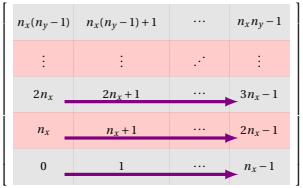


Illustrating portability with Kokkos

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

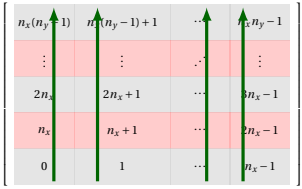
OpenMP // outer loop

```
#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

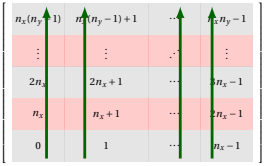
```
__global__ void compute(int *data)
{
  // adjacent memory cells
  // computed by adjacent threads
  int i = threadIdx.x + blockIdx.x*blockDim.x;

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



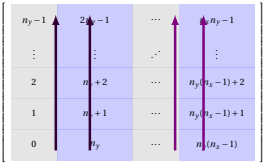
Illustrating portability with Kokkos

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

left layout / CUDA



right layout / OpenMP



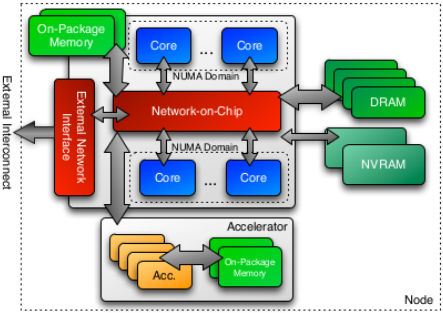
Kokkos single parallel version (CUDA+OpenMP)

- 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 a large subset of all shared-memory nodes made of
  - **latency-oriented cores** (contemporary CPU core)
  - **throughput-oriented cores** (GPU, ...)



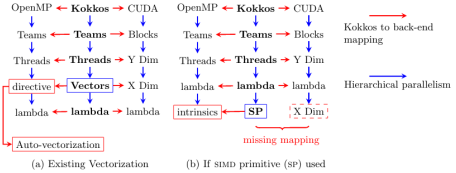


Figure 1: (left) Conceptual model of a current/future **HPC node**. (Kokkos User's Guide). (right) Abstractions mapping.

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::HIP, Kokkos::Cuda, Kokkos::OpenMP, Kokkos::Threads, Kokkos::Serial, etc...
  - **device = execution space + memory space**
- Each *Kokkos device* pre-defines some types
- Example **Kokkos device** (the definition itself is not required for a user, for a Kokkos developper only), 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) - What is a device ?

- There is always a default value for the execution space and memory space. It depends on cmake options used when building kokkos. See Hands-on 02
- The user can define a default device (if needed) like this:

```
using DefaultDevice =
Kokkos::Device<Kokkos::DefaultExecutionSpace, Kokkos::DefaultExecutionSpace::memory_space>;
```

- By default, unless explicitly specified,
  - all kernels will run on the default device
  - all memory allocation will happen on the default memory space
- You can also specified locally the pair (execution space, memory space) you want to use, provided the corresponding kokkos backend has been activated during building kokkos

Kokkos Concepts (4) - execution space, memory space

- **Memory space:** Where / how data are allocated/freed in memory (HostSpace, CudaSpace, CudaUVMSpace, CudaHostPinnedSpace, HBWSpace, ...)
  Every memory space has a default execution space.
- **Execution space:** Where should a parallel construct (`parallel_for`, ...) be executed
  - Special case: `class HostSpace`, special device (always defined) where execution space is either (Serial, Thread or OpenMP).
  - Each execution space is equipped with a `fence`: `Kokkos::Cuda::fence()`
    A parallel kernel is executed **asynchronously** on a given execution space with respect to where is was *launched*
    example: when host lauch a CUDA parallel kernel, the `Kokkos::Parallel_for` may return early (before the GPU actually finishes computation), so a call to `Kokkos::fence` might be required e.g. when doing time measurements

```
Kokkos::Timer timer;
// This operation is asynchronous, without a fence
// one would time only the launch overhead
Kokkos::parallel_for("Test", N, functor);
Kokkos::fence();
double time = timer.seconds();
```

- **Memory layout** (we will come back later on that)
- Other concepts, e.g. **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`

Kokkos Concepts (5) - pattern, policy, body

Specifying computation requires 3 ingredients

- a `pattern` : for, reduce, scan, ...

  a `execution policy` : Range, MDRange, Hierarchical (i.e. nested parallelism), ...

  a `body` : a functor or lamdba containing instructions to execute

- serial

  `for` (`int i=0; i<N; ++i`) `{ /* body */ }`

- parallel for with Kokkos

  `Kokkos::parallel_for` (`N` , `KOKKOS_LAMBDA (int i)` `{ /* body */ }`)

- `KOKKOS_LAMBDA` is a preprocessor macro for portable lambda function;
  - on host: `#define KOKKOS_LAMBDA [=]`
  - on gpu: `#define KOKKOS_LAMBDA [=] __host__ __device__`
  - $\Rightarrow$ lambda functions capture variables by **copy** (very important, more later)

Hands-On 01: Build kokkos (1)

Follow instructions from exercises/01-build-kokkos/Readme.md
Purpose:

- Clone kokkos git repository
- Get familiar with compile architecture flags and device backends
- Explore different ways of building Kokkos: standalone Makefile or cmake
- Build and run a simple kokkos application

I. Get Kokkos sources, use latest release 4.4.01

Practicals on calpyso:

1. use salloc to allocate a grace node: `salloc -p grace -N 1 --gres=gpu:1 -n 24`
2. `mkdir $HOME/install; cd $HOME/install`
   some kokkos tutorial examples have a Makefile configured for using that precise location.
3. `git clone https://github.com/kokkos/kokkos`
4. `cd kokkos; git checkout 4.4.01`

Hands-On 01: Build kokkos (2)

List of supported compilers (current Kokkos develop branch, some compiler can be used for multiple backends, see cmake/kokkos_compiler_id.cmake, line 161):

Clang(CPU)	8.0.0 or higher
Clang(CUDA)	10.0.0 or higher
GCC	8.2.0 or higher
Intel	19.0.5 or higher
IntelLLVM(CPU)	2021.1.1 or higher [1]
IntelLLVM(SYCL)	2023.0.0 or higher [2]
NVCC	11.0.0 or higher
HIPCC	5.2.0 or higher
NVHPC/PGI	22.3 or higher
MSVC	19.29 or higher
XL/XLClang	not supported

For hands-on on `calypso`, we will use gcc, nvcc and nvhpc compilers.

[1] but version 2023.1.0 is prefered if you want vectorize flags (see Kokkos_Macros.hpp, line 180)

[2] You also install Codeplay OneAPI plugin for Nvidia'GPU. You'll get a `clang++` compiler able to target `nvptx64-nvidia-cuda`

Hands-On 01: Build kokkos (3)

II. How to build and use Kokkos library There are different ways:

- build and install `kokkos`; then build use application
- build `kokkos` and user application together

Each of these options can be done either with cmake or plain Makefile's

1. **Use CMake (recommended)**
   - **Cmake is Kokkos's primary build system**
   - ⇒ Kokkos by design has **many different configurations possible** (hardware adaptability) : support many compiler toolchains and hardware backends
   - **Two ways of using Kokkos with cmake:**
     - Kokkos can be compiled and installed as a regular library
     - Kokkos sources can be embedded in user application; user application's cmake build will build kokkos first, and user application after
   - ⇒ it is important to understand that kokkos build flags and your application build flags should/must be the same (be consistent)
   - ⇒ **best practice and recommended use**: use a cmake macro in your application build system that **allow either to detect an installed version of Kokkos or to trigger building kokkos** (we'll see a skeleton app for that during hands-on session)
   - Additionnal slides on CMake/kokkos :
     (https://github.com/kokkos/kokkos-tutorials/blob/main/Intro-Full/Slides/KokkosTutorial_CMake.pdf

II. How to build and use Kokkos library

2. Use standalone Makefiles.kokkos (good for rapid testing):

This way of doing is for building both Kokkos and user application together.
If the final user application `Makefile` includes `Makefile.kokkos` (located at top level of kokkos sources), then the application and kokkos itself will be built together.
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. Power9, Ampere100, SNB, SKX, ARMv80, ROCm, ...

exercise:

- Go into kokkos sources; `cd example/tutorial/01_hello_world`
- `make -j 8 KOKKOS_DEVICES=OpenMP KOKKOS_ARCH=ARMV9_GRACE`
- `make -j 8 KOKKOS_DEVICES=OpenMP,Cuda KOKKOS_ARCH=ARMV9_GRACE,Hopper90`

This way of building kokkos and application is convenient but you have to explicit all options (e.g. no Cuda hardware detection, need to specify `KOKKOS_ARCH`)
All Kokkos examples (inside Kokkos sources) can be built that way, as well as Kokkos-tutorials

II. How to build and use Kokkos library

3. Use GNU generated Makefiles:

- **Main use: build Kokkos itself**
- use `gnu_generate_makefile.bash`, then `make -j 8; make install`
  Then a *modulefile* can be used to configure the environment for build user application

Example:

```
# ------------------------------------------------------
mkdir -p _build/openmp; cd _build/openmp
COMPILER=g++ ../../gnu_generate_makefile.bash --with-devices=OpenMP --arch=ARMV9_GRACE
↪  --prefix=$HOME/local/kokkos-openmp
make -j 8
# ------------------------------------------------------
export CXX=nvcc
mkdir -p _build/cuda; cd _build/cuda
../../gnu_generate_makefile.bash --with-devices=OpenMP,Cuda --arch=AMRV9_GRACE,Hopper90
↪  --prefix=$HOME/local/kokkos-cuda
make -j 8
```

II. How to build and use Kokkos library

4. **Use generated Makefiles (cmake wrapper):**
   - **Main use: build Kokkos itself**
   - Same commandline as before; e.g.

```
../../generate_makefile.bash --with-devices=OpenMP --arch=ARMV9_GRACE
↪  --prefix=$HOME/local/kokkos-openmp}
```

   - but will use cmake underneath to generate Makefile's
5. Kokkos can also be built and installed through spack package manager; see instructions inside Kokkos sources : https://github.com/kokkos/kokkos/blob/master/Spack.md
6. finally, 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).

Hands-On 01: Build kokkos (7)

Example build configurations using generated Makefiles (kind of deprecated, prefer using cmake)

- **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)**

```
../generate_makefile.bash --with-cuda --arch=Hopper90
 --prefix=$HOME/local/kokkos_cuda_openmp --with-cuda-options=enable_lambda
 --with-openmp --with-hwloc=/usr
```

Hands-On 01: Build kokkos (7)

- **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. (**recommended**) Use your own cmake-based build system : ease of configuring/exploring the large combinatorics of `DEVICES`, `ARCH`, compilers, compiler options, ...
  2. Use / adapt an existing Makefile from Kokkos tutorial, examples, ...

- additionnal slides : KokkosTutorial_CMake.pdf

  (https://github.com/kokkos/kokkos-tutorials/blob/main/Intro-Full/Slides/KokkosTutorial_CMake.pdf)

Kokkos - Documentation

- **Most up to date and recommended documentation is the wiki:**
  https://kokkos.github.io/kokkos-core-wiki/ (programming guide)
- Kokkos new website : https://kokkos.org/
- Kokkos source code itself, espacially reading unit tests code is very helpful
- Doxygen can also be useful; it can only be built from inside Trilinos source tree
  but version of the day can be browsed at
  https://trilinos.org/docs/dev/packages/kokkos/doc/html/index.html (though, much less convial as the wiki)

Additionnal resources:

- Tutorial slides and codes:
  https://github.com/kokkos/kokkos-tutorials/LectureSeries
- Tutorial videos:
  https://github.com/kokkos/kokkos-tutorials/wiki/Kokkos-Lecture-Series

Kokkos - initialize / finalize

https://kokkos.github.io/kokkos-core-wiki/ProgrammingGuide/Initialization.html

- `Kokkos::initialize / finalize`

```
#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 or generate_makefile.bash
  Kokkos::initialize();
  ...
  Kokkos::finalize();
}
```

- **What's happening inside `Kokkos::initialize`**
  - Defines DefaultExecutionSpace (as specified when kokkos itself was built, by order of **priority**: Cuda > OpenMPTarget > HIP > SYCL > OpenACC > OpenMP > Threads > HPX > Serial)
    e.g. if `--with-cuda` was not pass to `generate_makefile.bash`, but `--with-openmp` was, then `DefaultExecutionSpace` is OpenMP; see definition in Kokkos_Core_fwd.hpp
  - Defines DefaultHostExecutionSpace (can be e.g. OpenMP, Serial, ...). Serial is default value, if no other execution space activated
  - You can activate several execution spaces (recommended); one for device, one for host (can be identical, e.g. OpenMP)
  - all this information provided at compile time will internally be used inside Kokkos sources as default (hidden) template parameters

Kokkos - initialize / finalize

https://kokkos.github.io/kokkos-core-wiki/ProgrammingGuide/Initialization.html

- Kokkos::initialize / finalize

```
#include <Kokkos_Core.hpp>

int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  // Kokkos parallel region usually nested inside MPI
  // rare exception: on specific hardware, Kokkos might need to be initialized first
  Kokkos::initialize();
  ...
  Kokkos::finalize();
  MPI_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 if one wants to control device id, threads number, etc...
  Recommended to build kokkos with hwloc (Kokkos will use hwloc to probe hardware, and chose wise default values)
- **Example using command line argument**:
  `./my_kokkos_app --help; ./my_kokkos_app --kokkos-threads=10`
- You can also specify the number of threads, or control the CUDA device id by using environment variables like `OMP_NUM_THREADS` or `CUDA_VISIBLES_DEVICES`
- If you run on a supercomputer, usually the job scheduler will set OpenMP env variables or GPU/task mapping for you.

Kokkos - initialize / finalize with MPI

- Cross-control at runtime that kokkos initialized the resources you wanted.

```
void
print_kokkos_config()
{
  // only master MPI task print Kokkos config information
  if (rank() == 0)
  {
    std::cout << "##########################\n";
    std::cout << "KOKKOS CONFIG             \n";
    std::cout << "##########################\n";
    std::ostringstream msg;
    std::cout << "Kokkos configuration" << std::endl;
    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();
    std::cout << "##########################\n";
    std::cout << "END KOKKOS CONFIG         \n";
    std::cout << "##########################\n";
  }
}
```

Kokkos - initialize / finalize with MPI

- When using **MPI + Kokkos/CUDA**, you may want to cross-check MPI process rank with GPU device id mapping (we will come back into that with example code)

```
// on a large cluster, the scheduler should assign ressources
// in a way that each MPI task is mapped to a different GPU
// let's cross-checked that:

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

- In any case, **cross-check this information** with the job scheduler, e.g.

```
mpirun --report-bindings
```

Hands-On 02 : Kokkos `query_device` with hwloc using cmake

Purpose:

- just cross-checking Kokkos/Hwloc is working OK
- get familiar with the cmake build system
- get familiar with using kokkos via a modulefile
- get familiar with the notion of `DefaultExecutionSpace`

Follow instructions from exercises/02-build-a-kokkos-app-with-cmake/Readme.md

Hands-On 02 : Kokkos `query_device` without hwloc

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 - 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; assuming Kokkos already installed on your system

```
cmake_minimum_required(VERSION 3.18)
project(myproject CXX)

# C++17 is for Kokkos
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_EXTENSIONS OFF)

# find kokkos
find_package(kokkos 4.4.01 REQUIRED)

# build the user sources
add_executable(saxpy saxpy.cpp)
target_link_libraries(saxpy Kokkos::kokkos)
```

Kokkos - cmake integration (2)

- You can also chose to build kokkos, alltogether with your application. For that, we fetch kokkos sources, and integrate them into the application build system.

```
cmake_minimum_required(VERSION 3.18)
project(myproject CXX)

# C++17 is for Kokkos
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_EXTENSIONS OFF)

# fetch kokkos sources: it can be a git repo, a tarball archive, etc...
# see FetchContent_Declare documentation
FetchContent_Declare( kokkos_external
    GIT_REPOSITORY https://github.com/kokkos/kokkos.git
    GIT_TAG 4.4.01
)
FetchContent_MakeAvailable(kokkos_external)

# build the user sources
add_executable(saxpy saxpy.cpp)
target_link_libraries(saxpy PRIVATE Kokkos::kokkos)
```

Kokkos - cmake integration (3)

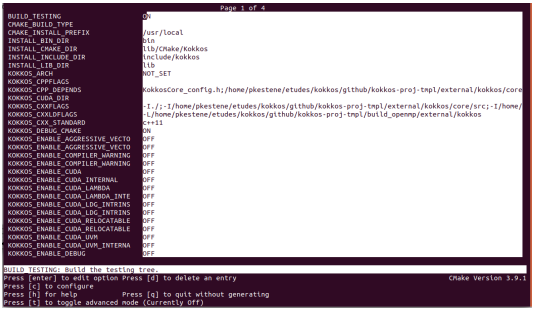
List of important kokkos-related **cmake variables**

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

  # Intel: SNB,HSW,BDW,SKL,SKX,ICL,ICX,SPR
  # NVIDIA: Kepler,Kepler30,Kepler32,Kepler35,Kepler37,Maxwell,
  Maxwell50,Maxwell52,Maxwell53,Pascal60,Pascal61,
  Volta70,Volta72,Turing75,Ampere80,Ampere86,Ada89,**Hopper90**
  # ARM: **ARMV9_GRACE**,A64FX,ARMv80,ARMv81,ARMv8-ThunderX,ARMv8-ThunderX2
  # IBM: BGQ,Power7,Power8,Power9
  # AMD-GPUS: GFX906,GFX908,GFX90A,GFX942,GFX1030,GFX1100
  # AMD-CPUS: AMDAVX,Zen,Zen2,Zen3

Kokkos - cmake integration (4)

- curse gui interface: `ccmake`



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

Hands-On 02 - complement

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 Hopper90)

```
mkdir -p _build/openmp; cd _build/openmp;
cmake -DKOKKOS_PROJ_TMPL_BACKEND=OpenMP ../..
make
```

- Build the sample application (saxpy): repeat as above to setup the Kokkos CUDA target (for arch Hopper90)

```
mkdir _build/cuda; cd _build/cuda;
cmake -DKOKKOS_PROJ_TMPL_BACKEND=Cuda -DKokkos_ARCH_HOPPER90=ON ../..
make
```

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

2. Kokkos - data containers and threads dispatch

Kokkos data Container (1)

`Kokkos::View<...>` is multidimensionnal data container [3] 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 **lightweight**, designed wrapping allocated memory buffer
  - View = pointer to data + metadata(array shapes, 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**).

[3] https://kokkos.org/kokkos-core-wiki/ProgrammingGuide/View.html#view-multidimensional-array

[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. Better for vectorization (SIMD).
- `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
- Most generic interface

```
template <class DataType [, class LayoutType] [, class MemorySpace] [, class MemoryTraits]>
class View;
```

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
  -
```
Kokkos::View<double*> a("a",100000); // allocated in default MemSpace
auto a_h = Kokkos::create_mirror_view(a);
Kokkos::deep_copy(a_h,a);
// 1. if default ExecSpace is e.g. Cuda, it will allocate a_h on host, and copy data
// 2. if default ExecSpace is a HostSpace (OpenMP, ...), it does nothing
// a and a_h reference the memory
//
// equivalent to above, but in one line:
auto a_h = Kokkos::create_mirror_view_and_copy(a);
```

Kokkos data Container (4)

A more complete way of declaring a `Kokkos::View`:

- **Kokkos::View** declaration example of a 1D array of doubles:
  `Kokkos::View<double*,Kokkos::LayoutLeft,Kokkos::CudaSpace> a("a",100);`
  - **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)
- Declaring a view accessible from both CPU and GPU (allocated in unified memory):
  `Kokkos::View<int*, Kokkos::SharedSpace> a("a",100);`
  Note that `Kokkos::SharedSpace` is an alias to either `CudaUVMSpace`, `HIPManagedSpace`, ...
  Ok, but is the default memory layout ? For `CudaUVMSpace`, associated execution space is `Cuda`, so it is `Kokkos::LeftLayout`
- `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 exercise about Mandelbrot set.

Kokkos data Container (5)

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

```
#include <type_traits>

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

Kokkos data Container (6)

1. **A View's entries are initialized to zero by default.**
2. You may decide to allocate a view without initializing its entries: useful when you know that you will initialize data in a kokkos parallel region (first touch placement policy [5])
3. a view can be resized (preserve content, the old view may/may not be deleted)
4. a view can even be reallocated (do not preserve content)

```
// 1. initialize entries with non-zero constant values
auto v = Kokkos::View<int*>("myview", 50);
Kokkos::deep_copy(v, 2.0);

// 2. allocate a view but do not initialize
Kokkos::View<int*> v2 (Kokkos::view_alloc(Kokkos::WithoutInitializing, "v2"), 100000);

// 3. resize
Kokkos::resize(v2, 500);

// 4. reallocate
Kokkos::realloc(Kokkos::WithoutInitializing, v2, 2500);
```

[5] see https://www.openmp.org/wp-content/uploads/SC18-BoothTalks-vanderPas.pdf

Kokkos data Container (7)

Some useful utilities for creating a mirror view on host of a given device view, and copy its data. host view has same memory layout as it device counter part.

```
// 1. always allocate host view and actual memory copy
auto data_h = Kokkos::create_mirror(data_d);
Kokkos::deep_copy(data_h, data_d);

// 2. maybe allocate host view and copy data (maybe)
// allocation only happens if the device view is not in Kokkos::HostSpace
auto data_h = Kokkos::create_mirror_view(data_d);
Kokkos::deep_copy(data_h, data_d);

// 3. (= 1+2)
auto data_h = Kokkos::create_mirror_view_and_copy(data_d);
```

Kokkos data Container (8)

Other interesting types

- static size `Kokkos::Array` (kind of equivalent to `std::array`, but device compatible)
- 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 * ptr, int size);`
  how to retrieve a raw pointer from a `Kokkos::View<int *> array`:
  `int *raw_ptr = array.data()`
  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
- Can also be used to interact with third party libraries

```
// 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);
```

Kokkos data Container (10)

Kokkos::subview

- How to create a subview of an existing Kokkos::View ?
- the subview will point to the same data as the original view: **no memory allocation, no memory copy**
- strided subview not currently supported
- ranges of indices are semi-open
- type of a Kokkos::subview may not be easy to know beforehand; dependding of the range bounds, it may be a Kokkos::View, with Kokkos::LayoutStride or a regular memory layout (Left/right)
- useful to slice data (e.g. extract slice for MPI send/recv operations, need to be careful about memory contiguity)

```
const size_t N0 = ...;
const size_t N1 = ...;
Kokkos::View<double**> A ("A", N0, N1);

// if on device, can use Kokkos::make_pair
auto A_sub  = Kokkos::subview (A, std::make_pair (2, 4), std::make_pair (3, 7));

// take all the lines for columns 3,4,5 and 6
auto A_sub2 = Kokkos::subview (A, Kokkos::ALL(), std::make_pair (3, 7));
```

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 computation but with a function pointer

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

// use a function pointer
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 computation but with a **function object (functor)**

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

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

// use a function pointer
void do_a_for_loop(std::vector<double>& data, DoSomething const & 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 computation but with a **lambda** : lambda = shorthand for a functor, context is captured from the surrounding code.

```
// use a function pointer
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
  - a name (`std::string`, optional but useful for debug and profiling)
  - **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
- Definition is `Kokkos_Parallel.hpp#L133`
- Very important: launching a kernel (thread dispatching) is by default **asynchronous**

```
// interface
template<class ExecPolicy, class FunctorType>
Kokkos::parallel_for(const std::string &name,
                     const ExecPolicy &policy,
                     const FunctorType &functor);
```

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 ("multiply_by_2", 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 if CUDA is enabled
- NB: In the code above, using integer `100` for execution policy is a short hand for `Kokkos::RangePolicy<>(0,100)`

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 functor(some_data); // create a functor instance
Kokkos::parallel_for ("multiply_by_2", 100, functor); // 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 ! It depends.

1. **Use Lambda functions.**
   - easy way for small compute kernels
   - For GPU, requires Cuda 7.5 (12.3 is current and latest CUDA version)
2. **Use a C++ functor class.**
   - More flexible, allow to design more complex kernels
3. **Programing constraints when designing a Kokkos lambda/functor:**
   - remember that a lambda/functor can be copied to device memory space: all methods are `const`, all data member must be copyable (Kokkos::View are designed that way); it is generally not valid to have pointer or reference members inside a kokkos functor.
   - A kokkos lambda/functor can be passed as a const object (however, e.g. the content of the Kokkos::View data member can be modified).
   - see https://kokkos.github.io/kokkos-core-wiki/ProgrammingGuide/ParallelDispatch.html

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`:

```
uint32_t              final_sum = 0;
Kokkos::Sum<uint32_t> reducer(final_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 ("do_sum_reduce", 100,
   KOKKOS_LAMBDA (const int i, int &local_sum) {
     local_sum += data(i);
   },
   reducer);
```

- Important note: `Reducer` is a c++ class instance that take by reference variable `final_sum` as a payload. It handles which type of reduction, we want to use. Reduction type can be: `Sum`, `Prod`, `Min`, `Max`, ... see https://kokkos.github.io/kokkos-core-wiki/ProgrammingGuide/Custom-Reductions-Built-In-Reducers.html
- If you want to reduce more complex data structure, you need to specify a **functor** with special member function:
  - `init`: how the local result is **initialized** (default 0)
  - `join`: how the different intermediate results are **joined**

About Kokkos::parallel_reduce with a functor

- Kokkos supplies a `init` / `join` operator implementation for a large number of reduction types ()
- If the reduce operator is not trivial (i.e. not built-in), or using complex data struct to reduce ⇒ you need to define a custom reducer, i.e. a class with methods `init` and `join`

```
class CustomReducer {
  private:
  value_type& value;

  public:
  KOKKOS_INLINE_FUNCTION
  CustomReducer(value_type& value_) : value(value_) {}

  // how each thread initializes its reduce result
  KOKKOS_INLINE_FUNCTION void
  init(value_type &val) const {...}

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

}
```

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

About Kokkos::parallel_reduce with a functor

- Using custom reducer

```
value_type final_value;

Kokkos::parallel_reduce(
"custom_reduction", Kokkos::RangePolicy<>(0,N),
KOKKOS_LAMBDA(const int i, value_type& update) {
  // do custom reduction (compute new update)
},
CustomReducer(final_value));
```

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:** mapping multi-dimensional range of iterations.
  See KokkosTutorial_03_MDRangeMoreViews.pdf

```
// create the MDrangePolicy object
auto range = Kokkos::MDRangePolicy< Kokkos::Rank<2>>( {0, 0}, {N0, N1} );

// use a special multidimensional parallel for launcher
Kokkos::parallel_for("some_computation", range, functor);
```

Parallel dispatch - execution policy

see also: KokkosTutorial_04_HierarchicalParallelism.pdf

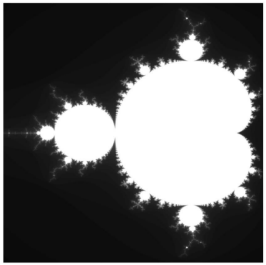
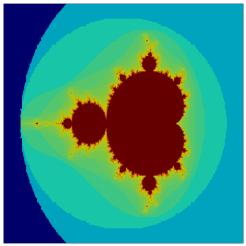
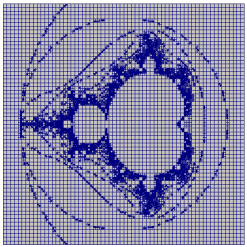
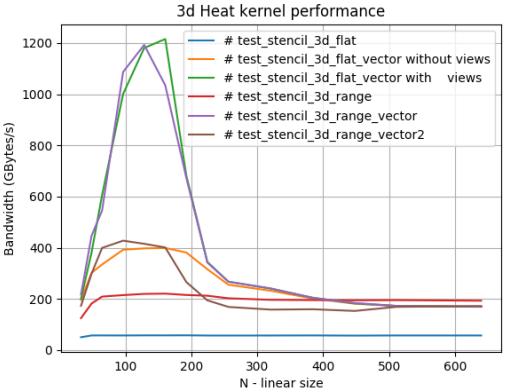
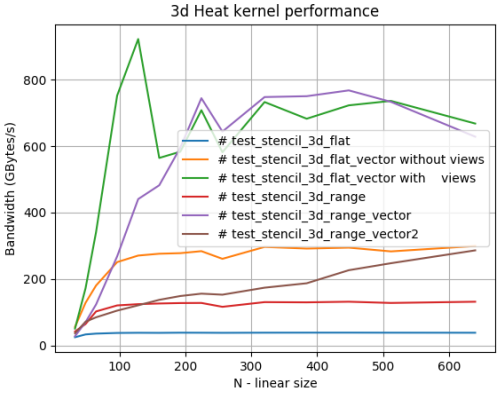
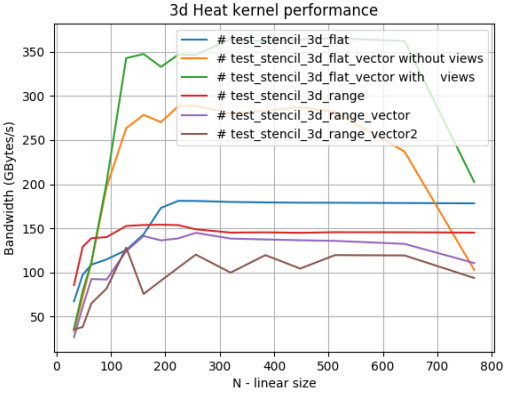
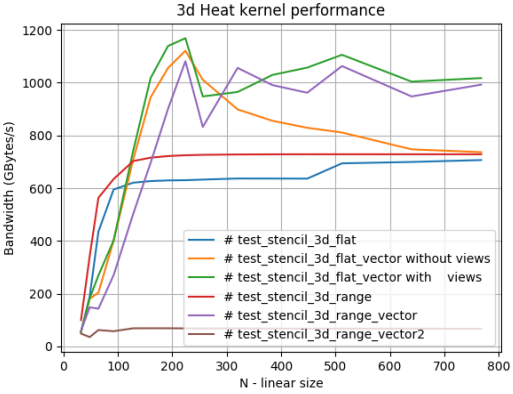
- **Team policy:** for **hierarchical parallelism**
  - threads team
  - threads inside a team
  - vector lanes
- ```
  // Using default execution space and launching
  // a league with league_size teams with team_size threads each
  Kokkos::TeamPolicy <>
  policy( league_size , team_size );
  ```

  equivalent to launching in CUDA a 1D grid of 1D blocks of threads.
  Team scratch pad memory $\Longleftrightarrow$ CUDA shared memory
- Lambda interface changed:

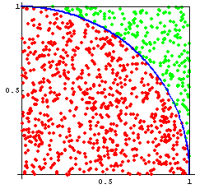
  ```
  KOKKOS_LAMBDA (const team_member& thread) { ...};
  ```

  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();`)
  $\Rightarrow$ `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 team
  // 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

See also : KokkosTutorial_05_SIMDStreamsTasking.pdf

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` $\Rightarrow$ 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 can help the compiler understand we are accessing memory with a stride 1 (assuming layout right, which the default for OpenMP device).

Hands-On 03 : SAXPY

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

- There 3 differents versions
- **1. Serial : no Kokkos)**
- **2. OpenMP : no Kokkos)**
- **3. Kokkos-Lambda : Kokkos with lambda for threads dispatch and data buffer (`Kokkos::View`)**

Proposed activity (get the sources):

1. Unless already done, make sure you cloned kokkos sources inside `${HOME}/install` (probably already done):

   ```
   mkdir -p ${HOME}/install; cd ${HOME}/install
   ```

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

2. From the provided material `cd exercises/03-saxpy`

   edit `exercises/03-saxpy/Readme.md`

Hands-On 03 : SAXPY

Proposed activity:

- **Saxpy serial (just for reference)**
  - `cd exercises/03-saxpy/Serial`
- **Saxpy regular OpenMP (also for reference)**
  - `cd exercises/03-saxpy/OpenMP`
- **Saxpy Kokkos** [6]

  See instructions in `exercises/03-saxpy/Kokkos-Lambda/Readme.md` for build instructions
  - `cd handson/03/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 ;
}

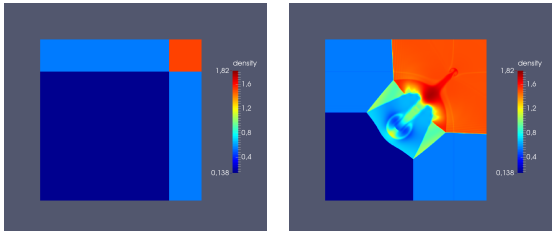
Kokkos::print_configuration( msg );
std::cout << msg.str();
```

[6] Make sure to use a very large data array.

3. Hands-on exercises

Hands-On 04 : Mandelbrot set

- **Illustrate Functor class + 1D `Kokkos::View` + linearized index**
- the original serial code use 1D `std::vector<unsigned char>` data with linearized index, i.e. $index = i + Nx * j$
- See serial code from `code/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 04 : Mandelbrot set

- Use code from directory `exercises/04-mandelbrot`; 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_hopper90`
  - `export CXX=/full/path/to/nvcc_wrapper`
  - `cmake -DKOKKOS_ENABLE_CUDA=ON -DKOKKOS_ARCH_HOPPER90=ON ..`
  - `make`
- **Compare performance** for a large Mandelbrot set 8192×8192 : OpenMP versus Cuda

Hands-On 04 : Mandelbrot set

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

  ```
  Kokkos::MDRangePolicy< Kokkos::Rank<2>>;
  ```

- **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 04 : 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 04 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





Hands-On 05 : Finite Difference / Stencil

- **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 `exercices/05-laplace`

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 (1)

- Perform **distributed computing** on a cluster with several GPU per node
- **How to build an MPI application when `KOKKOS_DEVICE` is Cuda ?**
  - Solution 1 (**recommended**): use with `find_package(MPI)` and `find_package(Kokkos)`, and everything will be ok
  - Solution 2: Use `mpicxx` and pass env variable `OMPI_CXX=nvcc_wrapper` [7]
  - Solution 3: 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";
```

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

MPI + Kokkos (2)

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

```
#!/bin/bash
#SBATCH -J test_mpi_kokkos_openmp                    # Job name
#SBATCH -N 2                                         # number of nodes
#SBATCH -n 2                                         # total number of MPI task
#SBATCH -c 8                                         # number of CPU cores per task
#SBATCH -o test_mpi_kokkos_openmp.%J.out             # stdout filename
#SBATCH --partition grace
#SBATCH --gres=gpu:1

# Set OMP_NUM_THREADS to the same value as -c
# with a fallback in case it isn't set.
# SLURM_CPUS_PER_TASK is set to the value of -c, but only if -c is explicitly set
if [ -n "$SLURM_CPUS_PER_TASK" ]; then
omp_threads=$SLURM_CPUS_PER_TASK
else
omp_threads=1
fi
export OMP_NUM_THREADS=$omp_threads
export OMP_PROC_BIND=spread
export OMP_PLACES=threads

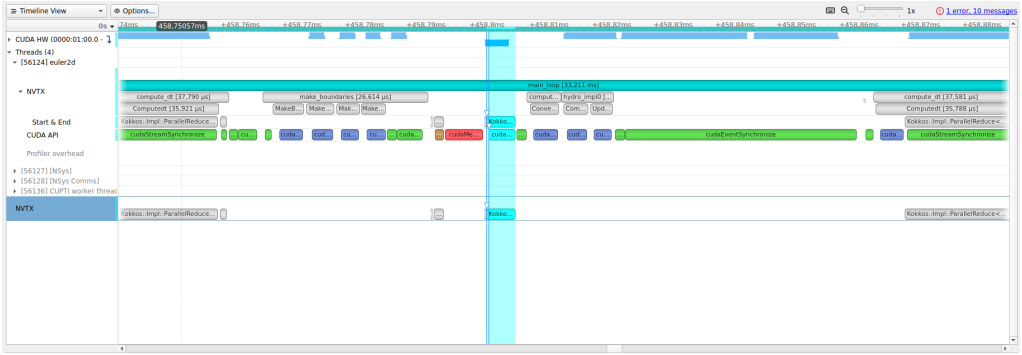
# report bindings for cross-checking
mpirun --report-bindings ./mpi_kokkos
```

About Slurm (job scheduler)

- Use code in `exercises/mpi_kokkos`; This application just reports bindings
- **Try to build this application against an installed version of Kokkos**, i.e. either OpenMP / Cuda
  - Follow the instructions from Readme.md
- Open and read `submit_calypso_cpu.sh` / `submit_calypso_gpu.sh`
- **Submit a job, read the output and check everything is what is expected**
- Slurm commands to know:
  - **submit:** `sbatch submit_calypso_cpu.sh`
  - **info/status:** `squeue -u <your user_name>`
  - **cancel/kill:** `scancel job_id`

Hands-On 05 : Laplace solver with KOKKOS + MPI

Slightly adapted/refactored from Nvidia's OpenACC exercise:
nvidia-advanced-openacc-course-sources
We will use code from `exercises/05-laplace`, 3 different versions of the 2D Laplace solver:

- serial (no kokkos)
- **kokkos with 2D views**
- **kokkos_mpi with MPI+CUDA and hwloc**

Hands-On 06 : 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 $\Leftrightarrow$ one functor
  One rng_state $\Leftrightarrow$ (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 ... ;
  }
};
```

Hands-On 06 : Random number generator with Kokkos

`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/cerfacs-training-kokkos/exercises/miniapp/fisher
```

- **This material contains multiple versions** of a **Reaction-Diffusion PDE solver (Fisher equation used e.g. in population 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/cerfacs-training-kokkos/exercises/miniapp/fisher/serial/miniapp.pdf`
   - Explore the serial version of the Fisher solver.
2. These **Kokkos exercises** are routed to use cmake, and modulefiles to load kokkos environment
   - 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/fisher/kokkos/cxx/Readme.md`
   - 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 and then Cuda

Euler equation solver

- 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 ?**



4. Additionnal Kokkos material

What does actually do Kokkos ?

What happends under the hood when compiling this code ?

```
Kokkos::parallel_for ("compute",
   Kokkos::RangePolicy<>(0,N),
   KOKKOS_LAMBDA (const int i) {
     data(i) = 2*i;
   });
```

- `Kokkos::parallel_for` is actually a templated c++ function, that just wraps a instance of class `Kokkos::Impl::ParallelFor`; each backend must provide an implementation
- the *most* important tempalte parameter is the execution policy
- the execution policy type allows the compiler to chose which overload to instanciate.
- there are actually different slightly different overload implementations; one for `RangePolicy`, `MDRangePolicy`, `TeamPolicy`, ...

What does actually do Kokkos ?

Using OpenMP as an example for Kokkos backend, the following class will actually be instanciated:

```
namespace Kokkos {
  namespace Impl {
    // slightly simplified for clarity
    template <class FunctorType, class... Traits>
    class ParallelFor<FunctorType, Kokkos::RangePolicy<Traits...>, Kokkos::OpenMP> {
      // ...
      execute_parallel() const {
        #pragma omp parallel for schedule(dynamic KOKKOS_OPENMP_OPTIONAL_CHUNK_SIZE) \
        num_threads(m_instance->thread_pool_size())
        KOKKOS_PRAGMA_IVDEP_IF_ENABLED
        for (auto iwork = m_policy.begin(); iwork < m_policy.end(); ++iwork) {
          exec_work(m_functor, iwork);
        }
      }
      // ...
    }
  }
}
```

- there is actually two variant of `execute_parallel` one for static, one for dynamic scheduling;
- this is *regular* OpenMP code.
- Using the same approach, Kokkos will provide different impl. of function `Kokkos::parallel_for` for all supported backends (Cuda, HIP, OpenMP, OpenMPTarget, SYCL, ...)

Using an installed Kokkos

- As you will surely **use multiple versions** of Kokkos (OpenMP, Cuda, ...), with/without Lambda, UVM, different compilers, debug, etc ... it is usefull to use some **modulefiles** for handling different version of Kokkos.
- 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 the companion code folder `modulefiles` (designed for `calypso` supercomputer); you can easily adapt these modulefiles to your own platform.

Using an installed Kokkos (2)

- A simple modulefile for Kokkos should at minimum set variable `CMAKE_PREFIX_PATH` pointing to the installed directory (the one which contains `KokkosConfig.cmake`
- **How to use Kokkos modulefiles on calypso ?** Just use the following:

```
# assuming CERFACS_TRAINING_DIR is where you clone the repository
# for companion code to this training
module use ${CERFACS_TRAINING_DIR}/kokkos/modulefiles
# e.g. load Kokkos for GPU
module load kokkos/4.4.01-cuda-12.4-gnu-12.3.0

# 4.4.01 is Kokkos version
# cuda-12.4 is cuda toolkit version
# 12.3.0 is g++ version
```

- **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/4.4.01-cuda-12.4-gnu-12.3.0
```

- See an example repo to store custom modulefiles: https://github.com/pkestene/mymodulefiles (most modulefiles comes with a readme to explain how to build tools)

Kokkos profiling interface (1)

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

```
# define path to the plugin
export KOKKOS_TOOLS_LIBS=/somewhere/kp_kernel_logger.so
# note: env var KOKKOS_PROFILE_LIBRARY is deprecated
# run as usal Kokkos application
```

- namespace `Kokkos::Tools` defines an interface, profiling hooks (i.e. functions pointers typedefs), and location where these hooks will be called (.e.g entering / exit a `parallel_for` region, etc...)
- Kokkos library is equipped with `hooks`, i.e. function pointers
  - if no profiling library loaded, nothing happend, zero overhead
  - if a profiling library is loaded, the functions pointers are *installed* during kokkos initialization, and the hooks executed at runtime
- Examples of Kokkos profile plugins can be found at https://github.com/kokkos/kokkos-tools largely independent of Kokkos config (which backend, etc...)

Kokkos profiling interface (2)

Automatic instrumentation of `parallel_for`, `deep_copy`, etc ...

- 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.
- see also https://github.com/kokkos/kokkos-tools/wiki

Kokkos profiling interface (3)

Explicit instrumentation:

```
// use push / pop
void foo() {
  Kokkos::Profiling::pushRegion("foo");
  bar();
  Kokkos::Profiling::popRegion();
}
// or use scoped region
{
  Kokkos::Profiling::ScopedRegion profReg("foo_and_bar");
  foo();
  bar();
}
```

Kokkos profiling interface (4)

List of tools:

- Utilities
  - KernelFilter: A tool which is used in conjunction with analysis tools, to restrict them to a subset of the application.
- Memory Analysis
  - MemoryHighWater: This tool outputs the high water mark of memory usage of the application. The high water mark of memory usage is the highest amount of memory that is being utilized during the application's execution.
  - MemoryUsage: Generates a per Memory Space timeline of memory utilization.
  - MemoryEvents: Tool to track memory events such as allocation and deallocation. It also provides the information of the MemoryUsage tool.
- Kernel Inspection
  - SimpleKernelTimer: Captures basic timing information for Kernels.
  - KernelLogger: Prints Kokkos Kernel and Region events during runtime.

Kokkos profiling interface (5)

Interface with 3rd party profiling tools, e.g. Nvidia Nsight Systems:
When using nvtx-connector, NVTX annotations will be added, e.g.

- `Kokkos::Profiling::pushRegion("foo");` will internally call `nvtxRangePush(name);`
- `Kokkos::Profiling::popRegion();` will internally call `nvtxRangePop();`



Kokkos for Fortran developpers

- OpenMP offload are OpenAcc are the first choice as programming model for refactoring large legacy Fortran application.
- Nevertheless there is a need to couple Fortran application, with C++ code that accelerator ready (e.g. by using Kokkos)
- Use KokkosTutorial_06_FortranPythonMPIAndPGAS.pdf to present Kokkos FLCL : Fortran Language Compatibility Layer
  - use code in `exercises/10-fortran/saxpy/kokkos-flcl`
- Present an alternative: plain C++/Kokkos + YAKL memory allocator and type wrapper + iso-c-bindings
  - use code in `exercises/10-fortran/saxpy/kokkos-yakl`
- Code example in climate sciences: SCREAM (Simple Cloud Resolving E3SM Atmosphere Model); won the 2023 Gordon Bell prize for climate modelling

Kokkos FLCL for beginners (1)

SAXPY example:

- 01-axpy-ndarray : only works on host, be careful
- 02-axpy-dualview : OK for GPU or CPU using `Kokkos::DualView`
- 03-axpy-view : OK for GPU or CPU using UVM (Unified Virtual Memory)
  a slightly variant of this exercise is proposed in `exercises/10-fortran/saxpy/kokkos-flcl`

Kokkos FLCL for beginners (2)

About UVM data types: from C++ side, internals, not needed from user point of view

```
// see flcl-cxx.hpp
#ifdef KOKKOS_ENABLE_CUDA
using HostMemorySpace = Kokkos::CudaUVMSpace;
#else
using HostMemorySpace = Kokkos::HostSpace;
#endif

// example view type; note usage of left layout for direct interoperability with Fortran
typedef Kokkos::View<flcl_view_r64_c_t*,Kokkos::LayoutLeft,flcl::HostMemorySpace> view_r64_1d_t;

// see flcl-cxx.hpp
// this is where the UVM view gets allocated
void c_kokkos_allocate_v_r64_1d(flcl::flcl_view_r64_c_t** A, flcl::view_r64_1d_t** v_A, const char* f
  const flcl::flcl_view_index_c_t e0t = std::max(*e0, view_index_one);
  std::string c_label( f_label );
  *v_A = (new flcl::view_r64_1d_t(c_label, e0t)); // placement new operator
  *A = (*v_A)->data();                            // pointer to data
}
```

Kokkos FLCL for beginners (2)

About UVM data types: from Fortran side, internals, not needed from user point of view

```
! see flcl-view-f.f90
module flcl_view_mod
  public view_r64_1d_t

  type view_r64_1d_t
    private
    type(c_ptr) :: handle
    contains
      procedure :: ptr => view_ptr_view_r64_1d_t
  end type view_r64_1d_t

  ! implementation of interface kokkos_allocate_view
  subroutine kokkos_allocate_v_r64_1d(A, v_A, n_A, e0)
    use, intrinsic :: iso_c_binding
    use flcl_util_strings_mod, only: char_add_null
    implicit none
    real(flcl_view_r64_f_t), pointer, dimension(:), intent(inout) :: A
    type(view_r64_1d_t), intent(out) :: v_A
    character(len=*), intent(in) :: n_A
    integer(flcl_view_index_f_t), intent(in) :: e0
    type(c_ptr) :: c_A

    character(len=:, kind=c_char), allocatable, target :: f_label

    call char_add_null( n_A, f_label )
    call f_kokkos_allocate_v_r64_1d(c_A, v_A%handle, f_label, e0)
    call c_f_pointer(c_A, A, shape=[e0])
  end subroutine kokkos_allocate_v_r64_1d


end module
```

Kokkos FLCL for beginners (3)

Example: saxpy using `type(view_r64_1d_t)` (on Fortran side), which are C++ `Kokkos::Views` wrapped types exercises/10-fortran/saxpy/kokkos-flcl/main.F90

```
!! kokkos view (actually UVM if e.g. Kokkos::Cuda or Kokkos::HIP activated)
type(view_r64_1d_t)                         :: x_view
type(view_r64_1d_t)                         :: y_view

!! pointers for C/Fortran interoperability
real(c_double), pointer, dimension(:)     :: x_ptr => null()
real(c_double), pointer, dimension(:)     :: y_ptr => null()

! initialize kokkos
write(*,*)'initializing kokkos'
call kokkos_initialize()

! allocate kokkos views (and host pointer)
call kokkos_allocate_view( y_ptr, y_view, 'y_ptr', int(length, c_size_t) )
call kokkos_allocate_view( x_ptr, x_view, 'x_ptr', int(length, c_size_t) )

! perform computation on host
call compute_saxpy(x_ptr, y_ptr, alpha)

! perform computation on device
call compute_saxpy_kokkos(x_view, y_view, alpha)
```

Kokkos + YAKL for beginners

- What is YAKL (Yet Another Kernel Library) : more or less the same goal as Kokkos (Performance Portability), lightweight, less generic, but additionnal interesting features for Fortran developpers
- short overview:
  - https://e3sm.org/wp-content/uploads/2022/03/220303_M_Norman.pdf
  - Portable C++ Code that can Look and Feel Like Fortran Code with Yet Another Kernel Launcher (YAKL), International Journal of Parallel Programming, vol 51, 209-230 (2023). Open Access.
- one interesting feature: the **memory allocator** called **gator**

YAKL UVM-based memory allocator (C++ and Fortran)

Why is **gator** interesting ?

- provide portable access to **UVM** (Unified Virtual Memory)
- `Fortran and Kokkos/YAKL use multidimensional array as primiary data container`
- Common practice, don't use `allocate` directly, but a wrapper
- ```
  ! original fortran allocation
  #define MY_MALLOC(ARR,SIZE) allocate(ARR SIZE)

  ! modified allocation : unified virtual memory (and portable, thanks to yakl)
  #define MY_MALLOC_MANAGED(array,size) call gator_allocate(array,size)
  ```
- `gatorAllocate` is a C++ routine defined in `YAKL` library (wrapped with `iso_c_bindings` as `gator_allocate`), it allows to call the *right* unified memory low-level allocator, i.e.
  - `cudaMallocManaged` (if Nvidia GPU),
  - `hipMallocManaged` (if AMD GPU),
  - `omp_target_associate_ptr` (if OpenMP target activated),
  - `acc_map_data` (if OpenAcc activated)
  ```

YAKL UVM-based memory allocator (C++ and Fortran)

YAKL allows to allocate, **on fortran side**, unified memory variables that are

1. visible as a multidimensional array on fortran side,
2. visible as GPU memory pointer on CUDA side,
3. visible as a multidimensional array on Kokkos side.

- $\Rightarrow$ **maximum flexibility and interoperability** (between new and legacy code)
- Unified memory also may require to add data prefetching at some locations, identified by profiling (Nvidia nsys).
- GPU variable life time management is really considerably simplified

Example: allocating a 2d array in UVM using ABI_MALLOC_MANAGED

```
! original code (CPU only):
integer, allocatable :: array(:,:)
MY_MALLOC(array,(dim1,dim2))

! refactored code (portable CPU/GPU)
! can be reused in pure Fortran subroutine (without interface change)
! can be passed to a Kokkos kernel call (wrapper via iso-c-bindings)
integer(kind=c_int32_t), contiguous, pointer :: array(:,:)
MY_MALLOC_MANAGED(array,(/dim1,dim2/))
```

YAKL UVM-based memory allocator (C++ and Fortran)

YAKL gator allocator internals (in Fortran)

```
subroutine gator_allocate_real8_1d( arr , dims , lbounds_in )
  integer, parameter :: ndims = 1
  real(8), pointer , intent(  out) :: arr       (:)
  integer          , intent(in   ) :: dims      (ndims)
  integer, optional, intent(in   ) :: lbounds_in(ndims)
  integer :: lbounds(ndims)
  type(c_ptr) :: data_ptr
  if (present(lbounds_in)) then
    lbounds = lbounds_in
  else
    lbounds = 1
  endif
  data_ptr = gator_allocate_c( int(product(dims)*sizeof(r8),c_size_t) ) ! here UVM mem. allocation
  call c_f_pointer( data_ptr , arr , dims )                             ! very similar to kokkos-flcl
  arr(lbounds(1):) => arr
end subroutine gator_allocate_real8_1d
```

YAKL + Kokkos SAXPY in Fortran

Example: exercises/10-fortran/saxpy/kokkos-yakl/main.F90 (this is an alternative to exercises/10-fortran/saxpy/kokkos-flcl/main.F90)

```
real(c_double), contiguous, pointer :: x(:) => null()
real(c_double), contiguous, pointer :: y(:) => null()

! initialize kokkos
call kokkos_initialize()

! initialize yakl memory allocator
call gator_init()

! allocate fortran array for both pure (CPU) computation and kokkos (CPU or GPU) computation
! less intrusive than Kokkos flcl type(view_r64_1d_t)
call gator_allocate(x,(/length/))
call gator_allocate(y,(/length/))

! pure (CPU) fortran computation
call compute_saxpy(x,y,alpha)

! Kokkos computation
call compute_saxpy_kokkos(C_LOC(x),C_LOC(y),length,alpha)

! ....
```

Using Kokkos in Fortran

Finally

- Using Kokkos + FLCL, the main data types are fortran wrapped `Kokkos::View`'s; plus additionnal Fortran pointer for pure CPU use
- Using Kokkos + YAKL, only one data type (a Fortran pointer allocated in UVM space); can be use in pure Fortran host code, or in Kokkos code:
  - no need to modify legacy Fortran code interface
  - maybe slightly more flexible than kokkos FLCL
- **time for hands-on**:
  - build and run the 2 alternatives (Kokkos-FLCL / Kokkos YAKL) on `calypso` for OpenMP and CUDA

Kokkos for python developpers

First, a refresher on how to interface C++ (and Cuda) with python

- How to proceed ? where to allocate memory (python side ? / C++ side ?)
- Automatic or manual binding ?
- which tool ? cython, swig, pybind11, nanobind, cppyy, ...
- How to integrate with your build system ?
- Finally discuss https://github.com/kokkos/pykokkos

CUDA and PYTHON

There are multiple ways to do GPU computing in python:

1. Drop-in replacement for simple kernels (with numpy interoperability)
   - numba, cupy, pycuda, legate/cuNumeric (introduced in 2021) [8]
2. Inlining CUDA kernels as strings + JIT compilation
   - requires CUDA/C++ knowledge
   - numba, cupy, pycuda
3. C/C++ extension
   - see https://docs.python.org/3/extending/extending.html
   - swig (a bit deprecated), cython, pybind11, cppyy
   - graalpython ? legate ?

ref: CUDA in your Python: Effective Parallel Programming on the GPU video on YouTube from Pytexas2019 conference.

[8] cupynumeric allows to do distributed computing on cluster of GPUs; see also https://github.com/NVIDIA/accelerated-computing-hub/blob/main/Accelerated_Python_User_Guide/notebooks/Chapter_X_Distributed_Computing_cuPyNumeric.ipynb

Using GPU hardware inside python app : CuPy

Using cupy as a drop-in replacement for numpy:

Python example on CPU with numpy:

```
import numpy as np
x = np.random.randn(10000000).astype(np.float32)
y = np.random.randn(10000000).astype(np.float32)
z = x + y
```

Using GPU hardware inside python app: CuPy

Using cupy as a drop-in replacement for numpy:

Python example on GPU with cupy

```
import cupy as cp
x = cp.random.randn(10000000).astype(np.float32)
y = cp.random.randn(10000000).astype(np.float32)
z = x + y
```

Using GPU hardware inside python app : Numba

What is Numba ?

- Translation of python functions to machine code at runtime using the LLVM compiler library
- Designed to be used with NumPy arrays
- Options to parallelize code for CPUs and GPUs and automatic SIMD Vectorization
- Support for both NVIDIA's CUDA and AMD's ROCm driver allowing to write parallel GPU code from Python.

Accelerate python app with Numba

Serial CPU version - pur python

```
def axpy(x,y,a):
    for i in range(x.shape[0]):
        x[i] = a*x[i] + y[i]
```

Accelerate python app with Numba

Serial CPU version - compiled to machine (no python interpreter)

```
import numba

@numba.jit(nopython=True)
def axpy(x,y,a):
    for i in range(x.shape[0]):
        x[i] = a*x[i] + y[i]
```

Accelerate python app with Numba

Parallel CPU version - multithreading + SIMD

range changed into prange

```
import numba

@numba.jit(nopython=True, parallel=True)
def axpy(x,y,a):
    for i in prange(x.shape[0]):
        x[i] = a*x[i] + y[i]
```

ref: https://github.com/numba/numba-examples/blob/master/notebooks/threads.ipynb

Parallel GPU version - CUDA

range changed into prange

```
import numba

@numba.cuda.jit('void(float32[:],float32[:])')
def axpy(x,y,a):
    i = cuda.grid(1)
    # i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    if i < x.shape[0]:
        x[i] = a*x[i] + y[i]
```

ref: https://github.com/numba/numba-examples/blob/master/notebooks/threads.ipynb

Numba reduction on CPU

```python
import numba

@numba.jit(nopython=True)
def reduce(x):
    x_sum = 0.0
    for i in range(x.shape[0]):
        x_sum += x[i]
    return x_sum
```

Numba reduction on GPU

```python
import numpy
from numba import cuda

@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (numpy.arange(1234, dtype=numpy.float64)) + 1
expect = A.sum()       # numpy sum reduction
got = sum_reduce(A)    # cuda sum reduction
assert expect == got
```

https://numba.pydata.org/numba-doc/dev/cuda/reduction.html

Accelerate python app with pycuda

Inlining Cuda/C++ as python string with pycuda

```
mod = SourceModule("""
    void __globall kernel_add_arrays(float *a, float *b, float *c, int N) {
      int gid = threadIdx.x + blockDim.x*blockIdx.x;
      while (gid < N) {
        c[gid] = a[gid] + b[gid];
        gid += blockDim.x*gridDim.x;
      }
    }
""")
```

then gets a callable object (this is where cuda kernel is compiled) for launching GPU computation

```
func = mod.get_function("kernel_add_arrays")
```

Additional resources

- template project for Cuda/python bindings using swig or cython : https://github.com/pkestene/npcuda-example
- template project for Cuda/python bindings using pybind11 and modern cmake : https://github.com/pkestene/pybind11-cuda
- python-hpc: https://github.com/eth-cscs/SummerUniversity2023/tree/main/python-hpc

Numba + Cupy

A very good starting point:
https://github.com/ContinuumIO/gtc2020-numba

1. install miniforge (= miniconda with conda-forge as default channel)

```
# use this on your own machine
mkdir -p ~/miniforge3
wget https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-Linux
bash ~/miniforge3/miniforge.sh -b -u -p ~/miniforge3
rm -rf ~/miniforge3/miniforge.sh
~/miniforge3/bin/conda init bash
# on calypso
module load python/gloenv3.12_arm
```

2. `conda install jupyter notebook`
3. install cupy (be sure to use pip from miniconda): `pip install pip install cupy-cuda101`
4. run `jupyter notebook` and open one of the tutorial notebook

Additionnal notes:

- By default, you don't need to install `cudatoolkit` from conda, if your Linux OS already has cuda in `/usr/local/cuda`

Numba snippets

minimal example in numba/cuda

```
import numpy as np
from numba import cuda

# create a CPU numpy array
arr = np.arange(1000)

# allocate a GPU array, and copy from host
d_arr = cuda.to_device(arr)

# cuda kernel launch
my_kernel[100, 100](d_arr)

# copy back results on host
result_array = d_arr.copy_to_host()
```

Numba snippets - device functions

Numba equivalent to `__device__ function` in cuda/c++:

device function in numba

```
from numba import cuda

@cuda.jit(device=True)
def a_device_function(a, b):
    return a + b
```

Reminder: device functions are functions that can only be call inside a CUDA kernel or inside another device function

Numba snippets - cuda kernels

CUDA kernel : sum of two 1D array

```
@cuda.jit
def max_example(a,b,c):
    """c = a + b"""
    tid = cuda.threadIdx.x
    bid = cuda.blockIdx.x
    bdim = cuda.blockDim.x

    start = (bid * bdim) + tid
    stride = cuda.blockDim.x * cuda.gridDim.x
    size = a.shape[0]

    for i in range(start, size, stride):
       c[i] = a[i] + b[i]
```

Numba snippets - cuda kernels for reduction

CUDA kernel : reduce example

```
"""https://numba.pydata.org/numba-doc/dev/cuda/reduction.html"""
@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (numpy.arange(1234, dtype=numpy.float64)) + 1
expect = A.sum()       # numpy sum reduction
got = sum_reduce(A)    # cuda sum reduction
assert expect == got
```

Others approach for CUDA/Python

E.g. in AI community:

- pytorch is using pybind11 to design extension modules written in C++ and CUDA, see https://pytorch.org/tutorials/advanced/cpp_extension.html
- tensorFlow is in evolving frow swig to pybind11 : https://github.com/tensorflow/community/blob/master/rfcs/20190208-pybind11.md

pykokkos-base and pykokkos

See article https://naderalawar.github.io/files/AlAwarETAL22PyKokkosTool.pdf

2 packages:

- pykokkos-base: minimal bindings for `Kokkos::initialize`, `Kokkos::deep_copy`, ..., `Kokkos::Views` (the maximun number of ranks must be chosen when installing pykokkos-base), ...; not really meant to be used by end-user. It is possible to use it, if you only wants to generate your own bindings for existing C++/Kokkos code. You need to be familliar with `pybind11`.
- pykokkos: high-level interface, based on pykokkos-base.
  From the readme: `PyKokkos translates type-annotated Python code into C++ Kokkos and automatically generating bindings for the translated C++ code.`

Building pykokkos-base and pykokkos

Using a slightly modified version of https://github.com/kokkos/pykokkos#readme

0. python env on calypso (`module load python/gloenv3.12_arm; conda init` [9]):

- ```
  git clone --recurse-submodules https://github.com/kokkos/pykokkos-base.git; cd pykokko
  ```
- ```
  conda create --name pyk --file requirements.txt python=3.11; conda activate pyk
  ```

1. build pykokkos-base on calypso:

- Mostly follow recipe at https://github.com/kokkos/pykokkos#readme with minor customization.
- `setup.py` is rooted to use `ninja` by default; on calypso your should prefer use `Unix Makefiles`

```
python setup.py install -- -DENABLE_LAYOUTS=ON -DENABLE_MEMORY_TRAITS=OFF -DENABLE_VIEW_RANKS=3 -DENABLE_CUDA=ON
-DENABLE_THREADS=OFF -DENABLE_OPENMP=ON -G "Unix Makefiles" -- -j 8
```

2. build pykokkos on calypso:

- ```
  cd ..; git clone https://github.com/kokkos/pykokkos.git; cd pykokkos
  ```
- ```
  conda install -c conda-forge pybind11 cupy patchelf
  ```
- ```
  pip install --user -e .
  ```

[9] must be done only once
  ```

Mini overview of pykokkos design

Documentation (be careful, the following is a bit **deprecated**):

- article : https://naderalawar.github.io/files/AlAwarETAL22PyKokkosTool.pdf
- (short) video : https://www.youtube.com/watch?v=1oFvhlhoDaY (watchout pkc has been removed, all examples can be run directly with python interpreter, it will trigger bindings generation and compilation automatically)
- just going through examples might be enough to capture the main features

pykokkos is a python framework that is:

- wrapping C++ Kokkos API with pybind11 $\Rightarrow$ **pykokkos-base**
- able to translate python code into c++/kokkos (using decorators) and to wrap generated C++ code into python, again via pybind11 $\Rightarrow$ **pykokkos**

Mini overview of pykokkos design (2)

2 slight variant programming styles:

1. defining **workunit** functions (very similar to C++/Kokkos lambda function)

```
import pykokkos as pk

# workunit is a decorator that allow to transform python function
# (or class member function) into kokkos/c++ kernel
# Note: types are explicit (to ease c++ translation)
@pk.workunit
def hello(i: int):
    pk.printf("Hello, World! from i = %d\n", i)

# run the kernel in the default execution space
# using a range policy with 10 iterations
pk.parallel_for(10, hello)
```

Mini overview of pykokkos design (3)

2 slight variant programming styles:

2. using **functor**

```
# A functor class can have multiple workunit's (as in c++ with tag dispatching)
@pk.functor
class Workload:
    def __init__(self, ....):

    @pk.workunit
    def do_work1(self, i: int):

    @pk.workunit
    def do_work2(self, i: int):

# define a range execution policy
p = pk.RangePolicy(pk.ExecutionSpace.OpenMP, 0, length)

# instantiate functor class
w = Workload(iterations, length, offset, scalar)

# e.g. launch parallel run of work1 workunit
pk.parallel_for(p, w.work1)
```

Mini overview of pykokkos design (4)

- **declaring kokkos views:**
  - `x: pk.View1D[pk.double] = pk.View([M], pk.double)`
  - `A: pk.View2D = pk.View([N, M], pk.double, layout=pk.Layout.LayoutRight)`
  - `y: pk.View1D[pk.double] = pk.View([M], pk.double, space=pk.MemorySpace.CudaSpace)`
  - Note : if pykokkos-base compiled c++/kokkos with `KOKKOS_ENABLE_CUDA_UVM` enabled, then Cuda UVM is the default memory space attached Cuda execution space

Mini overview of pykokkos design (5)

List of decorators used to annotate python to help C++/kokkos translation

- `pk.workload`: decorates a *workload* class, that must contain a method decorated by `pk.main` (where parallel dispatch is launched)
- `pk.main`: decorates a workload class member function, to be the entry of execution (lauched by `pk.execute`)

```
import pykokkos as pk

@pk.workload
class HelloWorld:
    def __init__(self, n):
        self.N: int = n

    @pk.main
    def run(self):
        pk.parallel_for(self.N, lambda i: pk.printf("Hello from i = %i\n", i))

if __name__ == "__main__":
    pk.execute(pk.ExecutionSpace.OpenMP, HelloWorld(10))
```

- `pk.workunit`
- `pk.functor`
- `pk.classtype`
- `pk.function`
- `pk.callback`

Mini overview of pykokkos design (5)

List of decorators used to annotate python to help C++/kokkos translation

- pk.workload
- pk.main
- pk.workunit: decorate a member function of a workload class, equivalent of `operator()` of a kokkos functor class in C++

```
import pykokkos as pk

@pk.workload
class HelloWorld:
    def __init__(self, n):
        self.N: int = n

    @pk.main
    def run(self):
        pk.parallel_for(self.N, self.hello)

    @pk.workunit
    def hello(self, i: int):
        pk.printf("Hello from i = %d\n", i)

if __name__ == "__main__":
    pk.execute(pk.ExecutionSpace.OpenMP, HelloWorld(10))
```

- pk.functor
- pk.classtype
- pk.function
- pk.callback

Mini overview of pykokkos design (5)

List of decorators used to annotate python to help C++/kokkos translation

- `pk.workload`
- `pk.main`
- `pk.workunit`
- `pk.functor`: decorates a class, very similar to `pk.worload`, except the class don't need a `pk.main` member; parallel dispatch is done in the calling code (this style of coding looks like C++)

```
import pykokkos as pk

@pk.functor
class SomeFunctor:
    def __init__(self, N: int):
        self.N: int = N
        self.data: pk.View1D[pk.double] = pk.View([N], pk.double)

    @pk.workunit
    def init(self, i: int):
        self.data[i] = 2.5 - 10 * i + i * i

def run() -> None:
    N: int = 10
    f = SomeFunctor(N)
    pk.parallel_for(pk.RangePolicy(0,N), f.init)
```

- `pk.classtype`
- `pk.function`
- `pk.callback`

Mini overview of pykokkos design (5)

List of decorators used to annotate python to help C++/kokkos translation

- `pk.workload`, `pk.main`, `pk.workunit`, `pk.functor`
- `pk.classtype`: a data class that can be instantiate on device
- `pk.function`: a function (either free function, or class member) that can be called inside a parallel region (i.e. inside a workunit)

```python
import pykokkos as pk

@pk.classtype
class TestClass:
    def __init__(self, x: float):
        self.x: float = x

    def test(self) -> float:
        return self.x * 2

@pk.workload
class Workload:
    def __init__(self, total_threads: int):
        self.total_threads: int = total_threads

    @pk.main
    def run(self) -> None:
        pk.parallel_for(self.total_threads, self.work)

    @pk.workunit
    def work(self, tid: int) -> None:
        pk.printf("%d\n", tid)

    @pk.function
    def fun(self, f: TestClass) -> None:
        f.x = 3
        x: float = f.x + 5

    @pk.function
    def test(self) -> TestClass:
        return TestClass(3.5)

if __name__ == "__main__":
    pk.execute(pk.ExecutionSpace.Default, Workload(10))
```

- `pk.callback`

Mini overview of pykokkos design (5)

List of decorators used to annotate python to help C++/kokkos translation

- `pk.workload`, `pk.main`, `pk.workunit`, `pk.functor`, `pk.classtype`, `pk.function`
- `pk.callback`: decorates a workload class member function, which is executed right after `pk.main` member; this a convenient to execute code afterwards, e.g for performing unit testing

```
import pykokkos as pk

@pk.workload
class SquareSum:
    def __init__(self, n):
        self.N: int = n
        self.total: pk.double = 0

    @pk.main
    def run(self):
        self.total = pk.parallel_reduce(self.N, self.squaresum)

    @pk.callback
    def results(self):
        true_sum = (self.N-1)*self.N*(2*self.N-1)/6
        if true_sum != self.total:
            print("Computation failed ! sum is {}".format(self.total))
        else:
            print("Computation passed ! sum is {}".format(self.total))

    @pk.workunit
    def squaresum(self, i: int, acc: pk.Acc[pk.double]):
        acc += i * i

if __name__ == "__main__":
    pk.execute(pk.ExecutionSpace.OpenMP, SquareSum(10))
```

Mini overview of pykokkos design (6)

Full example: a parallel reduction, you can e.g. chose exec space in main

```
import random

import pykokkos as pk

@pk.workload
class RandomSum:
    def __init__(self, n):
        self.N: int = n
        self.total: pk.int32 = 0
        self.a: pk.View1D[pk.int32] = pk.View([n], pk.int32)

        for i in range(self.N):
            self.a[i] = random.randint(0, 10)

        print("Initialized view:", self.a)

    @pk.main
    def run(self):
        self.total = pk.parallel_reduce(self.N, self.my_reduction)

    @pk.callback
    def results(self):
        print("Sum:", self.total)

    @pk.workunit
    def my_reduction(self, i: int, accumulator: pk.Acc[pk.int32]):
        accumulator += self.a[i]

if __name__ == "__main__":
    pk.set_default_space(pk.Cuda)
    n = 10
    pk.execute(pk.ExecutionSpace.Default, RandomSum(n))
```

Mini overview of pykokkos design (7)

How do pykokkos works ?

- At runtime, all decorators are applied and it triggers c++ code generation and python bindings
- a folder `pk_cpp` is created (in the run dire), and script `compile.sh` will compile into a shared library (.so) that can be loaded as a python module
- the whole process is automated (as with JIT mechanisms in e.g. numba)
- if a compilation error happens, it is advise to remove completely the `pk_cpp` before testing a fix (runing again the python script), to force the whole process to happens again

Mini overview of pykokkos design (8)

Known limitations (features not supported yet):

- `Kokkos::Reducer` $\Rightarrow$ currently pykokkos can only do sum reduction

Additionnal resources about python/hpc

- https://github.com/csc-training/hpc-python
- https://github.com/NVIDIA/accelerated-computing-hub/blob/main/Accelerated_Python_User_G

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

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`)

Kokkos resilience

https://github.com/kokkos/kokkos-resilience : Perform checkpoint/restart for application using kokkos data (e.g. Kokkos::View)

- Checkpoint/restor can be manual or automatic (support for several format: HDF5, ...)
- optional dependency: VeloC
- reference: Performance Portable and Productive Resilience Using Kokkos, https://www.osti.gov/servlets/purl/1766729
- Integrating process, control-flow, and data resiliency layers using a hybrid Fenix/Kokkos approach https://hal.science/hal-03772536/document

Additionnal links

- DOE-COE-Mtg-2016 / DOE-COE-Mtg-2017 : first DOE meetings on performance portability
- https://www.hpcwire.com/2016/04/19/compilers-makes-performance-portable/ : Compilers and More: What makes performance portable, Michael Wolfe (HPCWire article).
- Kokkos on github : https://github.com/kokkos/kokkos
- Kokkos new website : https://kokkos.org/
- programing guide : https://kokkos.github.io/kokkos-core-wiki/
- Kokkos tutorials: https://github.com/kokkos/kokkos-tutorials
- Kokkos Lecture slides and videos
- Kokkos slack channel: kokkosteam.slack.com
- list of applications using Kokkos : https://kokkos.org/applications/
- Kokkos 3: Programming Model Extensions for the Exascale Era, C. Trott et al., IEEE Transactions on Parallel and Distributed Systems (Vol. 33, Issue: 4, April 2022); https://doi.org/10.1109/TPDS.2021.3097283
- 2023 EuroTUG tutorial Day1 and Day2